eBook Gratuit

APPRENEZ C# Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#c#

Table des matières

À propos 1

Chapitre 1: Démarrer avec le langage C # 2

Remarques 2

Versions 2

Examples 2

Création d'une nouvelle application console (Visual Studio) 2

Explication 3

Utiliser la ligne de commande 3

Création d'un nouveau projet dans Visual Studio (application console) et exécution en mode 5

Créer un nouveau programme en utilisant Mono 9

Créer un nouveau programme en utilisant .NET Core 10

Sortie d'invite de commande 11

Création d'une nouvelle requête à l'aide de LinqPad 12

Création d'un nouveau projet à l'aide de Xamarin Studio 16

Chapitre 2: Accéder au dossier partagé du réseau avec le nom d'utilisateur et le mot de pa 23

Introduction 23

Examples 23

Code pour accéder au fichier partagé du réseau 23

Chapitre 3: Accéder aux bases de données 26

Examples 26

Connexions ADO.NET 26

Classes communes de fournisseur de données 26

Modèle d'accès commun pour les connexions ADO.NET 26

Connexions Entity Framework 27

Exécution de requêtes Entity Framework 28

Chaînes de connexion 28

Stockage de votre chaîne de connexion 29

Différentes connexions pour différents fournisseurs 29

Chapitre 4: Alias de types intégrés 30

Examples 30

Tableau des types intégrés 30
Chapitre 5: Analyse de regex 32
Syntaxe 32
Paramètres 32
Remarques 32
Examples 33
Match unique 33
Plusieurs correspondances 33
Chapitre 6: Annotation des données 34
Examples 34
DisplayNameAttribute (attribut d'affichage) 34
EditableAttribute (attribut de modélisation de données) 35
Attributs de validation 37
Exemple: RequiredAttribute 37
Exemple: StringLengthAttribute 37
Exemple: RangeAttribute 37
Exemple: CustomValidationAttribute 38
Création d'un attribut de validation personnalisé 38
Bases d'annotation des données 39
Usage 39
Exécuter manuellement les attributs de validation 39
Contexte de validation 40
Valider un objet et toutes ses propriétés 40
Valider une propriété d'un objet 40
Et plus 40
Chapitre 7: Arbres d'expression 41
Introduction 41
Syntaxe 41
Paramètres 41
Remarques 41
Introduction aux arbres d'expression 41

D'où nous venons 41
Comment éviter les problèmes de mémoire et de latence de l'inversion de flux 41
Les arbres d'expression sauvent la journée 42
Créer des arbres d'expression 42
Arbres d'expression et LINQ 43
Remarques 43
Examples 43
Création d'arbres d'expression à l'aide de l'API 43
Compilation d'arbres d'expression 44
Analyse d'arbres d'expression 44
Créer des arbres d'expression avec une expression lambda 44
Comprendre l'API des expressions 45
Arbre d'expression de base 45
Examen de la structure d'une expression à l'aide de Visitor 46
Chapitre 8: Arguments nommés 48
Examples 48
Les arguments nommés peuvent rendre votre code plus clair 48
Arguments nommés et paramètres facultatifs 48
L'ordre des arguments n'est pas nécessaire 49
Les arguments nommés évitent les bogues sur les paramètres facultatifs 49
Chapitre 9: Arguments nommés et facultatifs 51
Remarques 51
Examples 51
Arguments nommés 51
Arguments optionnels 53
Chapitre 10: AssemblyInfo.cs Exemples 56
Remarques 56
Examples 56
[AssemblyTitle] 56
[AssemblyProduct] 56
Global et local AssemblyInfo 56
[AssemblyVersion] 57

Lecture des attributs d'assemblage ... 57
Contrôle de version automatisé ... 57
Champs communs ... 58
[AssemblyConfiguration] ... 58
[InternalsVisibleTo] ... 58
[AssemblyKeyFile] ... 59
Chapitre 11: Async / wait, Backgroundworker, Exemples de tâches et de threads ... 60
Remarques ... 60
Examples ... 60
ASP.NET Configure Await ... 60
Blocage ... 60
ConfigureAwait ... 61
Async / attente ... 62
BackgroundWorker ... 63
Tâche ... 64
Fil ... 65
Tâche "exécuter et oublier" extension ... 66
Chapitre 12: Async-Await ... 67
Introduction ... 67
Remarques ... 67
Examples ... 67
Appels consécutifs simples ... 67
Try / Catch / Finalement ... 67
Configuration de Web.config sur la cible 4.5 pour un comportement asynchrone correct ... 68
Appels simultanés ... 69
Attendez l'opérateur et le mot-clé asynchrone ... 70
Retourner une tâche sans attendre ... 71
Le blocage sur du code asynchrone peut provoquer des blocages ... 72
Async / wait n'améliorera les performances que si elle permet à la machine d'effectuer des ... 73
Chapitre 13: BackgroundWorker ... 75
Syntaxe ... 75
Remarques ... 75

Examples ... 75
Affectation de gestionnaires d'événements à un BackgroundWorker ... 75
Affectation de propriétés à un BackgroundWorker ... 76
Création d'une nouvelle instance BackgroundWorker ... 76
Utiliser un BackgroundWorker pour effectuer une tâche ... 77
Le résultat est le suivant 78
Chapitre 14: Bibliothèque parallèle de tâches ... 79
Examples ... 79
Parallel.ForEach ... 79
Parallel.For ... 79
Parallel.Invoke ... 80
Une interrogation asynchrone annulable Tâche qui attend entre les itérations ... 80
Une tâche d'interrogation annulable à l'aide de CancellationTokenSource ... 81
Version asynchrone de PingUrl ... 82
Chapitre 15: BigInteger ... 83
Remarques ... 83
Quand utiliser ... 83
Des alternatives ... 83
Examples ... 83
Calculez le premier nombre de Fibonacci à 1 000 chiffres ... 83
Chapitre 16: BindingList ... 85
Examples ... 85
Eviter l'itération N * 2 ... 85
Ajouter un article à la liste ... 85
Chapitre 17: C # Script ... 86
Examples ... 86
Évaluation de code simple ... 86
Chapitre 18: Chronomètres ... 87
Syntaxe ... 87
Remarques ... 87
Examples ... 87
Créer une instance d'un chronomètre ... 87

IsHighResolution ... 87
Chapitre 19: Classe partielle et méthodes ... 89
Introduction ... 89
Syntaxe ... 89
Remarques ... 89
Examples ... 89
Classes partielles ... 89
Méthodes partielles ... 90
Classes partielles héritant d'une classe de base ... 91
Chapitre 20: Classes statiques ... 92
Examples ... 92
Mot-clé statique ... 92
Classes statiques ... 92
Durée de vie de la classe statique ... 93
Chapitre 21: CLSCompliantAttribute ... 95
Syntaxe ... 95
Paramètres ... 95
Remarques ... 95
Examples ... 95
Modificateur d'accès auquel les règles CLS s'appliquent ... 95
Violation de la règle CLS: types non signés / sbyte ... 96
Violation de la règle CLS: même dénomination ... 97
Violation de la règle CLS: Identifiant _ ... 97
Violation de la règle CLS: hériter de la classe non CLSComplaint ... 98
Chapitre 22: Code non sécurisé dans .NET ... 99
Remarques ... 99
Examples ... 99
Index de tableau dangereux ... 99
Utiliser des tableaux non sécurisés ... 100
Utilisation de unsafe avec des chaînes ... 100
Chapitre 23: Comment utiliser les structures C # pour créer un type d'union (similaire aux ... 102
Remarques ... 102

Examples ... 102
Unions de style C en C # ... 102
Les types d'union en C # peuvent également contenir des champs Struct ... 103
Chapitre 24: Commentaires et régions ... 105
Examples ... 105
commentaires ... 105
Commentaires sur une seule ligne ... 105
Multi lignes ou commentaires délimités ... 105
Les régions ... 106
Commentaires sur la documentation ... 107
Chapitre 25: Commentaires sur la documentation XML ... 109
Remarques ... 109
Examples ... 109
Annotation de méthode simple ... 109
Commentaires sur la documentation de l'interface et de la classe ... 109
Commentaire sur la documentation de la méthode avec les éléments param et return ... 110
Générer du code XML à partir de commentaires de documentation ... 111
Référencement d'une autre classe dans la documentation ... 112
Chapitre 26: Concaténation de cordes ... 114
Remarques ... 114
Examples ... 114
+ Opérateur ... 114
Concaténer des chaînes à l'aide de System.Text.StringBuilder ... 114
Concat éléments de tableau de chaînes à l'aide de String.Join ... 114
Concaténation de deux chaînes en utilisant $ ... 115
Chapitre 27: Constructeurs et finaliseurs ... 116
Introduction ... 116
Remarques ... 116
Examples ... 116
Constructeur par défaut ... 116
Appeler un constructeur d'un autre constructeur ... 117
Constructeur statique ... 118

Appel du constructeur de la classe de base 119
Finalisateurs sur les classes dérivées 120
Modèle de constructeur singleton 120
Forcer un constructeur statique à être appelé 121
Appeler des méthodes virtuelles dans un constructeur 121
Constructeurs statiques génériques 122
Exceptions dans les constructeurs statiques 123
Initialisation du constructeur et de la propriété 124
Chapitre 28: Constructions de flux de données TPL (Task Parallel Library) 126
Examples 126
JoinBlock 126
BroadcastBlock 127
WriteOnceBlock 128
BatchedJoinBlock 129
TransformBlock 130
ActionBlock 130
TransformManyBlock 131
BatchBlock 132
BufferBlock 133
Chapitre 29: Contexte de synchronisation dans Async-Await 135
Examples 135
Pseudocode pour asynchrone / en attente de mots-clés 135
Désactivation du contexte de synchronisation 135
Pourquoi SynchronizationContext est-il si important? 136
Chapitre 30: Contrats de code 138
Syntaxe 138
Remarques 138
Examples 138
Conditions préalables 139
Postconditions 139
Invariants 139
Définition de contrats sur l'interface 140

Chapitre 31: Contrats de code et assertions ... 143
Examples ... 143
Les affirmations pour vérifier la logique doivent toujours être vraies ... 143
Chapitre 32: Conventions de nommage ... 145
Introduction ... 145
Remarques ... 145
Choisissez des noms d'identifiant facilement lisibles ... 145
Favoriser la lisibilité sur la brièveté ... 145
N'utilisez pas de notation hongroise ... 145
Abréviations et acronymes ... 145
Examples ... 145
Conventions de capitalisation ... 145
Pascal Casing ... 146
Camel Casing ... 146
Majuscule ... 146
Règles ... 146
Interfaces ... 147
Champs privés ... 147
Affaire de chameau ... 147
Housse camel avec soulignement ... 147
Espaces de noms ... 148
Enums ... 148
Utilisez un nom singulier pour la plupart des énumérations ... 148
Utilisez un nom pluriel pour les types Enum qui sont des champs de bits ... 148
Ne pas ajouter 'enum' comme suffixe ... 149
N'utilisez pas le nom enum dans chaque entrée ... 149
Des exceptions ... 149
Ajouter 'exception' comme suffixe ... 149
Chapitre 33: Conversion de type ... 150
Remarques ... 150
Examples ... 150

Exemple d'opérateur implicite MSDN 150
Conversion de type explicite 151
Chapitre 34: Cordes Verbatim 152
Syntaxe 152
Remarques 152
Examples 152
Cordes multilignes 152
Échapper à des citations doubles 153
Cordes Verbatim Interpolées 153
Les chaînes verbatim indiquent au compilateur de ne pas utiliser les caractères d'échappem 153
Chapitre 35: Courant 155
Examples 155
Utiliser des flux 155
Chapitre 36: Création d'une application console à l'aide d'un éditeur de texte brut et du 157
Examples 157
Création d'une application console à l'aide d'un éditeur de texte brut et du compilateur C 157
Sauvegarder le code 157
Compiler le code source 157
Chapitre 37: Créer son propre MessageBox dans l'application Windows Form 160
Introduction 160
Syntaxe 160
Examples 160
Création du propre contrôle MessageBox 160
Comment utiliser son propre contrôle MessageBox créé dans une autre application Windows Fo 162
Chapitre 38: Cryptographie (System.Security.Cryptography) 164
Examples 164
Exemples modernes de chiffrement authentifié symétrique d'une chaîne 164
Introduction au chiffrement symétrique et asymétrique 175
Chiffrement symétrique 176
Chiffrement asymétrique 176
Hachage de mot de passe 177

Cryptage de fichiers symétrique simple ... 178
Données aléatoires sécurisées par cryptographie ... 178
Chiffrement de fichiers asymétrique rapide ... 179
Chapitre 39: Débordement ... 185
Examples ... 185
Débordement d'entier ... 185
Débordement pendant le fonctionnement ... 185
Questions de commande ... 185
Chapitre 40: Déclaration de verrouillage ... 187
Syntaxe ... 187
Remarques ... 187
Examples ... 188
Usage simple ... 188
Lancer une exception dans une déclaration de verrouillage ... 188
Retourner dans une déclaration de verrouillage ... 189
Utilisation d'instances d'objet pour le verrouillage ... 189
Anti-Patterns et Gotchas ... 189
Verrouillage sur une variable allouée / locale ... 189
En supposant que le verrouillage restreint l'accès à l'objet de synchronisation lui-même ... 190
Attend des sous-classes pour savoir quand verrouiller ... 190
Le verrouillage sur une variable ValueType en boîte ne se synchronise pas ... 192
Utiliser des verrous inutilement lorsqu'une alternative plus sûre existe ... 193
Chapitre 41: Délégués Func ... 195
Syntaxe ... 195
Paramètres ... 195
Examples ... 195
Sans paramètres ... 195
Avec plusieurs variables ... 196
Méthodes Lambda & Anonymous ... 196
Paramètres de type covariant et contre-polarisant ... 197
Chapitre 42: Des minuteries ... 198

Syntaxe ... 198
Remarques ... 198
Examples ... 198
Temporisateurs Multithread ... 198
Caractéristiques: ... 199
Créer une instance d'un temporisateur ... 200
Affectation du gestionnaire d'événements "Tick" à une minuterie ... 200
Exemple: utiliser une minuterie pour effectuer un simple compte à rebours ... 201
Chapitre 43: Diagnostic ... 203
Examples ... 203
Debug.WriteLine ... 203
Redirection de la sortie du journal avec TraceListeners ... 203
Chapitre 44: Directives du préprocesseur ... 204
Syntaxe ... 204
Remarques ... 204
Expressions conditionnelles ... 204
Examples ... 205
Expressions conditionnelles ... 205
Génération des avertissements et des erreurs du compilateur ... 206
Définition et suppression de symboles ... 206
Blocs de région ... 207
Autres instructions du compilateur ... 207
Ligne ... 207
Somme de contrôle Pragma ... 208
Utilisation de l'attribut conditionnel ... 208
Désactiver et restaurer les avertissements du compilateur ... 208
Préprocesseurs personnalisés au niveau du projet ... 209
Chapitre 45: En boucle ... 211
Examples ... 211
Styles de boucle ... 211
Pause ... 212
Boucle Foreach ... 213

En boucle ... 214
Pour boucle ... 214
Boucle Do-While ... 215
Boucles imbriquées ... 216
continuer ... 216
Chapitre 46: Enum ... 217
Introduction ... 217
Syntaxe ... 217
Remarques ... 217
Examples ... 217
Obtenez toutes les valeurs de membres d'un enum ... 217
Enum comme drapeaux ... 218
Tester les valeurs d'énumération de style avec la logique binaire ... 220
Enum pour enchaîner et revenir ... 220
Valeur par défaut pour enum == ZERO ... 221
Bases d'énum ... 222
Manipulation binaire utilisant des énumérations ... 223
Utiliser la notation pour les drapeaux ... 223
Ajout d'informations de description supplémentaires à une valeur enum ... 224
Ajouter et supprimer des valeurs de l'énumération marquée ... 225
Enums peuvent avoir des valeurs inattendues ... 225
Chapitre 47: Est égal à et GetHashCode ... 227
Remarques ... 227
Examples ... 227
Comportement par défaut égal ... 227
Écrire un bon remplacement GetHashCode ... 228
Remplacer Equals et GetHashCode sur les types personnalisés ... 229
Est égal à et GetHashCode dans IEqualityComparator ... 230
Chapitre 48: Événements ... 232
Introduction ... 232
Paramètres ... 232
Remarques ... 232

Examples ... 233
Déclarer et soulever des événements ... 233
Déclarer un événement ... 233
Élever l'événement ... 234
Déclaration d'événement standard ... 234
Déclaration de gestionnaire d'événement anonyme ... 235
Déclaration d'événement non standard ... 236
Création de EventArgs personnalisés contenant des données supplémentaires ... 236
Créer un événement annulable ... 238
Propriétés de l'événement ... 239
Chapitre 49: Exécution de requêtes HTTP ... 241
Examples ... 241
Création et envoi d'une requête HTTP POST ... 241
Création et envoi d'une requête HTTP GET ... 241
Gestion des erreurs de codes de réponse HTTP spécifiques (tels que 404 introuvable) ... 242
Envoi d'une requête HTTP POST asynchrone avec un corps JSON ... 242
Envoi de requête HTTP GET asynchrone et lecture de requête JSON ... 243
Récupérer le code HTML pour la page Web (simple) ... 243
Chapitre 50: Expressions conditionnelles ... 244
Examples ... 244
Déclaration If-Else ... 244
Déclaration If-Else If-Else ... 244
Changer les déclarations ... 245
Si les conditions de déclaration sont des expressions et des valeurs booléennes standard ... 246
Chapitre 51: Expressions lambda ... 248
Remarques ... 248
Examples ... 248
Passer une expression Lambda en tant que paramètre à une méthode ... 248
Expressions Lambda comme raccourci pour l'initialisation des délégués ... 248
Lambdas pour `Func` et` Action` ... 248
Expressions lambda avec plusieurs paramètres ou aucun paramètre ... 249
Mettre plusieurs instructions dans une déclaration Lambda ... 249

Les Lambdas peuvent être émises à la fois comme `Func` et` Expression` ... 249
Expression Lambda en tant que gestionnaire d'événements ... 250
Chapitre 52: Expressions lambda ... 252
Remarques ... 252
Fermetures ... 252
Examples ... 252
Expressions lambda de base ... 252
Expressions lambda de base avec LINQ ... 253
Utiliser la syntaxe lambda pour créer une fermeture ... 253
Syntaxe Lambda avec corps de bloc d'instructions ... 254
Expressions lambda avec System.Linq.Expressions ... 254
Chapitre 53: Extensions Réactives (Rx) ... 255
Examples ... 255
Observation de l'événement TextChanged sur une zone de texte ... 255
Diffusion de données à partir d'une base de données avec Observable ... 255
Chapitre 54: Fichier et flux I / O ... 257
Introduction ... 257
Syntaxe ... 257
Paramètres ... 257
Remarques ... 257
Examples ... 258
Lecture d'un fichier à l'aide de la classe System.IO.File ... 258
Écriture de lignes dans un fichier à l'aide de la classe System.IO.StreamWriter ... 258
Écriture dans un fichier à l'aide de la classe System.IO.File ... 259
Lentement lire un fichier ligne par ligne via un IEnumerable ... 259
Créer un fichier ... 259
Copier un fichier ... 260
Déplacer un fichier ... 260
Supprimer le fichier ... 261
Fichiers et répertoires ... 261
Async écrire du texte dans un fichier à l'aide de StreamWriter ... 261
Chapitre 55: FileSystemWatcher ... 262

Syntaxe ... 262
Paramètres ... 262
Examples ... 262
FileWatcher de base ... 262
IsFileReady ... 263
Chapitre 56: Filetage ... 264
Remarques ... 264
Examples ... 265
Démonstration simple et complète ... 265
Démo de threading simple et complète à l'aide de tâches ... 265
Parallélisation des tâches explicites ... 266
Parallélisme implicite des tâches ... 266
Créer et démarrer un deuxième thread ... 266
Commencer un thread avec des paramètres ... 267
Création d'un thread par processeur ... 267
Éviter de lire et d'écrire des données simultanément ... 267
Parallel.ForEach Loop ... 269
Deadlocks (deux threads en attente sur l'autre) ... 269
Deadlocks (maintenir la ressource et attendre) ... 271
Chapitre 57: Filtres d'action ... 274
Examples ... 274
Filtres d'action personnalisés ... 274
Chapitre 58: Fonction avec plusieurs valeurs de retour ... 276
Remarques ... 276
Examples ... 276
"objet anonyme" + solution "mot clé dynamique" ... 276
Solution tuple ... 276
Paramètres Ref et Out ... 277
Chapitre 59: Fonctionnalités C # 3.0 ... 278
Remarques ... 278
Examples ... 278
Variables typées implicitement (var) ... 278

Requêtes linguistiques intégrées (LINQ) ... 278
Expression Lambda ... 279
Types anonymes ... 280
Chapitre 60: Fonctionnalités C # 5.0 ... 282
Syntaxe ... 282
Paramètres ... 282
Remarques ... 282
Examples ... 282
Async & Attente ... 282
Informations sur l'appelant ... 284
Chapitre 61: Fonctionnalités C # 7.0 ... 285
Introduction ... 285
Examples ... 285
déclaration var out ... 285
Exemple ... 285
Limites ... 286
Les références ... 287
Littéraux binaires ... 287
Énumérations de drapeaux ... 287
Séparateurs de chiffres ... 288
Prise en charge linguistique pour Tuples ... 288
Les bases ... 288
Déconstruction de Tuple ... 289
Initialisation du tuple ... 291
h11 ... 291
Type d'inférence ... 291
Noms de champ de réflexion et de tuple ... 291
Utiliser avec des génériques et async ... 292
Utiliser avec des collections ... 292
Différences entre ValueTuple et Tuple ... 293

Les références 293
Fonctions locales 293
Exemple 293
Exemple 294
Exemple 294
Correspondance de motif 295
switch expression 295
is expression 296
Exemple 296
ref retour et ref local 297
Ref Return 297
Ref Local 297
Opérations de référence non sécurisées 297
Liens 298
jeter des expressions 299
Expression étendue liste des membres corporels 299
ValueTask 300
1. Augmentation de la performance 300
2. Flexibilité accrue de la mise en œuvre 301
Implémentation synchrone: 301
Implémentation asynchrone 301
Remarques 302
Chapitre 62: Fonctions C # 4.0 303
Examples 303
Paramètres facultatifs et arguments nommés 303
Variance 304
Mot clé ref optionnel lors de l'utilisation de COM 304
Recherche de membre dynamique 304
Chapitre 63: Fonctions C # 6.0 306
Introduction 306

Remarques 306
Examples 306
Nom de l'opérateurde 306
Solution de contournement pour les versions précédentes (plus de détails) 307
Membres de la fonction avec expression 308
Propriétés 308
Indexeurs 309
Les méthodes 309
Les opérateurs 310
Limites 310
Filtres d'exception 311
Utilisation de filtres d'exception 311
Article risqué quand 312
Enregistrement comme effet secondaire 313
Le bloc finally 314
Exemple: bloc finally 314
Initialiseurs de propriétés automatiques 316
introduction 316
Accesseurs avec une visibilité différente 316
Propriétés en lecture seule 316
Vieux style (pre C # 6.0) 316
Usage 317
Notes de mise en garde 318
Initialiseurs d'index 319
Interpolation de chaîne 321
Exemple de base 321
Utilisation de l'interpolation avec des littéraux de chaîne textuels 321
Expressions 322
Séquences d'échappement 323
Type de chaîne formatée 324

Conversions implicites 325
Méthodes de culture actuelles et invariantes 325
Dans les coulisses 326
Interpolation de chaînes et Linq 326
Cordes interpolées réutilisables 326
Interpolation et localisation de chaînes 327
Interpolation récursive 328
Attendez dans la prise et enfin 328
Propagation nulle 329
Les bases 330
Utiliser avec l'opérateur Null-Coalescing (??) 331
Utiliser avec des indexeurs 331
Utiliser avec fonctions vides 331
Utiliser avec l'invocation d'événement 331
Limites 332
Gotchas 332
En utilisant le type statique 333
Amélioration de la résolution de la surcharge 334
Changements mineurs et corrections de bugs 335
Utilisation d'une méthode d'extension pour l'initialisation de la collection 336
Désactiver les améliorations des avertissements 337
Chapitre 64: Fonctions de hachage 338
Remarques 338
Examples 338
MD5 338
SHA1 339
SHA256 339
SHA384 340
SHA512 340
PBKDF2 pour hachage de mot de passe 341
Solution complète de hachage de mot de passe avec Pbkdf2 342

Chapitre 65: Fonderie ... 346
Remarques ... 346
Examples ... 346
Lancer un objet sur un type de base ... 346
Casting explicite ... 347
Casting explicite sûr (opérateur `as`) ... 347
Casting implicite ... 347
Vérification de la compatibilité sans coulée ... 347
Conversions numériques explicites ... 348
Opérateurs de conversion ... 348
Opérations de coulée LINQ ... 349
Chapitre 66: Garbage Collector dans .Net ... 351
Examples ... 351
Compactage de tas de gros objets ... 351
Références faibles ... 351
Chapitre 67: Générateur de requêtes Lambda générique ... 354
Remarques ... 354
Examples ... 354
Classe QueryFilter ... 354
Méthode GetExpression ... 355
GetExpression Surcharge privée ... 356
Pour un filtre: ... 356
Pour deux filtres: ... 357
Méthode ConstantExpression ... 357
Usage ... 358
Sortie: ... 358
Chapitre 68: Génération de code T4 ... 359
Syntaxe ... 359
Examples ... 359
Génération de code d'exécution ... 359
Chapitre 69: Génération de nombres aléatoires en C # ... 360

Syntaxe ... 360
Paramètres ... 360
Remarques ... 360
Examples ... 360
Générer un int aléatoire ... 360
Générer un double aléatoire ... 361
Générer un int aléatoire dans une plage donnée ... 361
Générer la même séquence de nombres aléatoires encore et encore ... 361
Créer plusieurs classes aléatoires avec différentes graines simultanément ... 361
Générer un caractère aléatoire ... 362
Générer un nombre qui est un pourcentage d'une valeur maximale ... 362
Chapitre 70: Génériques ... 363
Syntaxe ... 363
Paramètres ... 363
Remarques ... 363
Examples ... 363
Paramètres de type (classes) ... 363
Paramètres de type (méthodes) ... 364
Paramètres de type (interfaces) ... 364
Inférence de type implicite (méthodes) ... 365
Contraintes de type (classes et interfaces) ... 366
Contraintes de type (class et struct) ... 367
Contraintes de type (new-mot-clé) ... 368
Type d'inférence (classes) ... 368
Réflexion sur les paramètres de type ... 369
Paramètres de type explicites ... 369
Utiliser une méthode générique avec une interface en tant que type de contrainte ... 370
Covariance ... 371
Contravariance ... 373
Invariance ... 373
Interfaces de variantes ... 374
Variantes de délégués ... 375

Types de variantes en tant que paramètres et valeurs de retour 375
Vérification de l'égalité des valeurs génériques 376
Casting de type générique 376
Lecteur de configuration avec conversion de type générique 377
Chapitre 71: Gestion de FormatException lors de la conversion d'une chaîne en d'autres typ 379
Examples 379
Conversion de chaîne en entier 379
Chapitre 72: Gestion des exceptions 381
Examples 381
Gestion des exceptions de base 381
Gestion de types d'exceptions spécifiques 381
Utiliser l'objet exception 381
Bloquer enfin 383
Implémentation d'IErrorHandler pour les services WCF 384
Création d'exceptions personnalisées 387
Création d'une classe d'exception personnalisée 387
lancer à nouveau 388
sérialisation 388
Utiliser l'Exception Parser 388
Problèmes de sécurité 389
Conclusion 390
Exception Anti-patrons 390
Exceptions à la déglutition 390
Gestion des exceptions de baseball 391
attraper (Exception) 391
Exceptions globales / exceptions multiples d'une méthode 392
Nidification des exceptions et essayez les blocs de capture 393
Les meilleures pratiques 394
Cheatsheet 394
NE PAS gérer la logique métier avec des exceptions 394
NE PAS rejeter les exceptions 396

NE PAS absorber les exceptions sans enregistrement ... 396
Ne pas intercepter les exceptions que vous ne pouvez pas gérer ... 396
Exception non gérée et thread ... 397
Lancer une exception ... 398
Chapitre 73: Gestionnaire d'authentification C # ... 399
Examples ... 399
Gestionnaire d'authentification ... 399
Chapitre 74: Guid ... 401
Introduction ... 401
Remarques ... 401
Examples ... 401
Obtenir la représentation sous forme de chaîne d'un Guid ... 401
Créer un guide ... 402
Déclaration d'un GUID nullable ... 402
Chapitre 75: Héritage ... 403
Syntaxe ... 403
Remarques ... 403
Examples ... 403
Hériter d'une classe de base ... 403
Héritage d'une classe et implémentation d'une interface ... 404
Hériter d'une classe et implémenter plusieurs interfaces ... 404
Tester et naviguer dans l'héritage ... 405
Extension d'une classe de base abstraite ... 406
Constructeurs dans une sous-classe ... 406
Héritage. Séquence d'appels des constructeurs ... 407
Méthodes d'héritage ... 409
Héritage Anti-patrons ... 410
Héritage incorrect ... 410
Classe de base avec spécification de type récursif ... 411
Chapitre 76: IClonable ... 415
Syntaxe ... 415
Remarques ... 415

Examples ... 415
Implémenter ICloneable dans une classe ... 415
Implémenter ICloneable dans une structure ... 416
Chapitre 77: IComparable ... 418
Examples ... 418
Trier les versions ... 418
Chapitre 78: Identité ASP.NET ... 420
Introduction ... 420
Examples ... 420
Comment implémenter le jeton de réinitialisation de mot de passe dans l'identité asp.net à ... 420
Chapitre 79: IEnumerable ... 424
Introduction ... 424
Remarques ... 424
Examples ... 424
IEnumerable ... 424
IEnumerable avec un énumérateur personnalisé ... 424
Chapitre 80: ILGenerator ... 426
Examples ... 426
Crée un DynamicAssembly contenant une méthode d'assistance UnixTimestamp ... 426
Créer une substitution de méthode ... 428
Chapitre 81: Immutabilité ... 429
Examples ... 429
Classe System.String ... 429
Cordes et immuabilité ... 429
Chapitre 82: Importer les contacts Google ... 431
Remarques ... 431
Examples ... 431
Exigences ... 431
Code source dans le contrôleur ... 431
Code source dans la vue ... 434
Chapitre 83: Indexeur ... 435

Syntaxe ... 435
Remarques ... 435
Examples ... 435
Un indexeur simple ... 435
Indexeur avec 2 arguments et interface ... 435
Surcharger l'indexeur pour créer un SparseArray ... 436
Chapitre 84: Initialisation des propriétés ... 437
Remarques ... 437
Examples ... 437
C # 6.0: initialiser une propriété implémentée automatiquement ... 437
Initialisation d'une propriété avec un champ de sauvegarde ... 437
Initialisation de la propriété dans le constructeur ... 437
Initialisation de la propriété pendant l'instanciation de l'objet ... 437
Chapitre 85: Initialiseurs d'objets ... 439
Syntaxe ... 439
Remarques ... 439
Examples ... 439
Usage simple ... 439
Utilisation avec des types anonymes ... 439
Utilisation avec des constructeurs autres que ceux par défaut ... 440
Chapitre 86: Initialiseurs de collection ... 441
Remarques ... 441
Examples ... 441
Initialiseurs de collection ... 441
Initialiseurs d'index C # 6 ... 442
Initialisation du dictionnaire ... 442
Initialiseurs de collection dans les classes personnalisées ... 443
Initialiseurs de collections avec tableaux de paramètres ... 444
Utilisation de l'initialiseur de collection à l'intérieur de l'initialiseur d'objet ... 444
Chapitre 87: Injection de dépendance ... 446
Remarques ... 446

Examples ... 446
Injection de dépendance à l'aide de MEF ... 446
Injection de dépendances C # et ASP.NET avec Unity ... 448
Chapitre 88: Interface IDisposable ... 452
Remarques ... 452
Examples ... 452
Dans une classe contenant uniquement des ressources gérées ... 452
Dans une classe avec des ressources gérées et non gérées ... 452
IDisposable, Dispose ... 453
Dans une classe héritée avec des ressources gérées ... 454
en utilisant le mot clé ... 454
Chapitre 89: Interface INotifyPropertyChanged ... 456
Remarques ... 456
Examples ... 456
Implémenter INotifyPropertyChanged en C # 6 ... 456
INotifyPropertyChanged Avec la méthode d'ensemble générique ... 457
Chapitre 90: Interface IQueryable ... 459
Examples ... 459
Traduire une requête LINQ en requête SQL ... 459
Chapitre 91: Interfaces ... 460
Examples ... 460
Implémenter une interface ... 460
Implémentation de plusieurs interfaces ... 460
Implémentation d'interface explicite ... 461
Allusion: ... 462
Remarque: ... 462
Pourquoi nous utilisons des interfaces ... 462
Bases de l'interface ... 464
"Cacher" les membres avec une implémentation explicite ... 466
IComparable comme exemple d'implémentation d'une interface ... 467
Chapitre 92: Interopérabilité ... 469

Remarques ... 469
Examples ... 469
Fonction d'importation à partir d'une DLL C ++ non gérée ... 469
Trouver la bibliothèque dynamique ... 469
Code simple pour exposer la classe pour com ... 470
C ++ nom mangling ... 470
Conventions d'appel ... 471
Chargement et déchargement dynamiques de DLL non gérées ... 472
Traitement des erreurs Win32 ... 473
Objet épinglé ... 474
Structures de lecture avec le maréchal ... 475
Chapitre 93: Interpolation de chaîne ... 477
Syntaxe ... 477
Remarques ... 477
Examples ... 477
Expressions ... 477
Format des dates en chaînes ... 477
Utilisation simple ... 478
Dans les coulisses ... 478
Rembourrage de la sortie ... 478
Rembourrage Gauche ... 478
Rembourrage Droit ... 479
Rembourrage avec spécificateurs de format ... 479
Mise en forme des nombres dans les chaînes ... 479
Chapitre 94: La mise en réseau ... 481
Syntaxe ... 481
Remarques ... 481
Examples ... 481
Client de communication TCP de base ... 481
Télécharger un fichier depuis un serveur Web ... 481
Async TCP Client ... 482

Client UDP de base ... 483
Chapitre 95: Lecture et écriture de fichiers .zip ... 485
Syntaxe ... 485
Paramètres ... 485
Examples ... 485
Ecrire dans un fichier zip ... 485
Écriture de fichiers Zip en mémoire ... 485
Obtenir des fichiers à partir d'un fichier Zip ... 486
L'exemple suivant montre comment ouvrir une archive zip et extraire tous les fichiers .txt ... 486
Chapitre 96: Les attributs ... 488
Examples ... 488
Création d'un attribut personnalisé ... 488
Utiliser un attribut ... 488
Lecture d'un attribut ... 488
Attribut DebuggerDisplay ... 489
Attributs d'information de l'appelant ... 490
Lecture d'un attribut depuis l'interface ... 491
Attribut obsolète ... 492
Chapitre 97: Les délégués ... 493
Remarques ... 493
Résumé ... 493
Types de délégué intégrés: Action<...> , Predicate<T> et Func<...,TResult> ... 493
Types de délégué personnalisés ... 493
Invoquer des délégués ... 493
Affectation aux délégués ... 493
Combinaison de délégués ... 493
Examples ... 494
Références sous-jacentes des délégués de méthodes nommées ... 494
Déclaration d'un type de délégué ... 494
Le func , Action et prédicat types de délégué ... 496
Affectation d'une méthode nommée à un délégué ... 497
Égalité des délégués ... 497

Assigner à un délégué par lambda ... 498
Passer des délégués en tant que paramètres ... 498
Combiner des délégués (délégués à la multidiffusion) ... 498
Appel de sécurité multicast délégué ... 500
Fermeture à l'intérieur d'un délégué ... 501
Encapsulation des transformations dans les funcs ... 502
Chapitre 98: Les itérateurs ... 503
Remarques ... 503
Examples ... 503
Exemple d'itérateur numérique simple ... 503
Création d'itérateurs à l'aide du rendement ... 503
Chapitre 99: Les méthodes ... 506
Examples ... 506
Déclaration d'une méthode ... 506
Appeler une méthode ... 506
Paramètres et arguments ... 507
Types de retour ... 507
Paramètres par défaut ... 508
Surcharge de méthode ... 509
Méthode anonyme ... 510
Des droits d'accès ... 511
Chapitre 100: Les opérateurs ... 512
Introduction ... 512
Syntaxe ... 512
Paramètres ... 512
Remarques ... 512
Priorité de l'opérateur ... 512
Examples ... 514
Opérateurs surchargeables ... 515
Opérateurs relationnels ... 516
Opérateurs de court-circuit ... 518
taille de ... 519

Surcharge des opérateurs d'égalité ... 520
Opérateurs membres de classe: Accès membres ... 521
Opérateurs membres de classe: accès conditionnel nul aux membres ... 521
Opérateurs membres de classe: invocation de fonction ... 521
Opérateurs membres de classe: indexation d'objets agrégés ... 521
Opérateurs membres de classe: indexation conditionnelle nulle ... 521
Opérateur "exclusif ou" ... 521
Opérateurs de transfert de bits ... 522
Opérateurs de diffusion implicite et de distribution explicite ... 522
Opérateurs binaires avec affectation ... 523
? : Opérateur ternaire ... 524
Type de ... 525
Opérateur par défaut ... 526
Type de valeur (où T: struct) ... 526
Type de référence (où T: classe) ... 526
nom de l'opérateur ... 526
? (Opérateur conditionnel nul) ... 526
Incrémentation et décrémentation du postfixe et du préfixe ... 527
=> Opérateur Lambda ... 528
Opérateur d'affectation '=' ... 529
?? Opérateur de coalescence nulle ... 529
Chapitre 101: Linq to Objects ... 530
Introduction ... 530
Examples ... 530
Comment LINQ to Object exécute les requêtes ... 530
Utiliser LINQ to Objects dans C # ... 530
Chapitre 102: LINQ to XML ... 535
Examples ... 535
Lire le XML en utilisant LINQ to XML ... 535
Chapitre 103: Lire et comprendre les empilements ... 537
Introduction ... 537
Examples ... 537

Trace de trace pour une exception NullReferenceException dans Windows Forms 537
Chapitre 104: Littéraux 539
Syntaxe 539
Examples 539
int littéraux 539
littéraux uint 539
littéraux de chaîne 539
littéraux char 540
littéraux octets 540
littéraux sbyte 540
littéraux décimaux 540
doubles littéraux 540
float littéraux 541
littéraux longs 541
ulong littéral 541
littéral court 541
Ushort littéral 541
littéraux bool 541
Chapitre 105: Manipulation de cordes 542
Examples 542
Changer la casse des caractères dans une chaîne 542
Trouver une chaîne dans une chaîne 542
Suppression (rognage) d'un espace blanc d'une chaîne 543
Remplacement d'une chaîne dans une chaîne 543
Fractionner une chaîne en utilisant un délimiteur 543
Concaténer un tableau de chaînes en une seule chaîne 544
Concaténation de chaînes 544
Chapitre 106: Méthodes d'extension 545
Syntaxe 545
Paramètres 545
Remarques 545
Examples 546

Méthodes d'extension - aperçu ... 546
Utiliser explicitement une méthode d'extension ... 549
Quand appeler les méthodes d'extension en tant que méthodes statiques ... 550
En utilisant statique ... 550
Vérification nulle ... 550
Les méthodes d'extension ne peuvent voir que les membres publics (ou internes) de la class ... 551
Méthodes d'extension génériques ... 551
Méthodes d'extension réparties selon le type statique ... 553
Les méthodes d'extension ne sont pas prises en charge par le code dynamique. ... 554
Méthodes d'extension en tant qu'encapsuleurs fortement typés ... 555
Méthodes d'extension pour le chaînage ... 555
Méthodes d'extension en combinaison avec des interfaces ... 556
IList Exemple de méthode d'extension: comparaison de 2 listes ... 557
Méthodes d'extension avec énumération ... 558
Les extensions et les interfaces permettent ensemble le code DRY et les fonctionnalités de ... 559
Méthodes d'extension pour la gestion de cas particuliers ... 560
Utilisation de méthodes d'extension avec des méthodes statiques et des rappels ... 560
Méthodes d'extension sur les interfaces ... 562
Utilisation de méthodes d'extension pour créer de belles classes de mappeurs ... 563
Utilisation de méthodes d'extension pour créer de nouveaux types de collection (par exempl ... 564
Chapitre 107: Méthodes DateTime ... 566
Examples ... 566
DateTime.Add (TimeSpan) ... 566
DateTime.AddDays (Double) ... 566
DateTime.AddHours (Double) ... 566
DateTime.AddMilliseconds (Double) ... 566
DateTime.Compare (DateTime t1, DateTime t2) ... 567
DateTime.DaysInMonth (Int32, Int32) ... 567
DateTime.AddYears (Int32) ... 567
Fonctions pures d'avertissement lors de l'utilisation de DateTime ... 568
DateTime.Parse (String) ... 568
DateTime.TryParse (String, DateTime) ... 568

Parse et TryParse avec informations culturelles ... 569
DateTime comme initialiseur dans for-loop ... 569
DateTime ToString, ToShortDateString, ToLongDateString et ToString formatés ... 569
Date actuelle ... 570
DateTime Formating ... 570
DateTime.ParseExact (String, String, IFormatProvider) ... 571
DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime) ... 572
Chapitre 108: Microsoft.Exchange.WebServices ... 575
Examples ... 575
Récupérer les paramètres d'absence de l'utilisateur spécifié ... 575
Mettre à jour les paramètres d'absence du bureau de l'utilisateur ... 576
Chapitre 109: Mise en cache ... 578
Examples ... 578
MemoryCache ... 578
Chapitre 110: Mise en œuvre d'un modèle de conception de décorateur ... 579
Remarques ... 579
Examples ... 579
Simuler une cafétéria ... 579
Chapitre 111: Mise en œuvre d'un modèle de conception de poids mouche ... 581
Examples ... 581
Mise en œuvre de la carte dans le jeu RPG ... 581
Chapitre 112: Mise en œuvre singleton ... 584
Examples ... 584
Singleton statiquement initialisé ... 584
Singleton paresseux et thread-safe (à l'aide du verrouillage à double contrôle) ... 584
Paresseux, Singleton (utilisant Lazy) ... 585
Paresseux, singleton thread-safe (pour .NET 3.5 ou version antérieure, implémentation alte ... 585
Élimination de l'instance Singleton lorsqu'elle n'est plus nécessaire ... 586
Chapitre 113: Modèles de conception créative ... 588
Remarques ... 588
Examples ... 588
Motif Singleton ... 588

Modèle de méthode d'usine ... 590
Motif de constructeur ... 593
Motif Prototype ... 597
Motif d'usine abstraite ... 598
Chapitre 114: Modèles de conception structurelle ... 602
Introduction ... 602
Examples ... 602
Modèle de conception d'adaptateur ... 602
Chapitre 115: Modificateurs d'accès ... 606
Remarques ... 606
Examples ... 606
Publique ... 606
privé ... 606
interne ... 607
protégé ... 607
protégé interne ... 608
Diagrammes de modificateurs d'accès ... 610
Chapitre 116: Mots clés ... 612
Introduction ... 612
Remarques ... 612
Examples ... 614
pilealloc ... 614
volatil ... 615
fixé ... 617
Variables fixes ... 617
Taille de tableau fixe ... 617
défaut ... 617
lecture seulement ... 618
comme ... 619
est ... 620
Type de ... 621
const ... 622

espace de noms 623
essayer, attraper, enfin lancer 624
continuer 625
ref, out 626
coché, non coché 627
aller à 628
goto tant que: 628
Étiquette: 628
Déclaration de cas: 629
Réessayer d'exception 629
enum 630
base 631
pour chaque 632
params 633
Pause 634
abstrait 636
float, double, décimal 637
flotte 637
double 638
décimal 638
uint 638
ce 639
pour 640
tandis que 641
revenir 643
dans 643
en utilisant 643
scellé 644
taille de 644
statique 644
Désavantages 647
int 647

longue 647
ulong 647
dynamique 648
virtuel, remplacement, nouveau 649
virtuel et remplacement 649
Nouveau 650
L'utilisation de la substitution n'est pas facultative 651
Les classes dérivées peuvent introduire un polymorphisme 652
Les méthodes virtuelles ne peuvent pas être privées 652
async, attend 653
carboniser 654
fermer à clé 654
nul 655
interne 656
où 657
Les exemples précédents montrent des contraintes génériques sur une définition de classe, 659
externe 660
bool 660
quand 661
décoché 661
Quand est-ce utile? 662
vide 662
si, si ... sinon, si ... sinon si 662
Important de noter que si une condition est remplie dans l'exemple ci-dessus, le contrôle 663
faire 664
opérateur 665
struct 666
commutateur 667
interface 668
peu sûr 668
implicite 671
vrai faux 671

chaîne ... 672
ushort ... 672
sbyte ... 672
var ... 672
déléguer ... 674
un événement ... 675
partiel ... 675
Chapitre 117: nom de l'opérateur ... 678
Introduction ... 678
Syntaxe ... 678
Examples ... 678
Utilisation de base: Impression d'un nom de variable ... 678
Impression d'un nom de paramètre ... 678
Événement Raising PropertyChanged ... 679
Gestion des événements PropertyChanged ... 679
Appliqué à un paramètre de type générique ... 680
Appliqué aux identificateurs qualifiés ... 681
Vérification des arguments et clauses de garde ... 681
Liens d'action MVC fortement typés ... 682
Chapitre 118: NullReferenceException ... 683
Examples ... 683
NullReferenceException expliqué ... 683
Chapitre 119: O (n) Algorithme de rotation circulaire d'un tableau ... 685
Introduction ... 685
Examples ... 685
Exemple de méthode générique qui fait pivoter un tableau par un décalage donné ... 685
Chapitre 120: ObservableCollection ... 687
Examples ... 687
Initialiser ObservableCollection ... 687
Chapitre 121: Opérateur d'égalité ... 688
Examples ... 688
Types d'égalité dans l'opérateur c # et d'égalité ... 688

Chapitre 122: Opérateur de coalescence nulle ... 689
Syntaxe ... 689
Paramètres ... 689
Remarques ... 689
Examples ... 689
Utilisation de base ... 689
Null fall-through et chaining ... 690
Opérateur de coalescence null avec appels de méthode ... 691
Utiliser existant ou créer de nouvelles ... 691
Initialisation des propriétés paresseuses avec un opérateur de coalescence nul ... 692
Fil de sécurité ... 692
C # 6 Sugar Syntactic utilisant des corps d'expression ... 692
Exemple dans le pattern MVVM ... 692
Chapitre 123: Opérateurs Null-Conditionnels ... 694
Syntaxe ... 694
Remarques ... 694
Examples ... 694
Opérateur Null-Conditionnel ... 694
Enchaînement de l'opérateur ... 695
Combinaison avec l'opérateur de coalescence nulle ... 695
L'indice Null-Conditionnel ... 695
Éviter les NullReferenceExceptions ... 695
L'opérateur Null-conditionnel peut être utilisé avec la méthode d'extension ... 696
Chapitre 124: Opérations sur les chaînes communes ... 697
Examples ... 697
Fractionnement d'une chaîne par caractère spécifique ... 697
Obtenir des sous-chaînes d'une chaîne donnée ... 697
Déterminer si une chaîne commence par une séquence donnée ... 697
Découpage des caractères indésirables au début et / ou à la fin des chaînes ... 697
String.Trim() ... 697
String.TrimStart() et String.TrimEnd() ... 698
Formater une chaîne ... 698

Joindre un tableau de chaînes dans une nouvelle 698
Remplissage d'une chaîne à une longueur fixe 698
Construire une chaîne à partir de tableau 698
Formatage avec ToString 699
Obtenir x caractères du côté droit d'une chaîne 700
Vérification de la chaîne vide à l'aide de String.IsNullOrEmpty () et String.IsNullOrWhite 701
Obtenir un caractère à un index spécifique et énumérer la chaîne 702
Convertir un nombre décimal en format binaire, octal et hexadécimal 702
Fractionnement d'une chaîne par une autre chaîne 703
Renverser correctement une chaîne 703
Remplacement d'une chaîne dans une chaîne 705
Changer la casse des caractères dans une chaîne 705
Concaténer un tableau de chaînes en une seule chaîne 706
Concaténation de chaînes 706
Chapitre 125: Parallèle LINQ (PLINQ) 707
Syntaxe 707
Examples 709
Exemple simple 709
WithDegreeOfParallelism 709
AsOrdered 709
Comme unordorded 710
Chapitre 126: Plate-forme de compilation .NET (Roslyn) 711
Examples 711
Créer un espace de travail à partir d'un projet MSBuild 711
Arbre de syntaxe 711
Modèle sémantique 712
Chapitre 127: Pointeurs 714
Remarques 714
Pointeurs et unsafe 714
Comportement non défini 714
Types prenant en charge les pointeurs 714
Examples 714

Pointeurs d'accès au tableau 714
Arithmétique de pointeur 715
L'astérisque fait partie du type 715
vide* 716
Accès membre utilisant -> 716
Pointeurs génériques 717
Chapitre 128: Pointeurs & Code dangereux 718
Examples 718
Introduction au code non sécurisé 718
Récupération de la valeur de données à l'aide d'un pointeur 719
Les pointeurs de passage comme paramètres des méthodes 719
Accès aux éléments du tableau à l'aide d'un pointeur 720
Compiler un code non sécurisé 721
Chapitre 129: Polymorphisme 723
Examples 723
Un autre exemple de polymorphisme 723
Types de polymorphisme 724
Polymorphisme ad hoc 724
Sous-typage 725
Chapitre 130: Pour commencer: Json avec C # 727
Introduction 727
Examples 727
Exemple Json Simple 727
Les premières choses: la bibliothèque pour travailler avec Json 727
Implémentation C # 727
La sérialisation 728
Désérialisation 729
Fonction Serialization & De-Serialization Common Utilities 729
Chapitre 131: Prise asynchrone 730
Introduction 730
Remarques 730
Examples 731

Exemple de socket asynchrone (client / serveur) ... 731
Chapitre 132: Programmation fonctionnelle ... 739
Examples ... 739
Func et Action ... 739
Immutabilité ... 739
Éviter les références nulles ... 741
Fonctions d'ordre supérieur ... 742
Collections immuables ... 742
Créer et ajouter des éléments ... 742
Création à l'aide du constructeur ... 743
Créer à partir d'un IEnumerable existant ... 743
Chapitre 133: Programmation orientée objet en C # ... 744
Introduction ... 744
Examples ... 744
Des classes: ... 744
Chapitre 134: Propriétés ... 745
Remarques ... 745
Examples ... 745
Diverses propriétés en contexte ... 745
Public Get ... 746
Ensemble public ... 746
Accès aux propriétés ... 746
Valeurs par défaut pour les propriétés ... 748
Propriétés implémentées automatiquement ... 748
Propriétés en lecture seule ... 749
Déclaration ... 749
Utilisation de propriétés en lecture seule pour créer des classes immuables ... 750
Chapitre 135: Récursivité ... 751
Remarques ... 751
Examples ... 751
Décrire récursivement une structure d'objet ... 751

Récursivité en anglais 752
Utilisation de la récursivité pour obtenir l'arborescence de répertoires 753
Séquence de Fibonacci 755
Calcul factoriel 756
Calcul PowerOf 756
Chapitre 136: Réflexion 758
Introduction 758
Remarques 758
Examples 758
Obtenir un System.Type 758
Obtenir les membres d'un type 758
Obtenez une méthode et invoquez-la 759
Obtenir et définir les propriétés 760
Attributs personnalisés 760
Faire le tour de toutes les propriétés d'une classe 762
Détermination des arguments génériques des instances de types génériques 762
Obtenez une méthode générique et invoquez-la 763
Créer une instance d'un type générique et appeler sa méthode 764
Classes d'instanciation qui implémentent une interface (par exemple, activation de plug-in 764
Création d'une instance d'un type 765
Avec la classe Activator 765
Sans classe d' Activator 765
Obtenir un type par nom avec un espace de noms 768
Obtenir un délégué fortement typé sur une méthode ou une propriété via la réflexion 769
Chapitre 137: Rendement 771
Introduction 771
Syntaxe 771
Remarques 771
Examples 771
Utilisation simple 771
Utilisation plus pertinente 772
Résiliation anticipée 772

Vérification correcte des arguments ... 773
Renvoie un autre Enumerable dans une méthode renvoyant Enumerable ... 775
Évaluation paresseuse ... 775
Essayez ... enfin ... 776
Utiliser les rendements pour créer un IEnumerator lors de l'implémentation de IEnumerable ... 777
Évaluation avide ... 778
Exemple d'évaluation paresseuse: numéros de Fibonacci ... 778
La différence entre break et break ... 779
Chapitre 138: Rendre un thread variable sûr ... 782
Examples ... 782
Contrôler l'accès à une variable dans une boucle Parallel.For ... 782
Chapitre 139: Requêtes LINQ ... 783
Introduction ... 783
Syntaxe ... 783
Remarques ... 785
Examples ... 785
Où ... 785
Syntaxe de la méthode ... 785
Syntaxe de requête ... 786
Sélectionner - Éléments de transformation ... 786
Méthodes de chaînage ... 786
Portée et répétition ... 787
Gamme ... 788
Répéter ... 788
Passer et prendre ... 788
Tout d'abord, FirstOrDefault, Last, LastOrDefault, Single et SingleOrDefault ... 789
Premier() ... 789
FirstOrDefault () ... 789
Dernier() ... 790
LastOrDefault () ... 791
Unique() ... 791

SingleOrDefault () 792
Recommandations 792
Sauf 793
SelectMany: Aplatir une séquence de séquences 795
SelectMany 796
Tout 797
1. paramètre vide 798
2. Expression Lambda en tant que paramètre 798
3. Collection vide 798
Collection de requêtes par type / éléments de distribution à taper 798
syndicat 799
JOINT 799
(Jointure interne 799
Jointure externe gauche 800
Right Outer Join 800
Cross Join 800
Full Outer Join 800
Exemple pratique 801
Distinct 802
GroupBy un ou plusieurs champs 802
Utiliser Range avec différentes méthodes Linq 803
Ordre des requêtes - OrderBy () ThenBy () OrderByDescending () ThenByDescending () 803
Les bases 804
Par groupe 805
Exemple simple 805
Exemple plus complexe 806
Tout 807
1. paramètre vide 807
2. Expression Lambda en tant que paramètre 807
3. Collection vide 807
ToDictionary 807
Agrégat 808

Définir une variable dans une requête Linq (mot clé let) 809
SkipWhile 810
DefaultIfEmpty 810
Utilisation dans les jointures à gauche : 810
SéquenceEqual 811
Count et LongCount 812
Créer progressivement une requête 812
Zip *: français 814
GroupJoin avec variable de plage externe 814
ElementAt et ElementAtOrDefault 814
Linq Quantifiers 815
Joindre plusieurs séquences 816
Se joindre à plusieurs clés 818
Sélectionnez avec Func sélecteur - Permet d'obtenir le classement des éléments 818
TakeWhile 819
Somme 820
Pour rechercher 820
Construisez vos propres opérateurs Linq pour IEnumerable 821
Utilisation de SelectMany au lieu de boucles imbriquées 822
Any and First (OrDefault) - meilleures pratiques 822
GroupBy Sum et Count 823
Sens inverse 824
Énumérer les Enumerables 825
Commandé par 827
OrderByDescending 828
Concat 828
Contient 829
Chapitre 140: Résolution de surcharge 831
Remarques 831
Examples 831
Exemple de surcharge de base 831
"params" n'est pas développé, sauf si nécessaire. 832
Passer null comme l'un des arguments 832

Chapitre 141: Runtime Compile 834
Examples 834
RoslynScript 834
CSharpCodeProvider 834
Chapitre 142: Séquences d'échappement de chaîne 835
Syntaxe 835
Remarques 835
Examples 835
Séquences d'échappement de caractères Unicode 835
Échapper à des symboles spéciaux dans les littéraux de caractère 836
Échapper aux symboles spéciaux dans les littéraux de chaîne 836
Les séquences d'échappement non reconnues génèrent des erreurs de compilation 836
Utilisation de séquences d'échappement dans les identificateurs 837
Chapitre 143: Sérialisation Binaire 838
Remarques 838
Examples 838
Rendre un objet sérialisable 838
Contrôle du comportement de sérialisation avec des attributs 838
Ajouter plus de contrôle en implémentant ISerializable 839
Les substituts de sérialisation (implémentation d'ISerializationSurrogate) 840
Classeur de sérialisation 843
Quelques pièges dans la compatibilité ascendante 844
Chapitre 144: String.Format 848
Introduction 848
Syntaxe 848
Paramètres 848
Remarques 848
Examples 848
Endroits où String.Format est "incorporé" dans la structure 848
Utilisation du format numérique personnalisé 849
Créer un fournisseur de format personnalisé 849
Aligner à gauche / à droite, pad avec des espaces 850

Formats numériques ... 850
Mise en forme de devise ... 850
Précision ... 851
Symbole de la monnaie ... 851
Position du symbole monétaire ... 851
Séparateur décimal personnalisé ... 851
Depuis C # 6.0 ... 852
Echappement des accolades dans une expression String.Format () ... 852
Formatage de date ... 852
ToString () ... 854
Relation avec ToString () ... 855
Restrictions relatives aux mises en garde et au formatage ... 855
Chapitre 145: StringBuilder ... 856
Examples ... 856
Qu'est-ce qu'un StringBuilder et quand l'utiliser ... 856
Utilisez StringBuilder pour créer une chaîne à partir d'un grand nombre d'enregistrements ... 857
Chapitre 146: Structs ... 859
Remarques ... 859
Examples ... 859
Déclarer une structure ... 859
Utilisation de la structure ... 860
Interface d'implémentation Struct ... 861
Les structures sont copiées lors de l'affectation ... 861
Chapitre 147: System.DirectoryServices.Protocols.LdapConnection ... 863
Examples ... 863
Connexion SSL LDAP authentifiée, le certificat SSL ne correspond pas au DNS inversé ... 863
LDAP anonyme super simple ... 864
Chapitre 148: System.Management.Automation ... 865
Remarques ... 865
Examples ... 865
Invoquer un pipeline synchrone simple ... 865
Chapitre 149: Tableaux ... 867

Syntaxe 867
Remarques 867
Examples 867
Covariance de tableau 868
Obtenir et définir des valeurs de tableau 868
Déclarer un tableau 868
Itérer sur un tableau 869
Tableaux multidimensionnels 870
Tableaux dentelés 870
Vérifier si un tableau contient un autre tableau 871
Initialisation d'un tableau rempli avec une valeur non par défaut répétée 872
Copier des tableaux 873
Créer un tableau de numéros séquentiels 873
Usage: 874
Comparer les tableaux pour l'égalité 874
Tableaux en tant qu'instances IEnumerable <> 874
Chapitre 150: Tuples 876
Examples 876
Créer des tuples 876
Accès aux éléments de tuple 876
Comparer et trier les tuples 876
Renvoie plusieurs valeurs d'une méthode 877
Chapitre 151: Type de valeur vs type de référence 878
Syntaxe 878
Remarques 878
introduction 878
Types de valeur 878
Types de référence 878
Différences majeures 878
Des types de valeur existent sur la pile, des types de référence existent sur le tas 879
Les types de valeur ne changent pas lorsque vous les modifiez dans une méthode, les types 879
Les types de valeur ne peuvent pas être null, les types de référence peuvent 879

Examples ... 879
Changer les valeurs ailleurs ... 880
En passant par référence ... 881
Passage par référence en utilisant le mot-clé ref ... 881
Affectation ... 882
Différence avec les paramètres de méthode réf et out ... 882
paramètres de réf vs out ... 884
Chapitre 152: Type dynamique ... 886
Remarques ... 886
Examples ... 886
Créer une variable dynamique ... 886
Retour dynamique ... 886
Créer un objet dynamique avec des propriétés ... 887
Gestion de types spécifiques inconnus à la compilation ... 887
Chapitre 153: Types anonymes ... 889
Examples ... 889
Créer un type anonyme ... 889
Anonyme vs dynamique ... 889
Méthodes génériques avec types anonymes ... 890
Instancier des types génériques avec des types anonymes ... 890
Égalité de type anonyme ... 890
Tableaux tapés implicitement ... 891
Chapitre 154: Types intégrés ... 892
Examples ... 892
Type de référence immuable - chaîne ... 892
Type de valeur - char ... 892
Type de valeur - short, int, long (entiers signés 16 bits, 32 bits, 64 bits) ... 892
Type de valeur - ushort, uint, ulong (entiers 16 bits non signés, 32 bits, 64 bits) ... 893
Type de valeur - bool ... 893
Comparaisons avec les types de valeur encadrés ... 894
Conversion de types de valeur encadrés ... 894
Chapitre 155: Types nullables ... 895

Syntaxe ... 895
Remarques ... 895
Examples ... 895
Initialiser un nullable ... 896
Vérifier si un Nullable a une valeur ... 896
Récupère la valeur d'un type nullable ... 896
Obtenir une valeur par défaut à partir d'un nullable ... 897
Vérifier si un paramètre de type générique est un type nullable ... 897
La valeur par défaut des types nullables est null ... 897
Utilisation efficace des Nullables sous-jacents argument ... 898
Chapitre 156: Un aperçu des collections c # ... 900
Examples ... 900
HashSet ... 900
SortedSet ... 900
T [] (Tableau de T) ... 900
liste ... 901
dictionnaire ... 901
Clé en double lors de l'initialisation de la collection ... 902
Empiler ... 902
LinkedList ... 903
Queue ... 903
Chapitre 157: Utiliser json.net ... 904
Introduction ... 904
Examples ... 904
Utiliser JsonConverter sur des valeurs simples ... 904
JSON (http://www.omdbapi.com/?i=tt1663662) ... 904
Modèle de film ... 905
RuntimeSerializer ... 905
L'appel ... 906
Collecte tous les champs de l'objet JSON ... 906
Chapitre 158: Utiliser la déclaration ... 909

Introduction 909
Syntaxe 909
Remarques 909
Examples 909
Utilisation des principes de base de l'instruction 909
Retourner de l'utilisation du bloc 910
Instructions d'utilisation multiples avec un bloc 911
Gotcha: retourner la ressource que vous disposez 912
L'utilisation d'instructions est null-safe 912
Gotcha: Exception dans la méthode Dispose masquant d'autres erreurs dans Utilisation des b 913
Utilisation des instructions et des connexions à la base de données 913
Classes de données communes IDisposable 914
Modèle d'accès commun pour les connexions ADO.NET 914
Utilisation des instructions avec DataContexts 915
Utilisation de la syntaxe Dispose pour définir une étendue personnalisée 915
Exécuter du code dans un contexte de contrainte 916
Chapitre 159: Utiliser la directive 918
Remarques 918
Examples 918
Utilisation de base 918
Référence à un espace de noms 918
Associer un alias à un espace de noms 918
Accéder aux membres statiques d'une classe 919
Associer un alias pour résoudre des conflits 919
Utiliser des directives d'alias 920
Chapitre 160: Utiliser SQLite en C # 921
Examples 921
Créer un CRUD simple en utilisant SQLite en C # 921
Exécution de la requête 925
Chapitre 161: Vérifié et décoché 927
Syntaxe 927
Examples 927

Vérifié et décoché ... 927
Vérifié et décoché comme une portée ... 927
Chapitre 162: Windows Communication Foundation ... 928
Introduction ... 928
Examples ... 928
Exemple de démarrage ... 928
Chapitre 163: XDocument et l'espace de noms System.Xml.Linq ... 931
Examples ... 931
Générer un document XML ... 931
Modifier un fichier XML ... 931
Générer un document XML en utilisant une syntaxe fluide ... 933
Chapitre 164: XmlDocument et l'espace de noms System.Xml ... 934
Examples ... 934
Interaction de base du document XML ... 934
Lecture du document XML ... 934
XmlDocument vs XDocument (exemple et comparaison) ... 935
Chapitre 165: Y compris les ressources de police ... 938
Paramètres ... 938
Examples ... 938
Instancier «Fontfamily» à partir des ressources ... 938
Méthode d'intégration ... 938
Utilisation avec un bouton ... 939
Crédits ... 940

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: csharp-language

It is an unofficial and free C# Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C# Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage C

Remarques

C # est un langage de programmation multi-paradigme, C-descendant de Microsoft. C # est un langage géré qui compile en CIL , intermédiaire qui peut être exécuté sous Windows, Mac OS X et Linux.

Les versions 1.0, 2.0 et 5.0 ont été normalisées par ECMA (comme ECMA-334), et les efforts de normalisation pour le C # moderne sont en cours.

Versions

Version	Date de sortie
1.0	2002-01-01
1.2	2003-04-01
2.0	2005-09-01
3.0	2007-08-01
4.0	2010-04-01
5.0	2013-06-01
6,0	2015-07-01
7.0	2017-03-07

Examples

Création d'une nouvelle application console (Visual Studio)

1. Ouvrez Visual Studio
2. Dans la barre d'outils, accédez à **Fichier** → **Nouveau projet**
3. Sélectionnez le type de projet d' **application console**
4. Ouvrez le fichier `Program.cs` dans l'explorateur de solutions
5. Ajoutez le code suivant à `Main()` :

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
```

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

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
           and disappearing before the programmer can see the contents
           of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. Dans la barre d'outils, cliquez sur **Déboguer** -> **Démarrer le débogage** ou appuyez sur **F5** ou **Ctrl + F5** (en cours d'exécution sans débogueur) pour exécuter le programme.

Démo en direct sur ideone

Explication

- `class Program` est une déclaration de classe. Le `Program` classe contient les définitions de données et de méthodes utilisées par votre programme. Les classes contiennent généralement plusieurs méthodes. Les méthodes définissent le comportement de la classe. Cependant, la classe de `Program` n'a qu'une seule méthode: `Main` .

- `static void Main()` définit la méthode `Main` , qui est le point d'entrée pour tous les programmes C #. La méthode `Main` indique ce que fait la classe lorsqu'elle est exécutée. Une seule méthode `Main` est autorisée par classe.

- `System.Console.WriteLine("Hello, world!");` méthode imprime une donnée donnée (dans cet exemple, `Hello, world!`) en tant que sortie dans la fenêtre de la console.

- `System.Console.ReadKey()` garantit que le programme ne se fermera pas immédiatement après l'affichage du message. Cela se fait en attendant que l'utilisateur appuie sur une touche du clavier. Toute pression sur la touche de l'utilisateur mettra fin au programme. Le programme se termine lorsqu'il a terminé la dernière ligne de code de la méthode `main()` .

Utiliser la ligne de commande

Pour compiler via la ligne de commande, utilisez `MSBuild` ou `csc.exe` *(le compilateur C #)* , tous deux faisant partie du package Microsoft Build Tools .

Pour compiler cet exemple, exécutez la commande suivante dans le même répertoire que `HelloWorld.cs` :

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

Il est également possible que vous ayez deux méthodes principales dans une même application. Dans ce cas, vous devez dire au compilateur principale méthode pour exécuter en tapant la

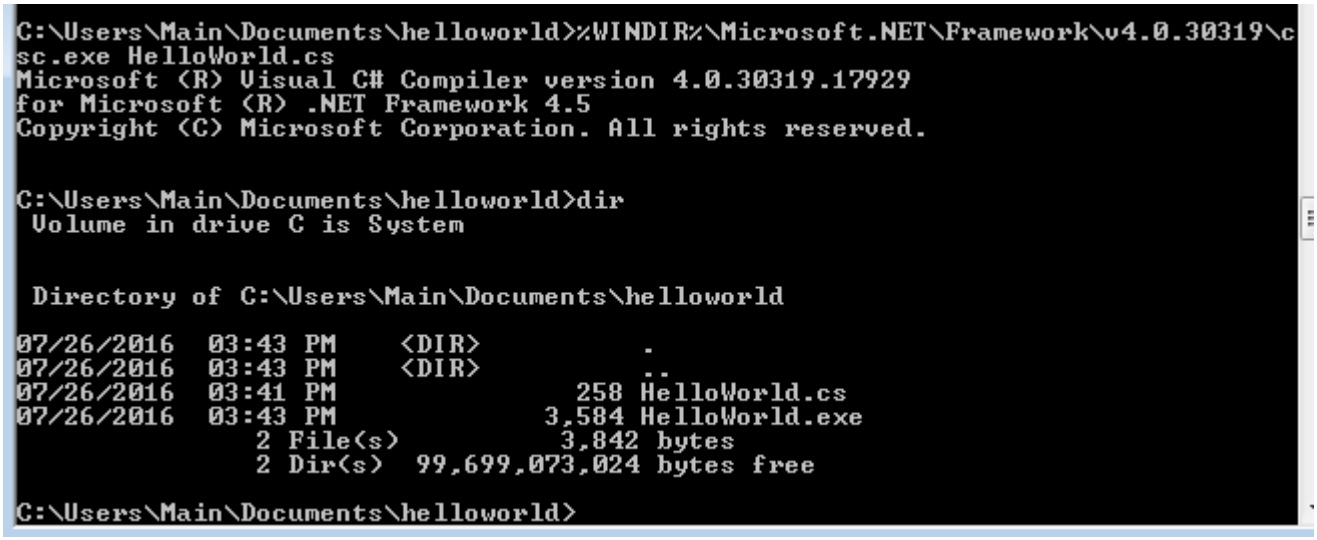
commande suivante dans la **console.** (Classe supposons `ClassA` a aussi une méthode principale dans le même `HelloWorld.cs` fichier dans l' espace de noms HelloWorld)

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

où HelloWorld est un espace de noms

***Note** : Il s'agit du chemin où **.NET Framework v4.0** se trouve en général. Changez le chemin en fonction de votre version .NET. En outre, le répertoire peut être un **framework** au lieu de **framework64** si vous utilisez le .NET Framework 32 bits. À partir de l'invite de commandes Windows, vous pouvez répertorier tous les chemins de structure csc.exe en exécutant les commandes suivantes (les premières pour les cadres 32 bits):*

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```



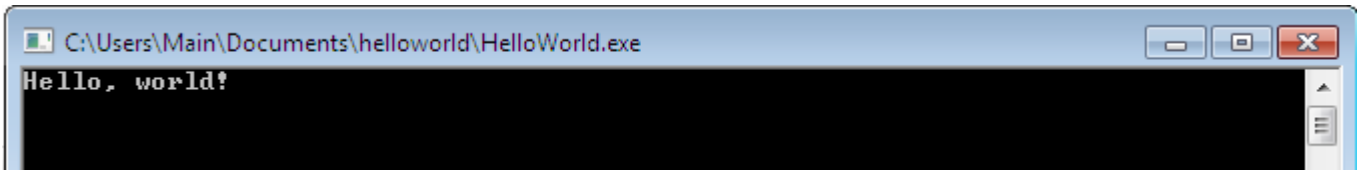
Il devrait maintenant y avoir un fichier exécutable nommé `HelloWorld.exe` dans le même répertoire. Pour exécuter le programme à partir de l'invite de commande, tapez simplement le nom de l'exécutable et appuyez sur `Entrée` comme suit:

```
HelloWorld.exe
```

Cela produira:

> Bonjour le monde!

Vous pouvez également double-cliquer sur l'exécutable et lancer une nouvelle fenêtre de console avec le message " **Hello, world!** "

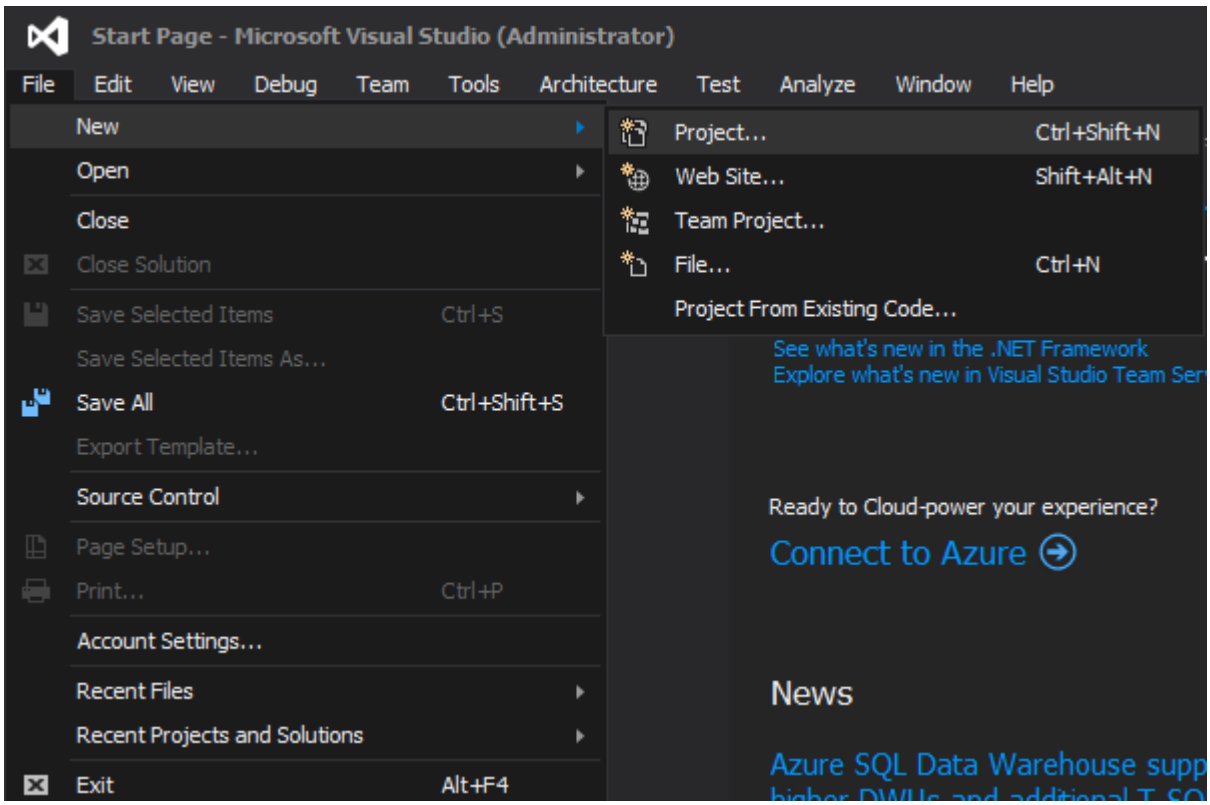


Création d'un nouveau projet dans Visual Studio (application console) et exécution en mode débogage

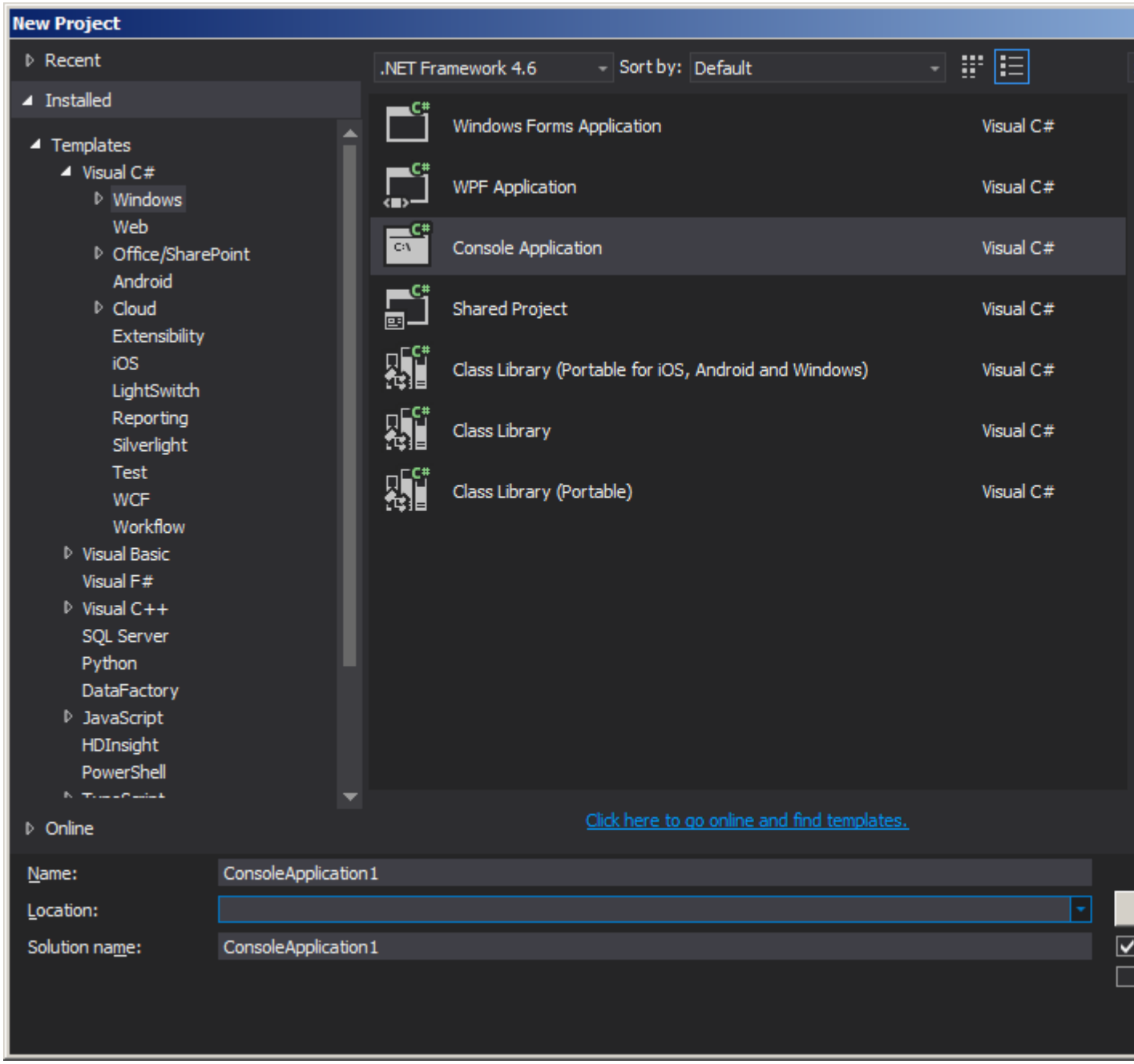
1. **Téléchargez et installez Visual Studio** . Visual Studio peut être téléchargé à partir de VisualStudio.com . L'édition communautaire est suggérée d'abord parce qu'elle est gratuite et d'autre part parce qu'elle concerne toutes les caractéristiques générales et peut être étendue.

2. **Ouvrez Visual Studio.**

3. **Bienvenue.** Allez dans **Fichier → Nouveau → Projet** .

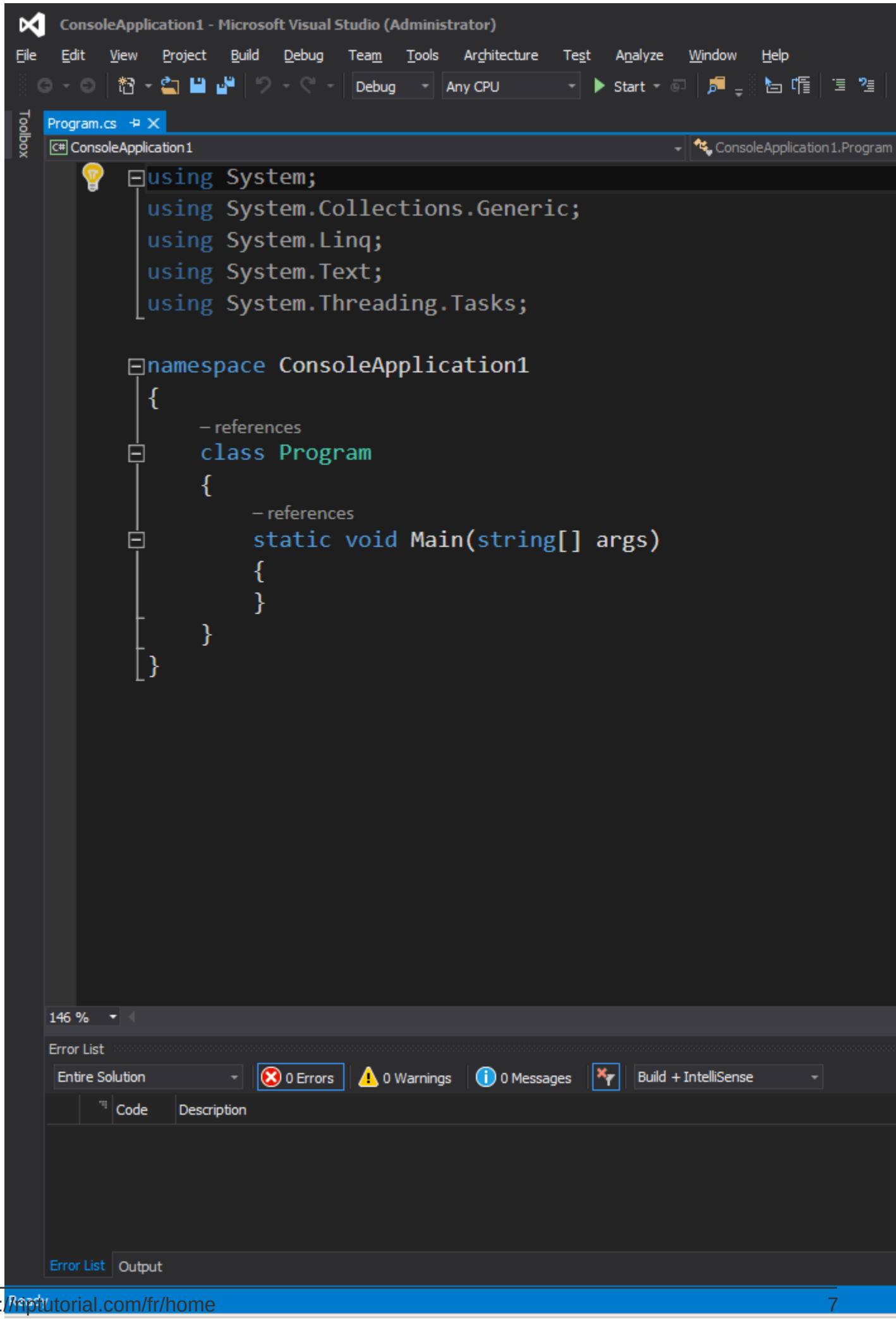


4. Cliquez sur **Modèles** → **Visual C #** → **Application console**



5. **Après avoir sélectionné Application console,** entrez un nom pour votre projet et un emplacement à enregistrer, puis appuyez sur `OK` . Ne vous souciez pas du nom de la solution.

6. **Projet créé** . Le projet nouvellement créé ressemblera à:


ConsoleApplication1 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
Toolbox
Program.cs
ConsoleApplication1
ConsoleApplication1.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
0 references
class Program
{
0 references
static void Main(string[] args)
{
}
}
}
146 %
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Code
Description
Error List
Output
Ready

(Utilisez toujours des noms descriptifs pour les projets afin de pouvoir les distinguer facilement des autres projets. Il est recommandé de ne pas utiliser d'espaces dans le nom du projet ou de la classe.)

7. **Ecrire le code.** Vous pouvez maintenant mettre à jour votre `Program.cs` pour présenter "Hello world!" à l'utilisateur.

   pour présenter "Hello world!" à l'utilisateur.

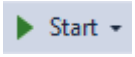
```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

   Ajoutez les deux lignes suivantes à l'objet `public static void Main(string[] args)` dans `Program.cs` : (assurez-vous qu'il se trouve entre les accolades)

```
Console.WriteLine("Hello world!");
Console.Read();
```

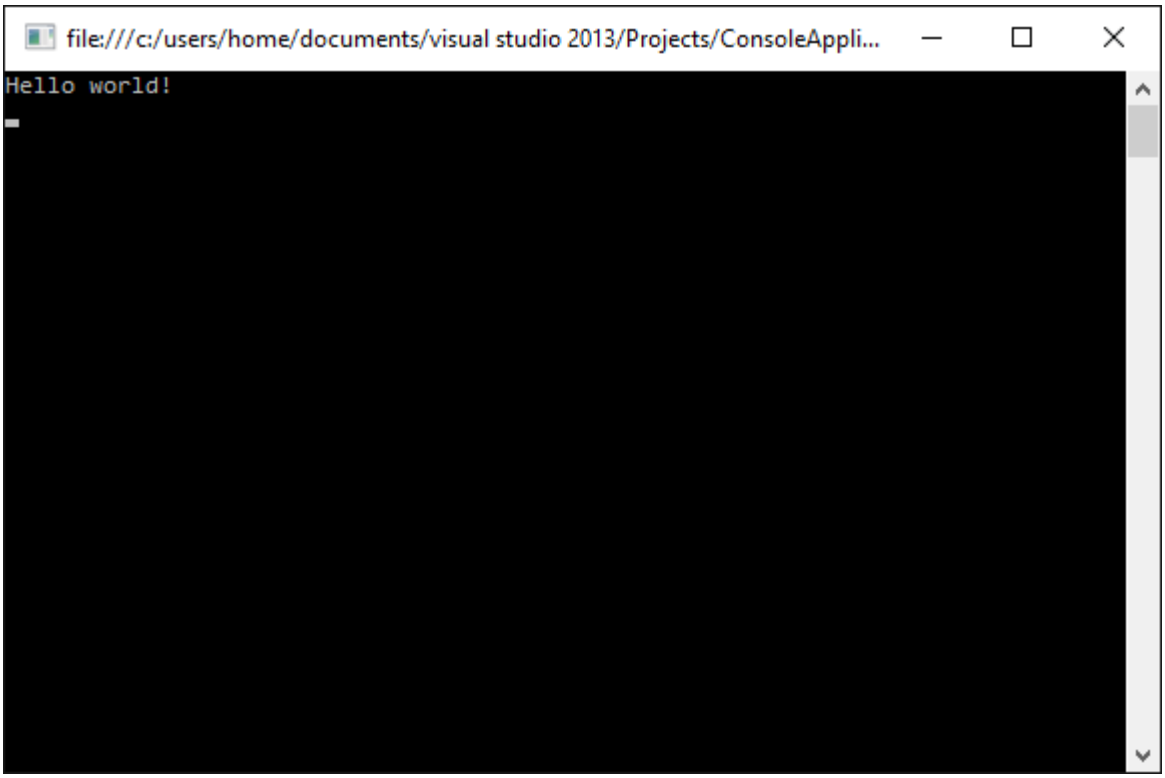
   Pourquoi `Console.Read()` **?** La première ligne affiche le texte "Hello world!" à la console, et la deuxième ligne attend qu'un seul caractère soit entré; En effet, le programme interrompt l'exécution du programme afin que vous puissiez voir la sortie lors du débogage. Sans `Console.Read();` , lorsque vous commencez à déboguer l'application, elle affichera simplement "Hello world!" à la console puis fermez immédiatement. Votre fenêtre de code devrait maintenant ressembler à ceci:

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. **Déboguer votre programme.** Appuyez sur le bouton Démarrer dans la barre d'outils située en haut de la fenêtre. 

ou appuyez sur `F5` sur votre clavier pour exécuter votre application. Si le bouton n'est pas présent, vous pouvez exécuter le programme depuis le menu principal: **Debug → Start Debugging** . Le programme compilera puis ouvrira une

fenêtre de console. Il devrait ressembler à la capture d'écran suivante:



9. **Arrêtez le programme.** Pour fermer le programme, appuyez simplement sur une touche de votre clavier. La `Console.Read()` nous avons ajoutée était dans le même but. Une autre façon de fermer le programme consiste à accéder au menu où se trouvait le bouton `Démarrer` et à cliquer sur le bouton `Arrêter` .

Créer un nouveau programme en utilisant Mono

Installez d'abord Mono en suivant les instructions d'installation de la plate-forme de votre choix, comme décrit dans la section Installation .

Mono est disponible pour Mac OS X, Windows et Linux.

Une fois l'installation terminée, créez un fichier texte, nommez-le `HelloWorld.cs` et copiez-y le contenu suivant:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

Si vous utilisez Windows, exécutez l'invite de commande Mono incluse dans l'installation Mono et assurez-vous que les variables d'environnement nécessaires sont définies. Si sous Mac ou Linux, ouvrez un nouveau terminal.

Pour compiler le fichier nouvellement créé, exécutez la commande suivante dans le répertoire contenant `HelloWorld.cs` :

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

Le `HelloWorld.exe` résultant peut alors être exécuté avec:

```
mono HelloWorld.exe
```

qui produira la sortie:

```
Hello, world!
Press any key to exit..
```

Créer un nouveau programme en utilisant .NET Core

Installez d'abord le **.NET Core SDK** en parcourant les instructions d'installation de la plate-forme de votre choix:

- les fenêtres
- OSX
- Linux
- Docker

Une fois l'installation terminée, ouvrez une invite de commande ou une fenêtre de terminal.

1. Créez un nouveau répertoire avec `mkdir hello_world` et changez dans le répertoire nouvellement créé avec `cd hello_world` .

2. Créez une nouvelle application console avec la `dotnet new console` .
   Cela produira deux fichiers:

   - **hello_world.csproj**

     ```
     <Project Sdk="Microsoft.NET.Sdk">

       <PropertyGroup>
         <OutputType>Exe</OutputType>
         <TargetFramework>netcoreapp1.1</TargetFramework>
       </PropertyGroup>

     </Project>
     ```

   - **Program.cs**

     ```
     using System;

     namespace hello_world
     {
         class Program
     ```

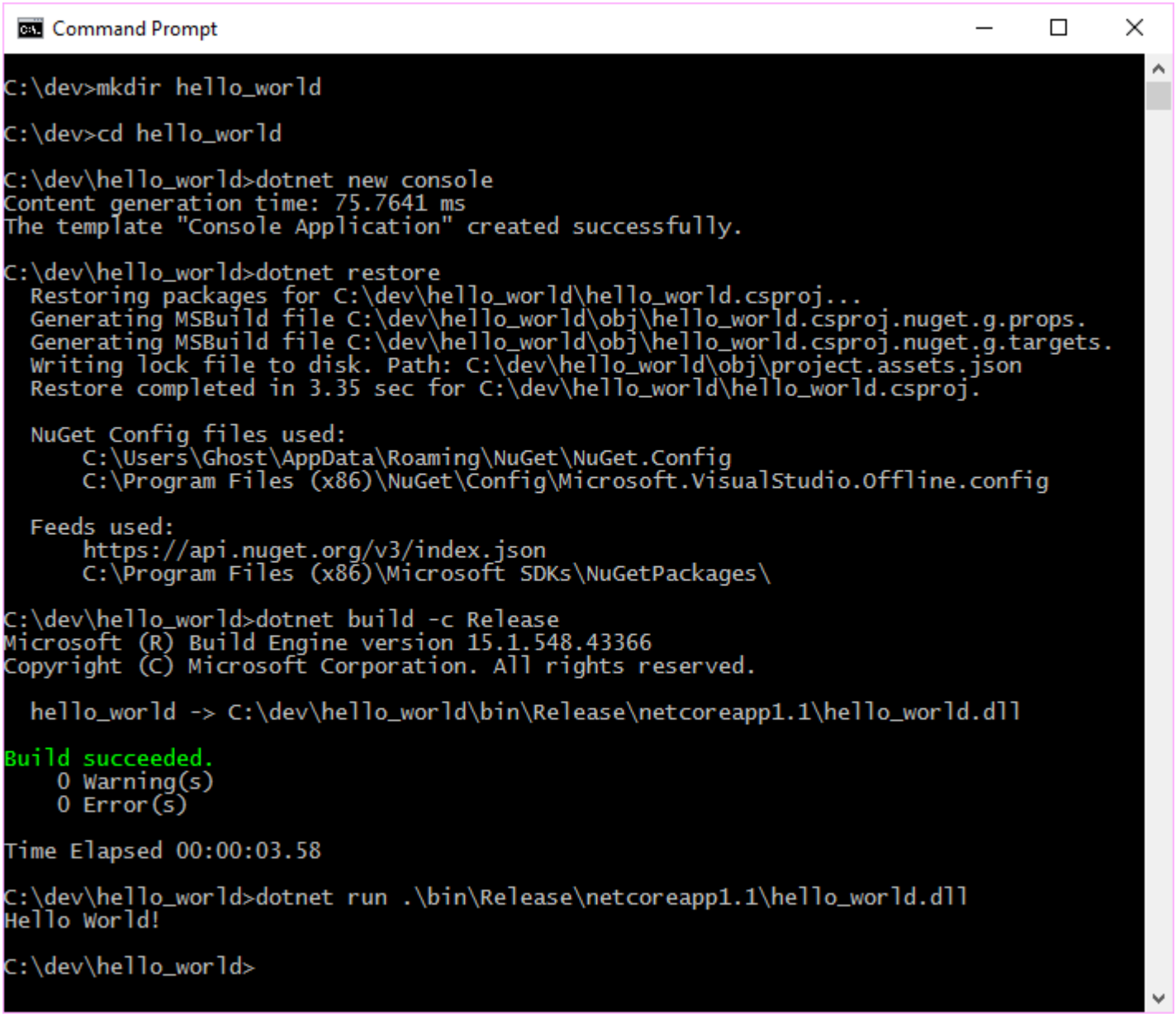
```
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

3. Restaurez les paquets nécessaires avec `dotnet restore` .

4. *Facultatif* Générez l'application avec `dotnet build` pour Debug ou `dotnet build -c Release` pour Release. `dotnet run` également le compilateur et lancera des erreurs de construction, le cas échéant.

5. Exécutez l'application avec `dotnet run` pour Debug ou `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` pour la version.

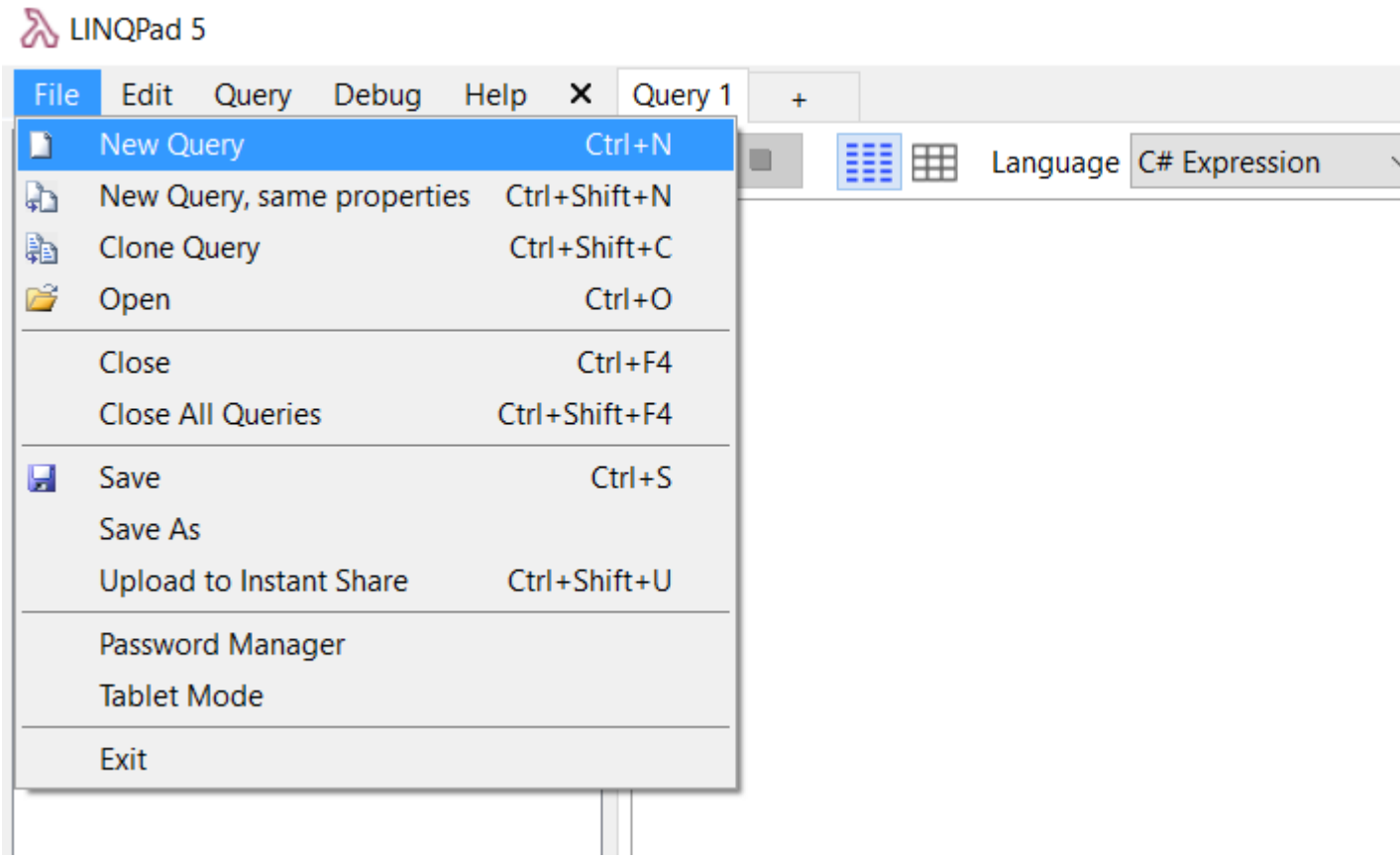
Sortie d'invite de commande



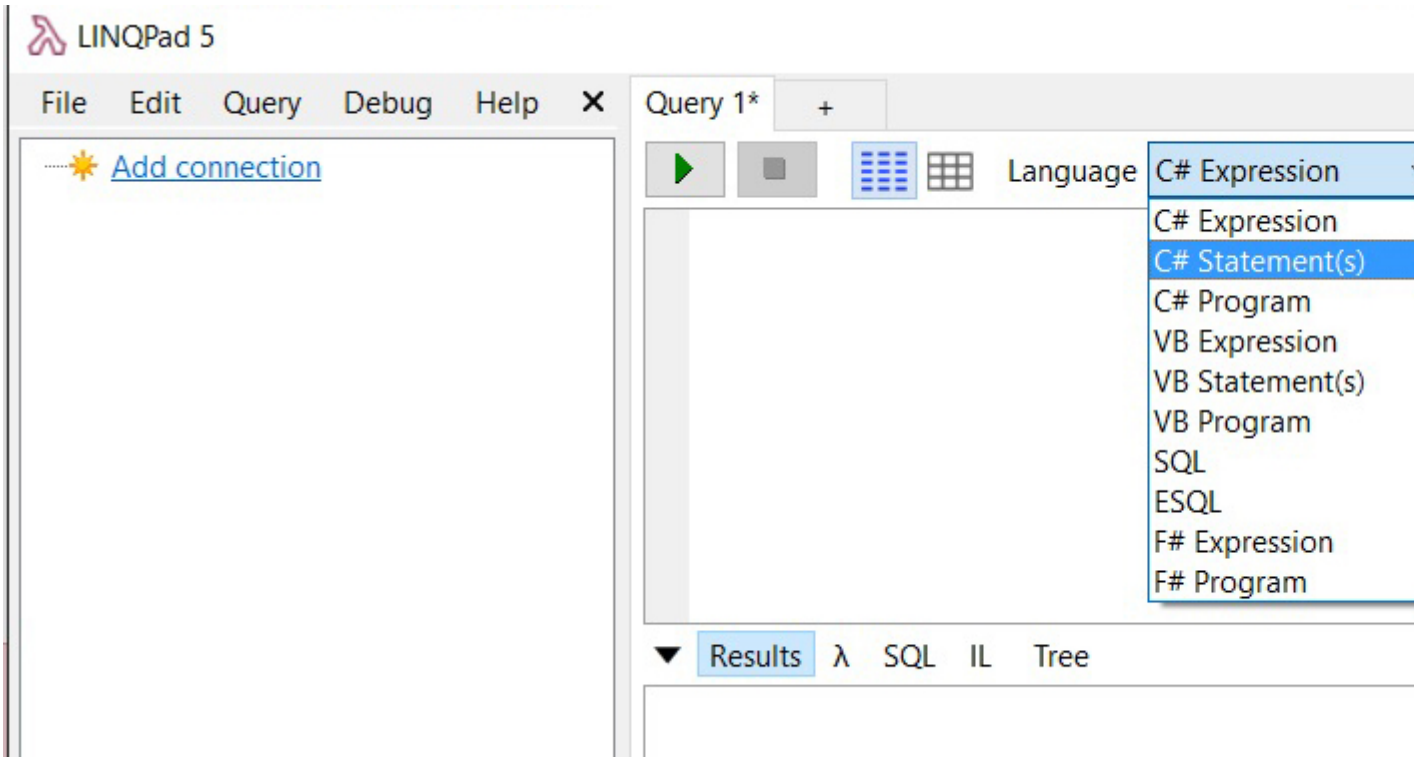
Création d'une nouvelle requête à l'aide de LinqPad

LinqPad est un excellent outil qui vous permet d'apprendre et de tester les fonctionnalités des langages .Net (C #, F # et VB.Net.)

1. Installez LinqPad
2. Créer une nouvelle requête (`Ctrl` + `N`)

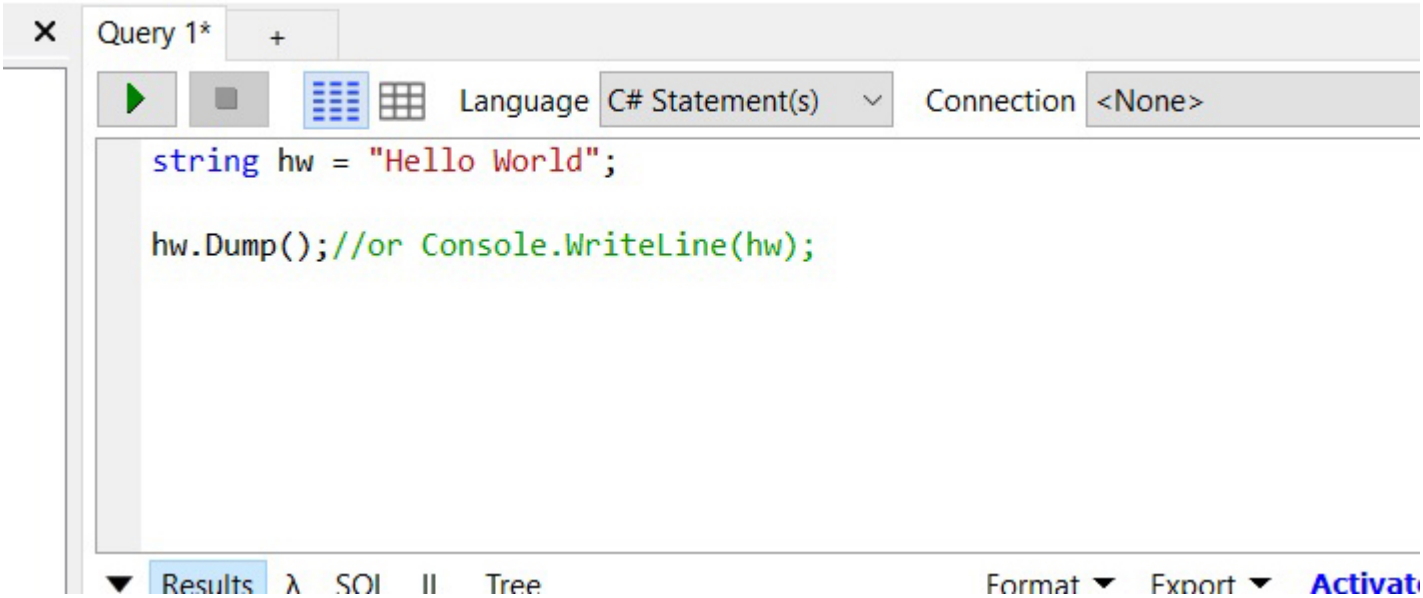


3. Sous la langue, sélectionnez "instructions C #"

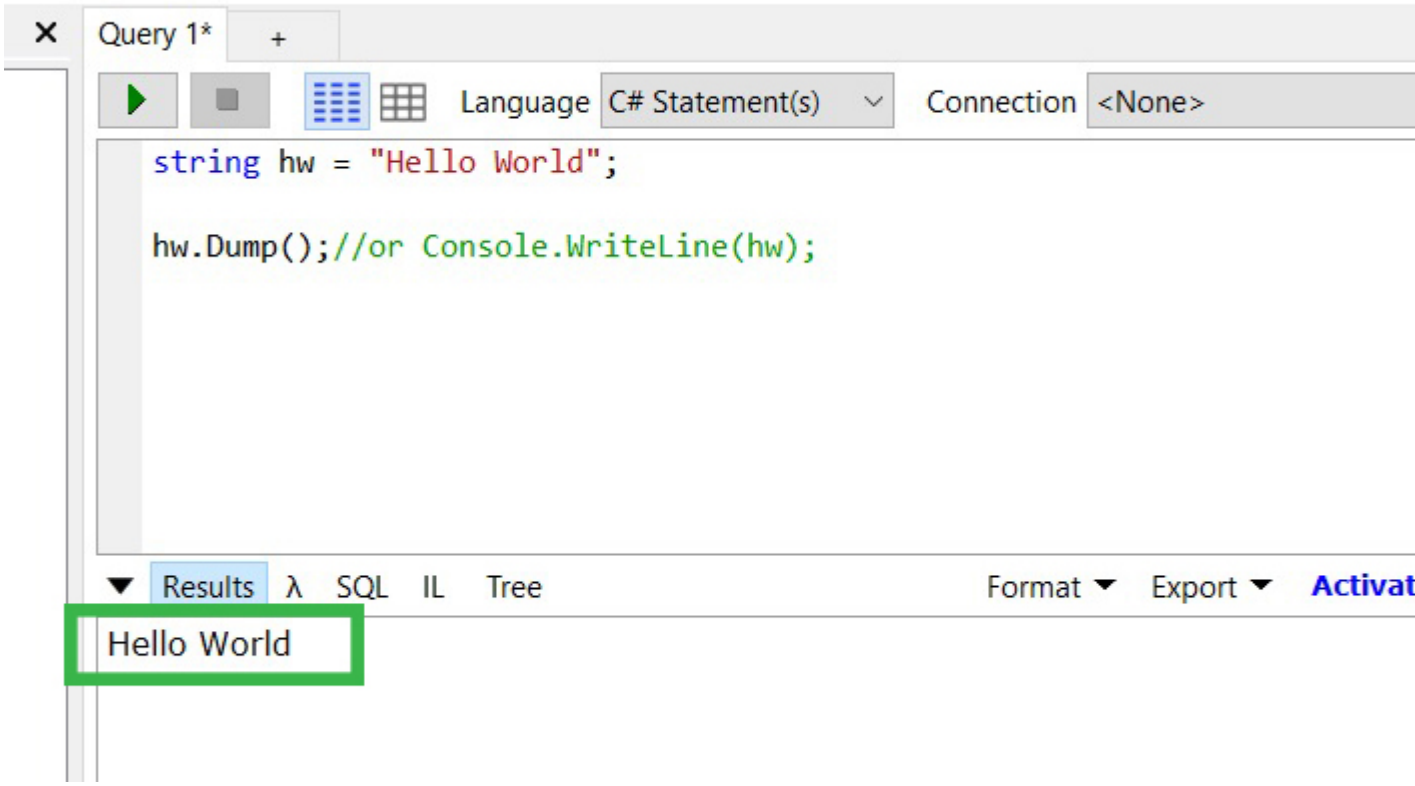


4. Tapez le code suivant et appuyez sur Exécuter (`F5`)

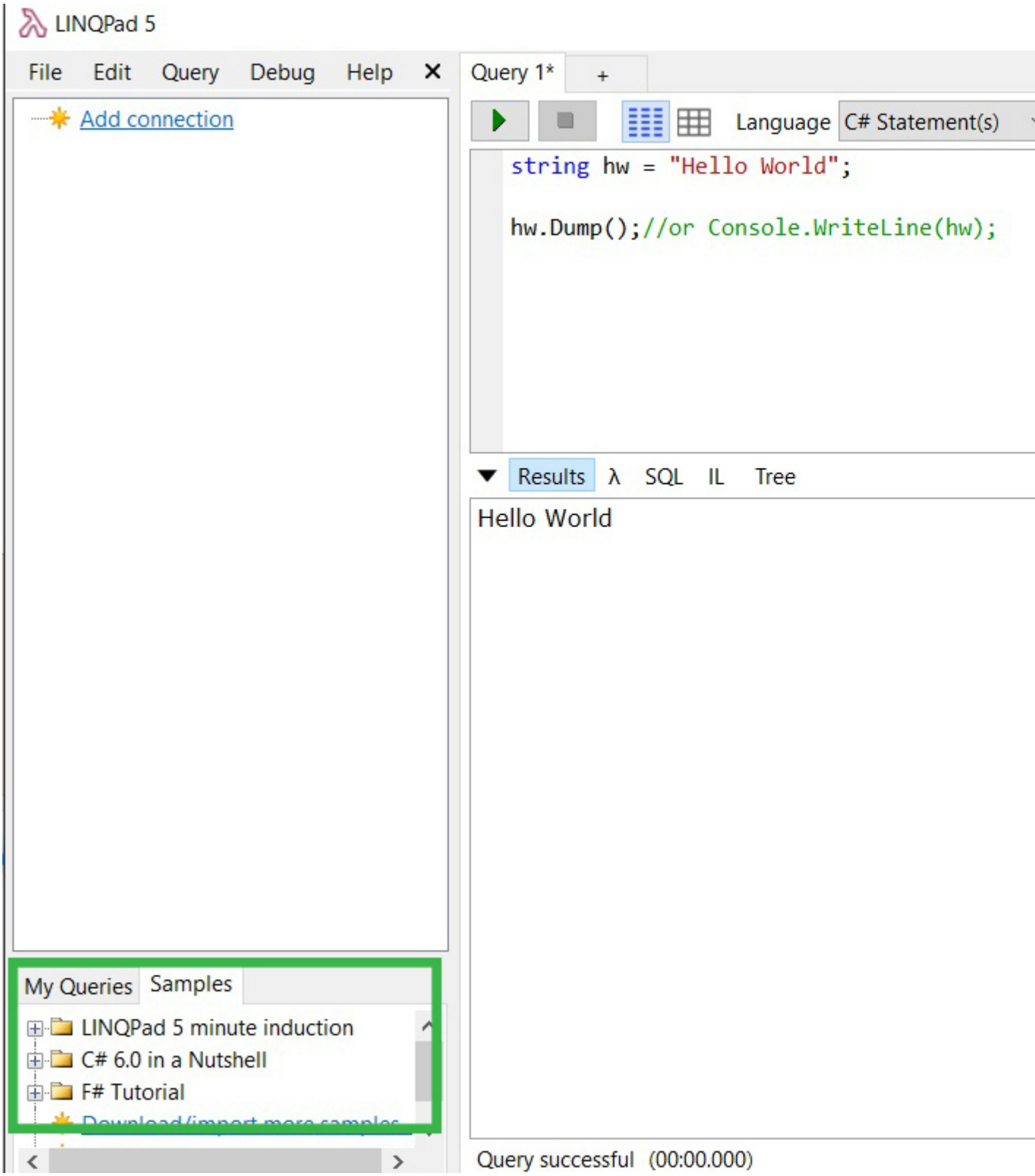
```
string hw = "Hello World";

hw.Dump(); //or Console.WriteLine(hw);
```



5. Vous devriez voir "Hello World" imprimé dans l'écran des résultats.

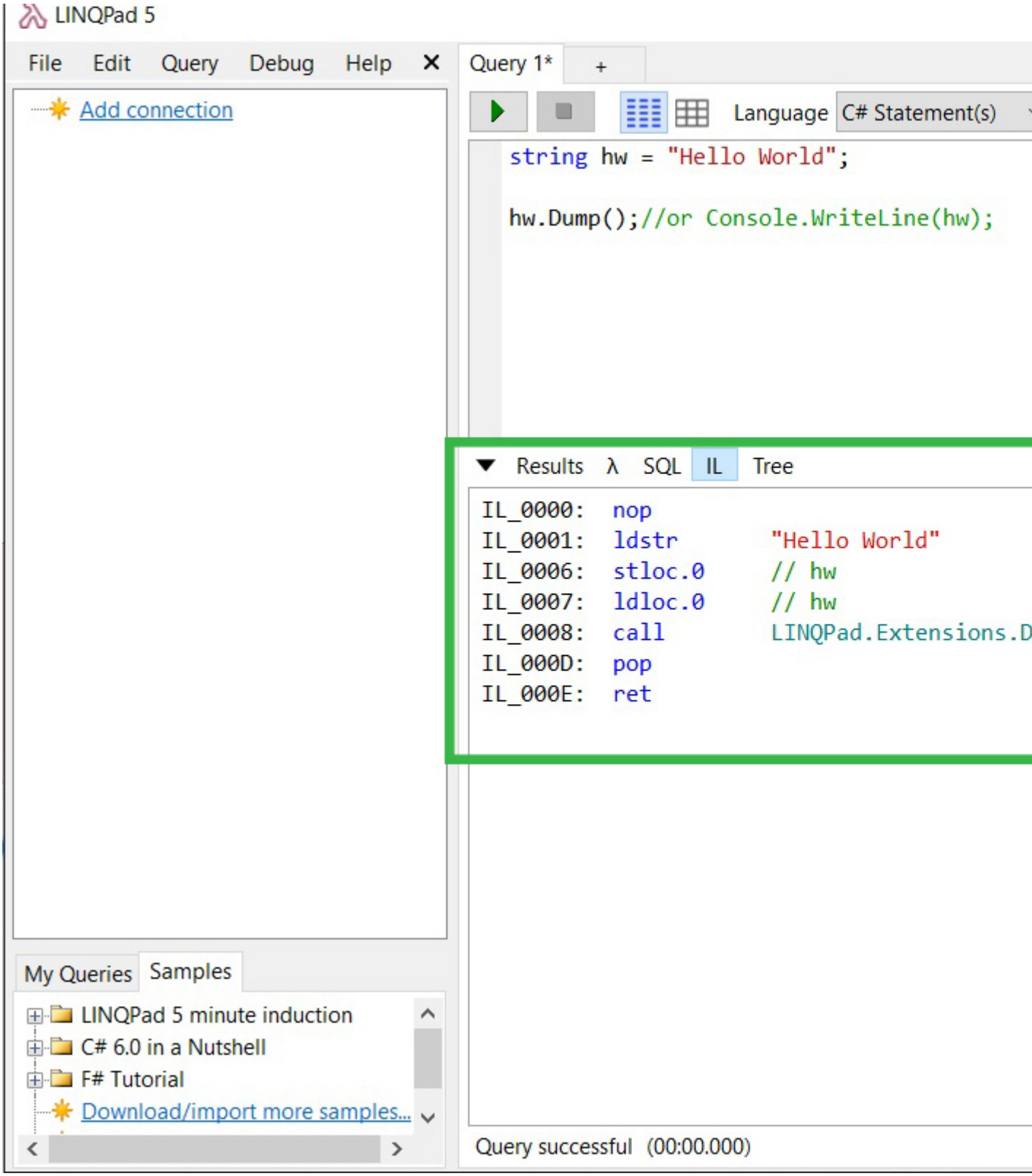


6. Maintenant que vous avez créé votre premier programme .Net, consultez les exemples inclus dans LinqPad via le navigateur "Samples". Il existe de nombreux exemples qui vous montreront de nombreuses fonctionnalités différentes des langues .Net.



Remarques:

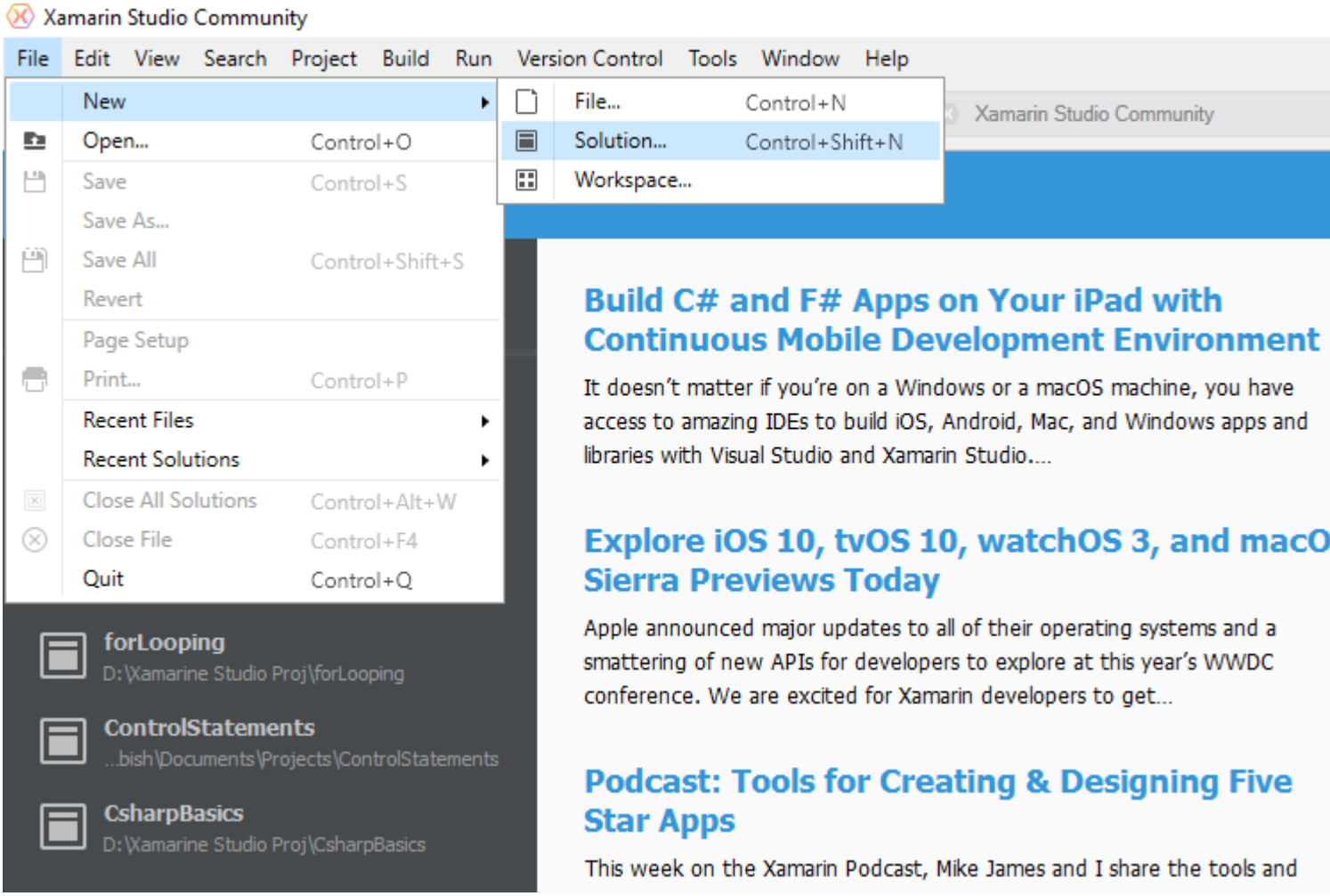
1. Si vous cliquez sur "IL", vous pouvez inspecter le code IL généré par votre code .net. C'est un excellent outil d'apprentissage.



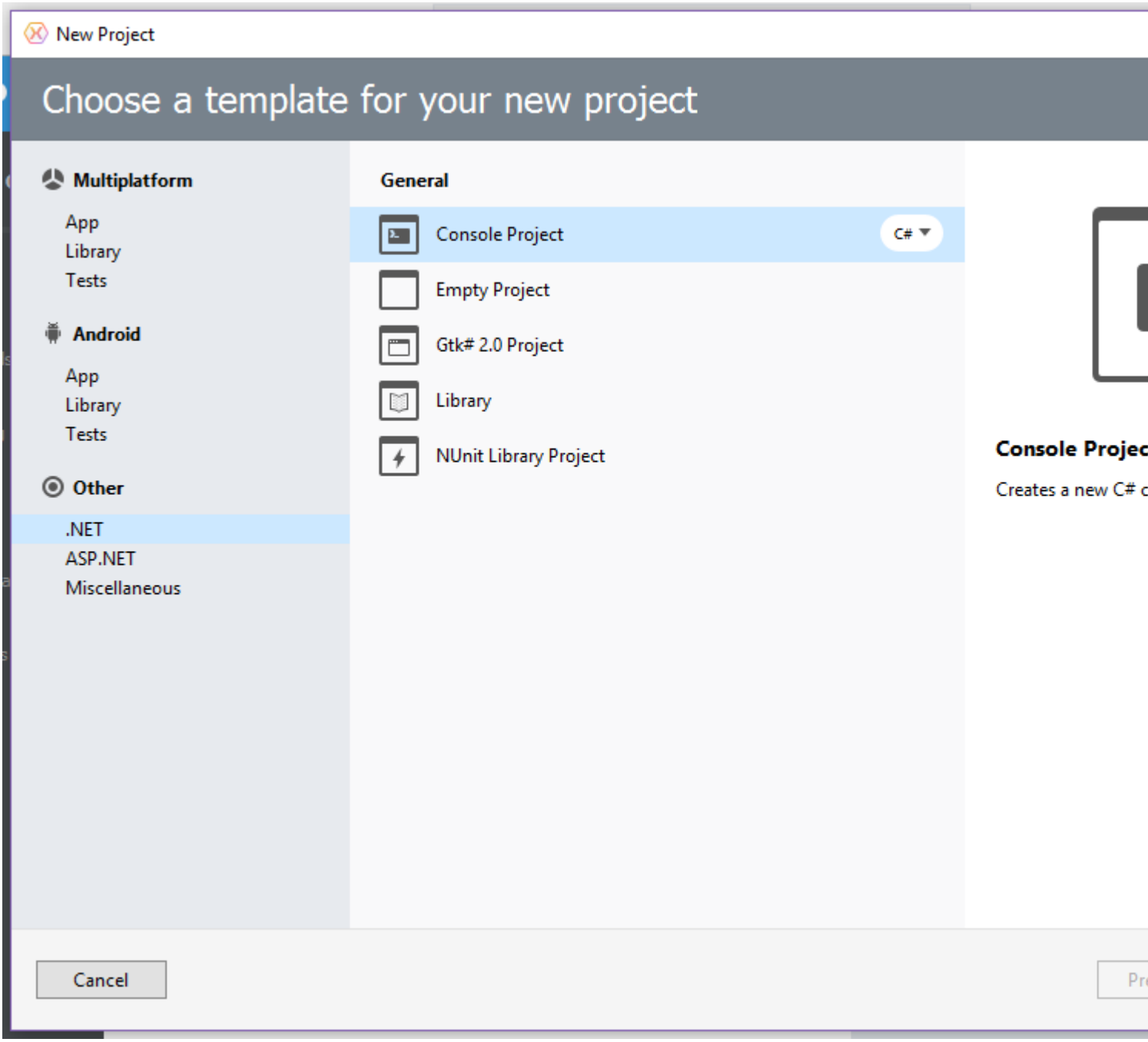
2. Lorsque vous utilisez `LINQ to SQL` ou `Linq to Entities` vous pouvez inspecter le code SQL généré, ce qui constitue un autre excellent moyen d'apprendre à connaître LINQ.

Création d'un nouveau projet à l'aide de Xamarin Studio

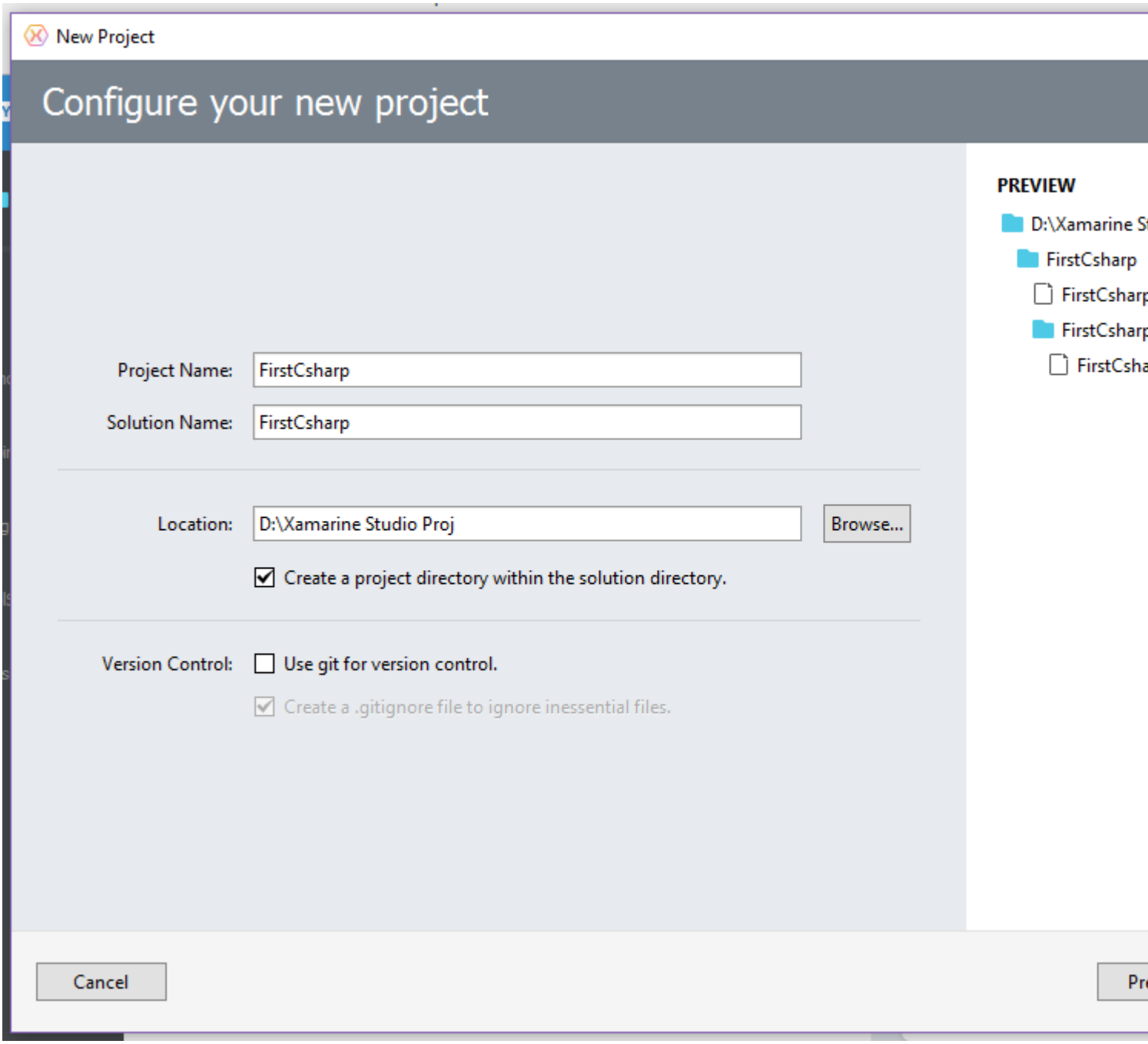
1. Téléchargez et installez Xamarin Studio Community .
2. Ouvrez Xamarin Studio.
3. Cliquez sur **Fichier** → **Nouveau** → **Solution** .



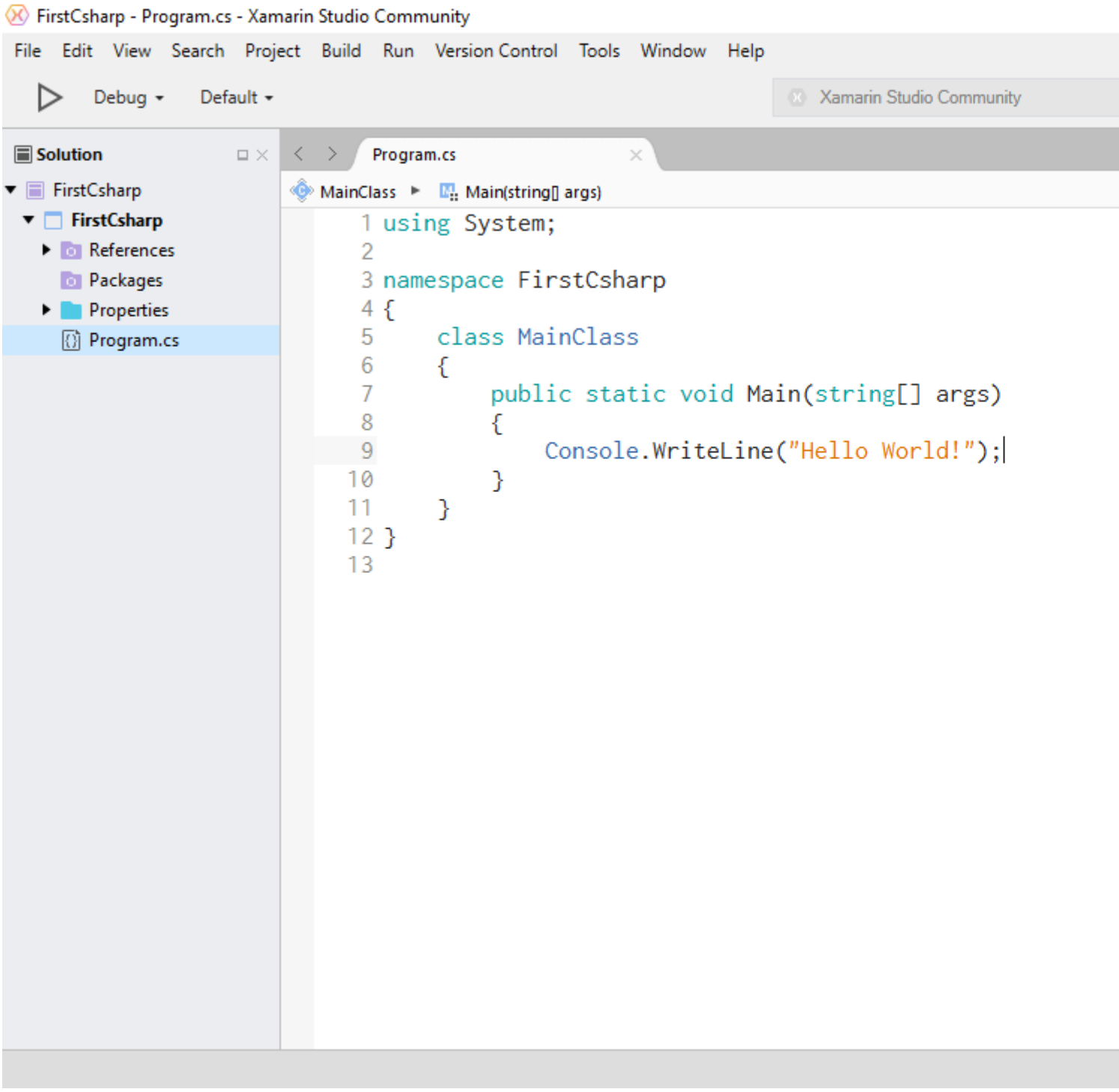
4. Cliquez sur **.NET** → **Projet de console** et choisissez **C #** .
5. Cliquez sur `Suivant` pour continuer.



6. Entrez le **nom** du **projet** et `Parcourir ...` pour un **emplacement** à enregistrer, puis cliquez sur `Créer` .



7. Le projet nouvellement créé ressemblera à:

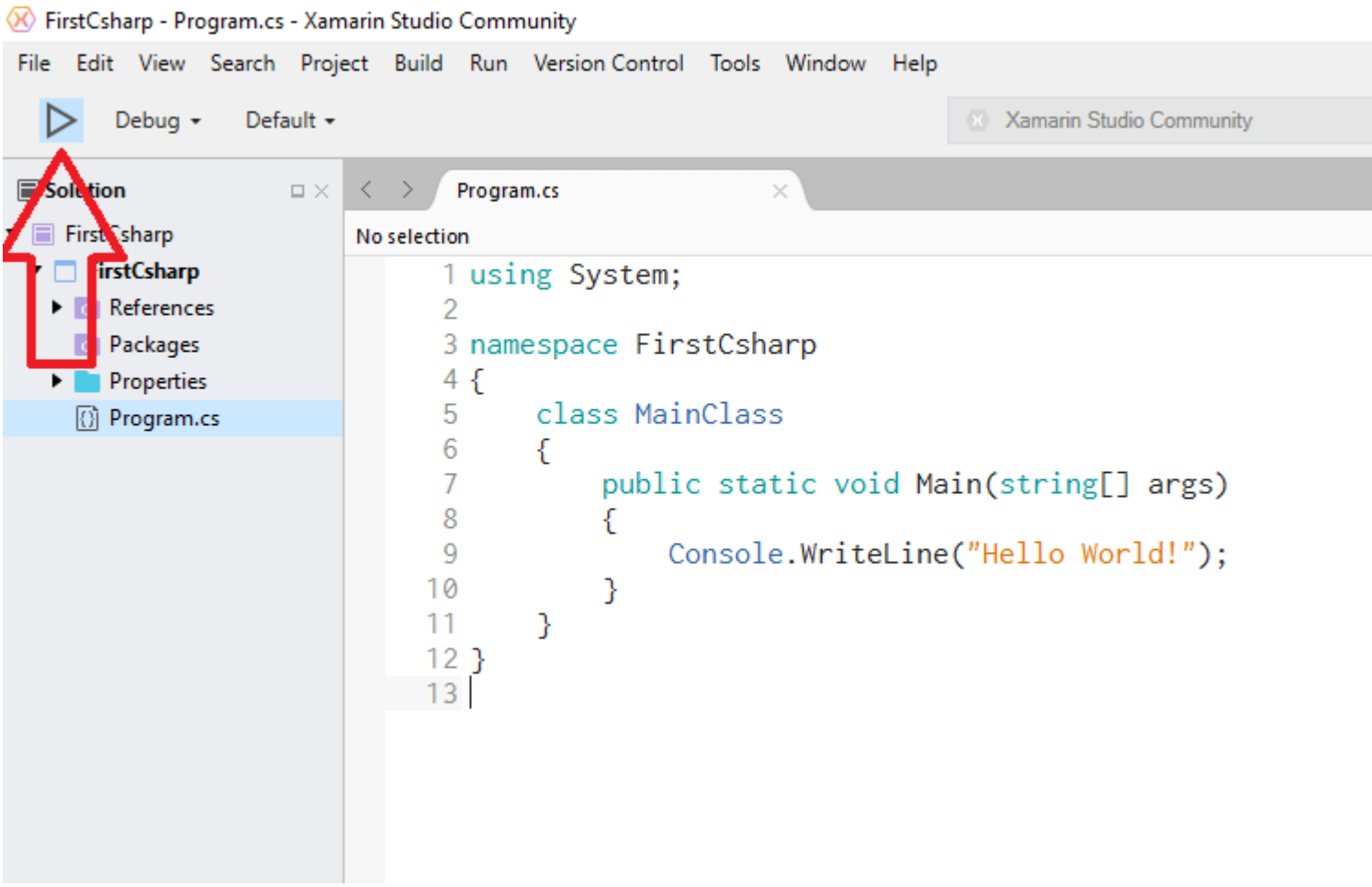


8. Ceci est le code dans l'éditeur de texte:

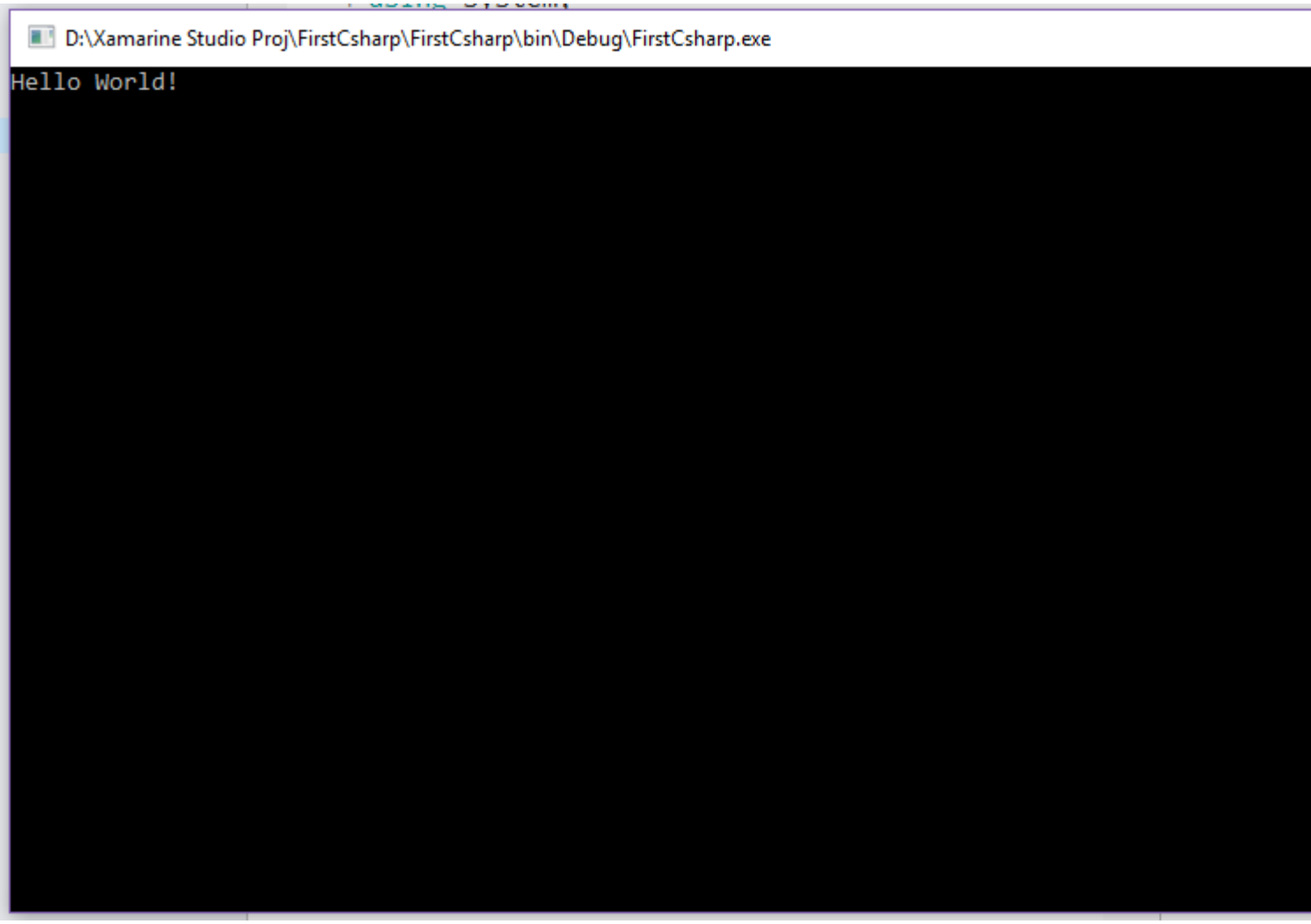
```
using System;

namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
```

```
}
```

9. Pour exécuter le code, appuyez sur `F5` ou cliquez sur le **bouton de lecture** comme indiqué ci-dessous:



10. Voici la sortie:



Lire Démarrer avec le langage C # en ligne: https://riptutorial.com/fr/csharp/topic/15/demarrer-avec-le-langage-c-sharp

Chapitre 2: Accéder au dossier partagé du réseau avec le nom d'utilisateur et le mot de passe

Introduction

Accéder au fichier de partage réseau à l'aide de PInvoke.

Examples

Code pour accéder au fichier partagé du réseau

```
public class NetworkConnection : IDisposable
    {
        string _networkName;

        public NetworkConnection(string networkName,
            NetworkCredential credentials)
        {
            _networkName = networkName;

            var netResource = new NetResource()
            {
                Scope = ResourceScope.GlobalNetwork,
                ResourceType = ResourceType.Disk,
                DisplayType = ResourceDisplaytype.Share,
                RemoteName = networkName
            };

            var userName = string.IsNullOrEmpty(credentials.Domain)
                ? credentials.UserName
                : string.Format(@"{0}\{1}", credentials.Domain, credentials.UserName);

            var result = WNetAddConnection2(
                netResource,
                credentials.Password,
                userName,
                0);

            if (result != 0)
            {
                throw new Win32Exception(result);
            }
        }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
```

```
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);

    [DllImport("mpr.dll")]
    private static extern int WNetCancelConnection2(string name, int flags,
        bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
```

```
    Tree = 0x0a,
    Ndscontainer = 0x0b
}
```

Lire Accéder au dossier partagé du réseau avec le nom d'utilisateur et le mot de passe en ligne: https://riptutorial.com/fr/csharp/topic/9627/acceder-au-dossier-partage-du-reseau-avec-le-nom-d-utilisateur-et-le-mot-de-passe

Chapitre 3: Accéder aux bases de données

Examples

Connexions ADO.NET

Les connexions ADO.NET sont l'un des moyens les plus simples de se connecter à une base de données à partir d'une application C #. Ils reposent sur l'utilisation d'un fournisseur et d'une chaîne de connexion qui pointe vers votre base de données pour effectuer des requêtes.

Classes communes de fournisseur de données

La plupart des classes suivantes sont couramment utilisées pour interroger les bases de données et leurs espaces de noms associés:

- `SqlConnection` , `SqlCommand` , `SqlDataReader` de `System.Data.SqlClient`
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` partir de `System.Data.OleDb`
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` partir de `MySql.Data`

Tous ces éléments sont couramment utilisés pour accéder aux données via C # et seront couramment rencontrés lors de la création d'applications centrées sur les données. De nombreuses autres classes non mentionnées qui implémentent les mêmes `FooConnection` , `FooCommand` , `FooDataReader` peuvent se comporter de la même manière.

Modèle d'accès commun pour les connexions ADO.NET

Un modèle commun pouvant être utilisé pour accéder à vos données via une connexion ADO.NET peut se présenter comme suit:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

Ou si vous réalisiez une simple mise à jour et que vous n'avez pas besoin de lecteur, le même concept de base s'appliquerait:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

Vous pouvez même programmer sur un ensemble d'interfaces communes et ne pas avoir à vous soucier des classes spécifiques au fournisseur. Les interfaces principales fournies par ADO.NET sont:

- IDbConnection - pour gérer les connexions aux bases de données
- IDbCommand - pour exécuter des commandes SQL
- IDbTransaction - pour gérer les transactions
- IDataReader - pour lire les données renvoyées par une commande
- IDataAdapter - pour canaliser des données vers et depuis des ensembles de données

```
var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; //for Oracle use
"Oracle.ManagedDataAccess.Client"
//most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}";    //this needs to be tailored for each database
system

        using(var reader = command.ExecuteReader()) {
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Connexions Entity Framework

Entity Framework expose les classes d'abstraction utilisées pour interagir avec les bases de données sous-jacentes sous la forme de classes telles que `DbContext` . Ces contextes sont généralement des `DbSet<T>` qui exposent les collections disponibles pouvant être interrogées:

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

`DbContext` lui-même les connexions avec les bases de données et lit généralement les données de chaîne de connexion appropriées dans une configuration pour déterminer comment établir les connexions:

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
    public ExampleContext() : base("ExampleContextEntities")
    {
    }

    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Exécution de requêtes Entity Framework

Effectivement, l'exécution d'une requête Entity Framework peut être assez simple et nécessite simplement de créer une instance du contexte, puis d'utiliser les propriétés disponibles pour extraire ou accéder à vos données.

```
using(var context = new ExampleContext())
{
      // Retrieve all of the Widgets in your database
      var data = context.Widgets.ToList();
}
```

Entity Framework fournit également un système complet de suivi des modifications qui peut être utilisé pour gérer les entrées de mise à jour dans votre base de données en appelant simplement la méthode `SaveChanges()` pour transmettre les modifications à la base de données:

```
using(var context = new ExampleContext())
{
      // Grab the widget you wish to update
      var widget = context.Widgets.Find(w => w.Id == id);
      // If it exists, update it
      if(widget != null)
      {
           // Update your widget and save your changes
           widget.Updated = DateTime.UtcNow;
           context.SaveChanges();
      }
}
```

Chaînes de connexion

Une chaîne de connexion est une chaîne qui spécifie des informations sur une source de données

particulière et explique comment s'y connecter en stockant des informations d'identification, des emplacements et d'autres informations.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Stockage de votre chaîne de connexion

En règle générale, une chaîne de connexion sera stockée dans un fichier de configuration (tel que `app.config` ou `web.config` dans les applications ASP.NET). Voici un exemple de ce que peut être une connexion locale dans l'un de ces fichiers:

```
<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;"/>
</connectionStrings>

<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;"/>
</connectionStrings>
```

Cela permettra à votre application d'accéder à la chaîne de connexion par programmation via `WidgetsContext` . Bien que `Integrated Security=SSPI` et `Integrated Security=True` remplissent tous deux la même fonction; `Integrated Security=SSPI` est préférable car il fonctionne avec le fournisseur SQLClient & OleDB où `Integrated Security=true` renvoie une exception lorsqu'il est utilisé avec le fournisseur OleDb.

Différentes connexions pour différents fournisseurs

Chaque fournisseur de données (SQL Server, MySQL, Azure, etc.) présente tous sa propre syntaxe de syntaxe pour ses chaînes de connexion et expose différentes propriétés disponibles. ConnectionStrings.com est une ressource extrêmement utile si vous ne savez pas à quoi ressembler le vôtre.

Lire Accéder aux bases de données en ligne: https://riptutorial.com/fr/csharp/topic/4811/acceder-aux-bases-de-donnees

Chapitre 4: Alias de types intégrés

Examples

Tableau des types intégrés

Le tableau suivant indique les mots-clés pour les types `C#` intégrés, qui sont des alias de types prédéfinis dans les espaces de noms System.

Type C #	Type de framework .NET
bool	System.Boolean
octet	System.Byte
sbyte	System.SByte
carboniser	System.Char
décimal	System.Decimal
double	System.Double
flotte	System.Single
int	System.Int32
uint	System.UInt32
longue	System.Int64
ulong	System.UInt64
objet	System.Object
court	System.Int16
ushort	System.UInt16
chaîne	System.String

Les mots-clés de type `C#` et leurs alias sont interchangeables. Par exemple, vous pouvez déclarer une variable entière en utilisant l'une des déclarations suivantes:

```
int number = 123;
System.Int32 number = 123;
```

Lire Alias de types intégrés en ligne: https://riptutorial.com/fr/csharp/topic/1862/alias---de-types-integres

Chapitre 5: Analyse de regex

Syntaxe

- `new Regex(pattern);` *// Crée une nouvelle instance avec un modèle défini.*
- `Regex.Match(input);` *// Démarre la recherche et renvoie la correspondance.*
- `Regex.Matches(input);` *// Démarre la recherche et retourne une MatchCollection*

Paramètres

prénom	Détails
Modèle	Le modèle de `string` à utiliser pour la recherche. Pour plus d'informations: msdn
RegexOptions *[Facultatif]*	Les options communes ici sont `Singleline` et `Multiline` . Ils modifient le comportement des éléments de modèle comme le point (.) `NewLine` ne couvre pas une `NewLine` (\ n) en `Multiline-Mode` mais en `SingleLine-Mode` . Comportement par défaut: msdn
Timeout *[Facultatif]*	Lorsque les modèles deviennent plus complexes, la recherche peut prendre plus de temps. C'est le délai d'attente passé pour la recherche, comme cela a été le cas pour la programmation réseau.

Remarques

Nécessaire en utilisant

```
using System.Text.RegularExpressions;
```

Agréable d'avoir

- Vous pouvez tester vos modèles en ligne sans avoir besoin de compiler votre solution pour obtenir des résultats ici: Cliquez sur moi
- Regex101 Exemple: Cliquez sur moi

En particulier, les débutants ont tendance à exagérer leurs tâches avec regex car ils se sentent puissants et au bon endroit pour des recherches plus complexes basées sur du texte. C'est à ce stade que les gens essaient d'analyser des documents XML avec regex sans même se demander s'ils pourraient avoir une classe déjà terminée pour cette tâche, comme `XmlDocument` .

Regex devrait être la dernière arme à prendre en compte la complexité. Au moins, n'oubliez pas de faire quelques efforts pour trouver le `right way` avant d'écrire 20 lignes de motifs.

Examples

Match unique

```
using System.Text.RegularExpressions;
```

```
string pattern = ":(.*?):";
string lookup = "--:text in here:--";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
// Get the match from your regex-object
Match mLookup = rgxLookup.Match(lookup);

// The group-index 0 always covers the full pattern.
// Matches inside parentheses will be accessed through the index 1 and above.
string found = mLookup.Groups[1].Value;
```

Résultat:

```
found = "text in here"
```

Plusieurs correspondances

```
using System.Text.RegularExpressions;
```

```
List<string> found = new List<string>();
string pattern = ":(.*?):";
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
MatchCollection mLookup = rgxLookup.Matches(lookup);

foreach(Match match in mLookup)
{
    found.Add(match.Groups[1].Value);
}
```

Résultat:

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

Lire Analyse de regex en ligne: https://riptutorial.com/fr/csharp/topic/3774/analyse-de-regex

Chapitre 6: Annotation des données

Examples

DisplayNameAttribute (attribut d'affichage)

`DisplayName` définit le nom d'affichage d'une propriété, d'un événement ou d'une méthode vide publique contenant 0 (0) arguments.

```
public class Employee
{
    [DisplayName(@"Employee first name")]
    public string FirstName { get; set; }
}
```

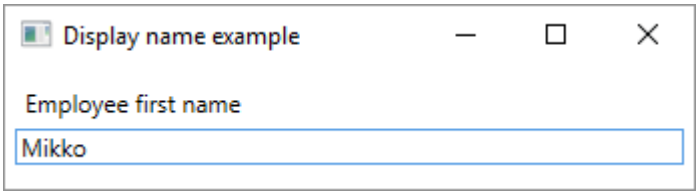
Exemple d'utilisation simple dans l'application XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="100" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:DisplayNameConverter x:Key="DisplayNameConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- Label (DisplayName attribute) -->
        <Label Content="{Binding Employee, Converter={StaticResource DisplayNameConverter},
ConverterParameter=FirstName}" />
        <!-- TextBox (FirstName property value) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>

</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee();

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }
```

```
        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class DisplayNameConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // Get display name for given instance type and property name
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<DisplayNameAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.DisplayName : string.Empty;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```



EditableAttribute (attribut de modélisation de données)

`EditableAttribute` définit si les utilisateurs doivent pouvoir modifier la valeur de la propriété de classe.

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

Exemple d'utilisation simple dans l'application XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="70" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- TextBox Text (FirstName property value) -->
        <!-- TextBox IsEnabled (Editable attribute) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable"};

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // return editable attribute's value for given instance property,
            // defaults to true if not found
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<EditableAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.AllowEdit : true;
```

```
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Attributs de validation

Les attributs de validation sont utilisés pour appliquer diverses règles de validation de manière déclarative aux classes ou aux membres de classe. Tous les attributs de validation proviennent de la classe de base ValidationAttribute .

Exemple: RequiredAttribute

Une fois validé via la méthode `ValidationAttribute.Validate` , cet attribut renvoie une erreur si la propriété `Name` est null ou ne contient que des espaces.

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

Exemple: StringLengthAttribute

`StringLengthAttribute` valide si une chaîne est inférieure à la longueur maximale d'une chaîne. Il peut éventuellement spécifier une longueur minimale. Les deux valeurs sont inclusives.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and
twenty characters.")]
    public string Name { get; set; }
}
```

Exemple: RangeAttribute

Le `RangeAttribute` donne la valeur maximale et minimale pour un champ numérique.

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

Exemple: CustomValidationAttribute

La classe `CustomValidationAttribute` permet d' `CustomValidationAttribute` une méthode `static` personnalisée pour la validation. La méthode personnalisée doit être `static ValidationResult [MethodName] (object input)` .

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Déclaration de méthode:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

Création d'un attribut de validation personnalisé

Des attributs de validation personnalisés peuvent être créés en dérivant de la classe de base `ValidationAttribute` , puis en remplaçant `virtual` méthodes `virtual` si nécessaire.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }
```

```
        return isValid;
    }
}
```

Cet attribut peut alors être utilisé comme ceci:

```
public class Model
{
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]
    public string FavoriteFruit { get; set; }
}
```

Bases d'annotation des données

Les annotations de données permettent d'ajouter plus d'informations contextuelles aux classes ou aux membres d'une classe. Il existe trois grandes catégories d'annotations:

- Attributs de validation: ajout de critères de validation aux données
- Attributs d'affichage: spécifiez comment les données doivent être affichées à l'utilisateur
- Attributs de modélisation: ajoute des informations sur l'utilisation et la relation avec d'autres classes

Usage

Voici un exemple où deux `ValidationAttribute` et un `DisplayAttribute` sont utilisés:

```
class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }
    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars
or more than 50 chars
    public string Name { get; set; }
    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the
time)
    public DateTime Birthday { get; set; }
}
```

Les annotations de données sont principalement utilisées dans des structures telles que ASP.NET. Par exemple, dans `ASP.NET MVC` , lorsqu'un modèle est reçu par une méthode de contrôleur, `ModelState.IsValid()` peut être utilisé pour indiquer si le modèle reçu respecte tous ses `ValidationAttribute` . `DisplayAttribute` est également utilisé dans `ASP.NET MVC` pour déterminer comment afficher les valeurs sur une page Web.

Exécuter manuellement les attributs de validation

La plupart du temps, les attributs de validation sont utilisés dans des frameworks (tels que ASP.NET). Ces frameworks prennent en charge l'exécution des attributs de validation. Mais que faire si vous voulez exécuter les attributs de validation manuellement? Utilisez simplement la classe `Validator` (pas de réflexion nécessaire).

Contexte de validation

Toute validation nécessite un contexte pour donner des informations sur ce qui est en cours de validation. Cela peut inclure diverses informations telles que l'objet à valider, certaines propriétés, le nom à afficher dans le message d'erreur, etc.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of
validation context. It contains only a reference to the object being validated.
```

Une fois le contexte créé, il existe plusieurs façons de procéder à la validation.

Valider un objet et toutes ses propriétés

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates
the object and its properties using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

Valider une propriété d'un objet

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidatePropery(objectToValidate.PropertyToValidate, vc, results,
true); // Validates the property using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

Et plus

Pour en savoir plus sur la validation manuelle, voir:

- Documentation de la classe ValidationContext
- Documentation de classe du validateur

Lire Annotation des données en ligne: https://riptutorial.com/fr/csharp/topic/4942/annotation-des-donnees

Chapitre 7: Arbres d'expression

Introduction

Les arbres d'expression sont des expressions organisées dans une structure de données arborescente. Chaque nœud de l'arborescence est une représentation d'une expression, une expression étant du code. Une représentation en mémoire d'une expression Lambda serait un arbre d'expression qui contiendrait les éléments réels (c'est-à-dire le code) de la requête, mais pas son résultat. Les arbres d'expression rendent la structure d'une expression lambda transparente et explicite.

Syntaxe

- Expression <TDelegate> name = lambdaExpression;

Paramètres

Paramètre	Détails
Délégué TD	Le type de délégué à utiliser pour l'expression
lambdaExpression	L'expression lambda (ex. `num => num < 5`)

Remarques

Introduction aux arbres d'expression

D'où nous venons

Les arborescences d'expression sont toutes axées sur la consommation de "code source" au moment de l'exécution. Envisagez une méthode qui calcule la taxe de vente due sur une `decimal CalculateTotalTaxDue(SalesOrder order)` . Utiliser cette méthode dans un programme .NET est facile - vous appelez simplement `decimal taxDue = CalculateTotalTaxDue(order);` . Que faire si vous souhaitez l'appliquer à tous les résultats d'une requête distante (SQL, XML, un serveur distant, etc.)? Ces sources de requêtes distantes ne peuvent pas appeler la méthode! Traditionnellement, vous devez inverser le flux dans tous ces cas. Effectuez l'intégralité de la requête, stockez-la en mémoire, puis parcourez les résultats et calculez la taxe pour chaque résultat.

Comment éviter les problèmes de mémoire et de latence de l'inversion de flux

Les arbres d'expression sont des structures de données dans un format d'arbre, où chaque noeud contient une expression. Ils sont utilisés pour traduire les instructions compilées (comme les méthodes utilisées pour filtrer les données) dans des expressions qui pourraient être utilisées en dehors de l'environnement du programme, par exemple dans une requête de base de données.

Le problème ici est qu'une requête distante *ne peut pas accéder à notre méthode* . Nous pourrions éviter ce problème si, à la place, nous avons envoyé les *instructions* pour la méthode à la requête distante. Dans notre exemple `CalculateTotalTaxDue` , cela signifie que nous envoyons ces informations:

1. Créer une variable pour stocker la taxe totale
2. Traverser toutes les lignes de la commande
3. Pour chaque ligne, vérifiez si le produit est taxable
4. Si c'est le cas, multipliez le total de la ligne par le taux de taxe applicable et ajoutez ce montant au total.
5. Sinon ne rien faire

Avec ces instructions, la requête distante peut exécuter le travail lors de la création des données.

Il y a deux défis à relever pour la mettre en œuvre. Comment transformer une méthode .NET compilée en une liste d'instructions et comment formater les instructions de manière à ce qu'elles puissent être utilisées par le système distant?

Sans arbres d'expression, vous ne pouvez résoudre le premier problème avec MSIL. (MSIL est le code de type assembleur créé par le compilateur .NET.) L'analyse MSIL est *possible* , mais ce n'est pas facile. Même si vous analysez correctement le contenu, il peut être difficile de déterminer quelle était l'intention du programmeur d'origine avec une routine particulière.

Les arbres d'expression sauvent la journée

Les arbres d'expression traitent ces problèmes exacts. Ils représentent des instructions de programme dans une structure de données arborescente où chaque nœud représente *une instruction* et contient des références à toutes les informations dont vous avez besoin pour exécuter cette instruction. Par exemple, un objet `MethodCallExpression` fait référence à 1) le `MethodInfo` qu'il va appeler, 2) une liste d' `Expression` qu'il passera à cette méthode, 3) pour les méthodes d'instance, l' `Expression` que vous appellerez la méthode. Vous pouvez "parcourir l'arborescence" et appliquer les instructions sur votre requête distante.

Créer des arbres d'expression

Le moyen le plus simple de créer un arbre d'expression est d'utiliser une expression lambda. Ces expressions sont presque identiques aux méthodes C # normales. Il est important de réaliser que c'est *la magie du compilateur* . Lorsque vous créez une expression lambda pour la première fois, le compilateur vérifie ce que vous lui attribuez. S'il s'agit d'un type `Delegate` (y compris `Action` ou `Func`), le compilateur convertit l'expression lambda en un délégué. Si c'est un `LambdaExpression` (ou une `Expression<Action<T>>` ou `Expression<Func<T>>` qui est fortement typé `LambdaExpression`), le compilateur le transforme en `LambdaExpression` . C'est là `LambdaExpression` la magie. En arrière-plan, le compilateur *utilise l'API API* pour transformer votre expression lambda en une expression

`LambdaExpression` .

Les expressions lambda ne peuvent pas créer chaque type d'arbre d'expression. Dans ces cas, vous pouvez utiliser l'API Expressions manuellement pour créer l'arborescence requise. Dans l'exemple de l' API Comprendre les expressions , nous créons l'expression `CalculateTotalSalesTax` à l'aide de l'API.

NOTE: Les noms sont un peu confus ici. Une *expression lambda* (deux mots, minuscule) fait référence au bloc de code avec un indicateur `=>` . Il représente une méthode anonyme en C # et est converti en `Delegate` ou `Expression` . Un *`LambdaExpression`* (un mot, PascalCase) fait référence au type de noeud dans l'API Expression qui représente une méthode que vous pouvez exécuter.

Arbres d'expression et LINQ

L'une des utilisations les plus courantes des arborescences d'expression concerne les requêtes LINQ et de base de données. LINQ associe une arborescence d'expression à un fournisseur de requêtes pour appliquer vos instructions à la requête distante cible. Par exemple, le fournisseur de requêtes LINQ to Entity Framework transforme un arbre d'expression en SQL qui est exécuté directement sur la base de données.

En rassemblant toutes les pièces, vous pouvez voir le véritable pouvoir derrière LINQ.

1. Ecrivez une requête en utilisant une expression lambda: `products.Where(x => x.Cost > 5)`
2. Le compilateur transforme cette expression en une arborescence d'expression avec les instructions "vérifier si la propriété Cost du paramètre est supérieure à cinq".
3. Le fournisseur de requêtes analyse l'arborescence des expressions et génère une requête SQL valide `SELECT * FROM products WHERE Cost > 5`
4. L'ORM projette tous les résultats dans des POCO et vous obtenez une liste d'objets

Remarques

- Les arbres d'expression sont immuables. Si vous souhaitez modifier une arborescence d'expression pour en créer une nouvelle, copiez celle existante dans la nouvelle (pour parcourir une arborescence d'expression que vous pouvez utiliser avec `ExpressionVisitor`) et apportez les modifications souhaitées.

Examples

Création d'arbres d'expression à l'aide de l'API

```
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 });
```

Compilation d'arbres d'expression

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
Console.WriteLine(expr.Compile()(4));
```

Analyse d'arbres d'expression

```
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);

// Decomposed expression: num => num LessThan 5
```

Créer des arbres d'expression avec une expression lambda

Voici l'arbre d'expression le plus fondamental créé par lambda.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

Pour créer des arbres d'expression "à la main", il faut utiliser la classe `Expression` .

L'expression ci-dessus serait équivalente à:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two
expressions (num == 42)
Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter);
```

Comprendre l'API des expressions

Nous allons utiliser l'API API pour créer un arbre `CalculateSalesTax` . En clair, voici un résumé des étapes nécessaires à la création de l'arborescence.

1. Vérifiez si le produit est taxable
2. Si c'est le cas, multipliez le total de la ligne par le taux de taxe applicable et retournez ce montant
3. Sinon retourne 0

```
//For reference, we're using the API to build this lambda expression
    orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method.  We specify it's type (OrderLine) and the
name of the parameter.
    ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable;  First we need to access the is taxable property, then
check if it's true
    PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
    MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
    UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);

//Before creating the if, we need to create the braches
    //If the line is taxable, we'll return the total times the tax rate; get the total and tax
rate, then multiply
    //Get the total
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //Get the order
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //Get the tax rate - notice that we pass the getOrder expression directly to the member
access
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //Multiply the two - notice we pass the two operand expressions directly to multiply
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

//If the line is not taxable, we'll return a constant value - 0.0 (decimal)
    ConstantExpression zero = Expression.Constant(0M);

//Create the actual if check and branches
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
multiplyTotalByRate, zero);

//Wrap the whole thing up in a "method" - a LambdaExpression
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
decimal>>(ifTaxableTernary, orderLine);
```

Arbre d'expression de base

Les arbres d'expression représentent du code dans une structure de données arborescente, où

chaque nœud est une expression

Expression Trees permet la modification dynamique du code exécutable, l'exécution de requêtes LINQ dans diverses bases de données et la création de requêtes dynamiques. Vous pouvez compiler et exécuter du code représenté par des arbres d'expression.

Celles-ci sont également utilisées dans le DLR (Dynamic Language Runtime) pour assurer l'interopérabilité entre les langages dynamiques et le .NET Framework et pour permettre aux auteurs du compilateur d'émettre des arborescences d'expression au lieu du langage intermédiaire Microsoft (MSIL).

Des arbres d'expression peuvent être créés via

1. Expression lambda anonyme,
2. Manuellement en utilisant l'espace de noms System.Linq.Expressions.

Arbres d'expression des expressions lambda

Lorsqu'une expression lambda est affectée à une variable de type Expression, le compilateur émet du code pour générer une arborescence d'expression qui représente l'expression lambda.

Les exemples de code suivants montrent comment faire en sorte que le compilateur C # crée un arbre d'expression qui représente l'expression lambda num => num <5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Arbres d'expression à l'aide de l'API

Les arbres d'expression ont également été créés à l'aide de la classe d' **expression** . Cette classe contient des méthodes de fabrique statiques qui créent des nœuds d'arbre d'expression de types spécifiques.

Vous trouverez ci-dessous quelques types de nœuds d'arbre.

1. ParameterExpression
2. MethodCallExpression

L'exemple de code suivant montre comment créer une arborescence d'expression qui représente l'expression lambda num => num <5 en utilisant l'API.

```
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 });
```

Examen de la structure d'une expression à l'aide de Visitor

Définissez une nouvelle classe de visiteur en remplaçant certaines des méthodes d' ExpressionVisitor :

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Appelez `Visit` pour utiliser ce visiteur sur une expression existante:

```
Expression<Func<int,bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Lire Arbres d'expression en ligne: https://riptutorial.com/fr/csharp/topic/75/arbres-d-expression

Chapitre 8: Arguments nommés

Examples

Les arguments nommés peuvent rendre votre code plus clair

Considérez cette classe simple:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
    {
         // Some code
    }
}
```

Avant C # 3.0 c'était:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

vous pouvez rendre cet appel de méthode encore plus clair avec les **arguments nommés** :

```
var result = SmsUtil.SendMessage(
    from: "Mehran",
    to:  "Maryam",
    message "Hello there!",
    retryCount: 12,
    attachment: null);
```

Arguments nommés et paramètres facultatifs

Vous pouvez combiner des arguments nommés avec des paramètres facultatifs.

Laissez voir cette méthode:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
         // Some code
    }
}
```

Lorsque vous souhaitez appeler cette méthode *sans* définir l'argument `retryCount` :

```
var result = SmsUtil.SendMessage(
                        from       : "Cihan",
                        to         : "Yakar",
```

```
                        message    : "Hello there!",
                        attachment : new object());
```

L'ordre des arguments n'est pas nécessaire

Vous pouvez placer des arguments nommés dans l'ordre de votre choix.

Méthode d'échantillon:

```
public static string Sample(string left, string right)
{
     return string.Join("-",left,right);
}
```

Exemple d'appel:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

Résultats:

```
A-B
B-A
```

Les arguments nommés évitent les bogues sur les paramètres facultatifs

Toujours utiliser les arguments nommés pour les paramètres facultatifs, afin d'éviter les bogues potentiels lorsque la méthode est modifiée.

```
class Employee
{
    public string Name { get; private set; }

    public string Title { get; set; }

    public Employee(string name = "<No Name>", string title = "<No Title>")
    {
        this.Name = name;
        this.Title = title;
    }
}

var jack = new Employee("Jack", "Associate");   //bad practice in this line
```

Le code ci-dessus compile et fonctionne correctement, jusqu'à ce que le constructeur soit modifié un jour comme:

```
//Evil Code: add optional parameters between existing optional parameters
public Employee(string name = "<No Name>", string department = "intern", string title = "<No
Title>")
{
    this.Name = name;
```

```
    this.Department = department;
    this.Title = title;
}

//the below code still compiles, but now "Associate" is an argument of "department"
var jack = new Employee("Jack", "Associate");
```

Meilleure pratique pour éviter les bugs lorsque "quelqu'un d'autre dans l'équipe" a commis des erreurs:

```
var jack = new Employee(name: "Jack", title: "Associate");
```

Lire Arguments nommés en ligne: https://riptutorial.com/fr/csharp/topic/2076/arguments-nommes

Chapitre 9: Arguments nommés et facultatifs

Remarques

Arguments nommés

Ref: Les arguments nommés *MSDN* vous permettent de spécifier un argument pour un paramètre particulier en associant l'argument au nom du paramètre plutôt qu'à la position du paramètre dans la liste des paramètres.

Comme indiqué par MSDN, un argument nommé,

- Vous permet de passer l'argument à la fonction en associant le nom du paramètre.
- Pas besoin de se souvenir de la position des paramètres que nous ne connaissons pas toujours.
- Pas besoin de regarder l'ordre des paramètres dans la liste de paramètres de la fonction appelée.
- Nous pouvons spécifier le paramètre pour chaque argument par son nom.

Arguments optionnels

Ref: MSDN La définition d'une méthode, d'un constructeur, d'un indexeur ou d'un délégué peut spécifier que ses paramètres sont obligatoires ou qu'ils sont facultatifs. Tout appel doit fournir des arguments pour tous les paramètres requis, mais peut omettre des arguments pour des paramètres facultatifs.

Comme indiqué par MSDN, un argument facultatif,

- Nous pouvons omettre l'argument dans l'appel si cet argument est un argument optionnel
- Chaque argument optionnel a sa propre valeur par défaut
- Il faudra une valeur par défaut si nous ne fournissons pas la valeur
- Une valeur par défaut d'un argument facultatif doit être un
    - Expression constante
    - Doit être un type de valeur tel que enum ou struct.
    - Doit être une expression du formulaire par défaut (valueType)
- Il doit être défini à la fin de la liste des paramètres

Examples

Arguments nommés

Considérez ce qui suit est notre appel de fonction.

```
FindArea(120, 56);
```

Dans ce cas, notre premier argument est length (ie 120) et le second argument est width (ie 56).

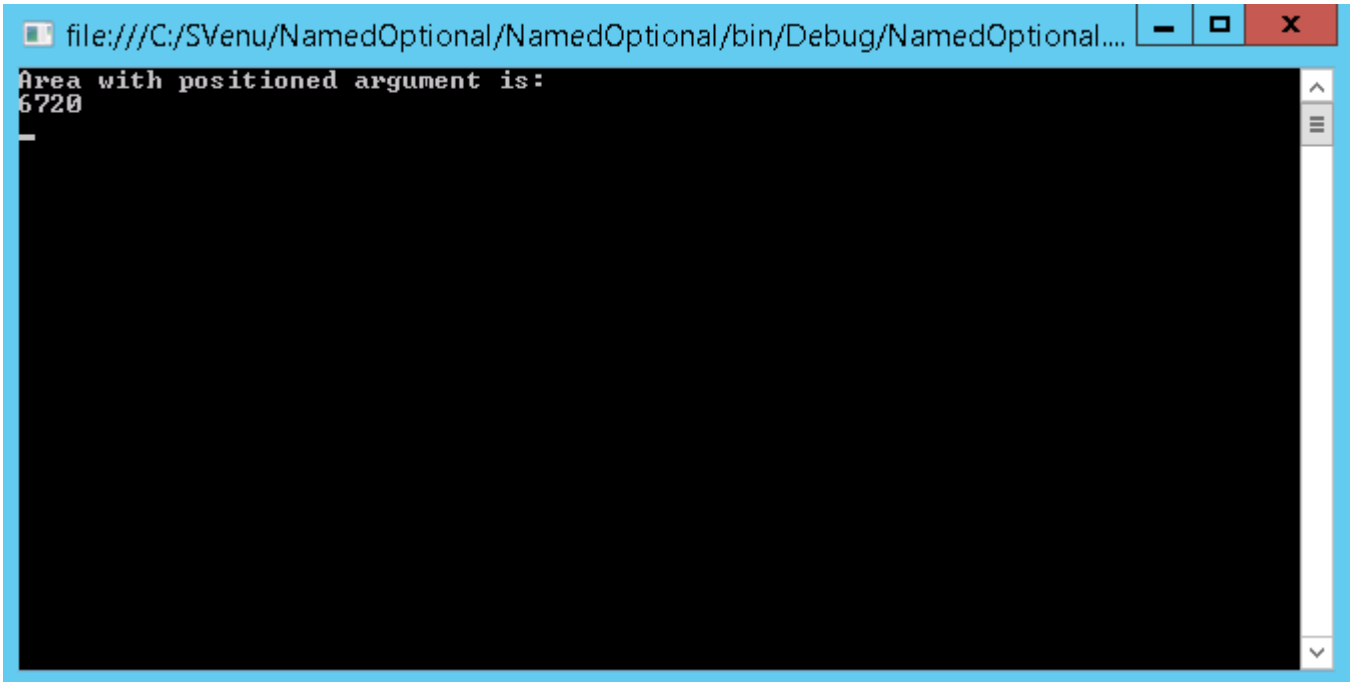
Et nous calculons la superficie par cette fonction. Et voici la définition de la fonction.

```
private static double FindArea(int length, int width)
       {
           try
           {
               return (length* width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Donc, dans le premier appel de fonction, nous venons de passer les arguments par sa position. Droite?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

Si vous exécutez ceci, vous obtiendrez une sortie comme suit.



Maintenant, voici les caractéristiques d'un argument nommé. Veuillez consulter l'appel de fonction précédent.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

Nous donnons ici les arguments nommés dans l’appel de méthode.

```
area = FindArea(length: 120, width: 56);
```

Maintenant, si vous exécutez ce programme, vous obtiendrez le même résultat. Nous pouvons donner les noms vice-versa dans l'appel à la méthode si nous utilisons les arguments nommés.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

L'une des utilisations importantes d'un argument nommé est que, lorsque vous l'utilisez dans votre programme, il améliore la lisibilité de votre code. Il dit simplement ce que votre argument doit être, ou ce que c'est?

Vous pouvez aussi donner les arguments de position. Cela signifie une combinaison de l'argument positionnel et de l'argument nommé.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
            area = FindArea(120, width: 56);
            Console.WriteLine(area);
            Console.Read();
```

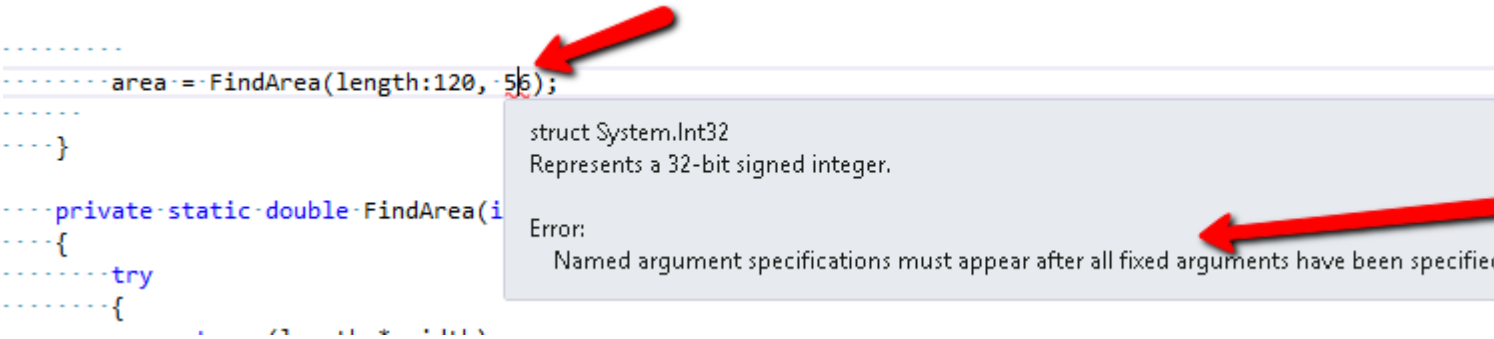
Dans l'exemple ci-dessus, nous avons passé 120 en tant que longueur et 56 en tant qu'argument nommé pour la largeur du paramètre.

Il y a aussi des limites. Nous allons discuter de la limitation d'un argument nommé maintenant.

Limitation de l'utilisation d'un argument nommé

La spécification d'argument nommé doit apparaître après que tous les arguments fixes ont été spécifiés.

Si vous utilisez un argument nommé avant un argument fixe, vous obtiendrez une erreur de compilation comme suit.



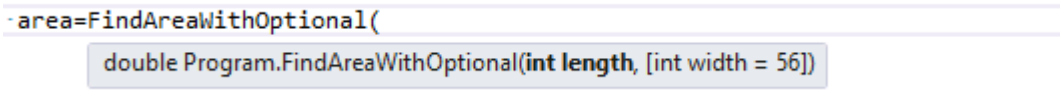
La spécification d'argument nommé doit apparaître après que tous les arguments fixes ont été spécifiés

Arguments optionnels

Considérer précédent est notre définition de fonction avec des arguments facultatifs.

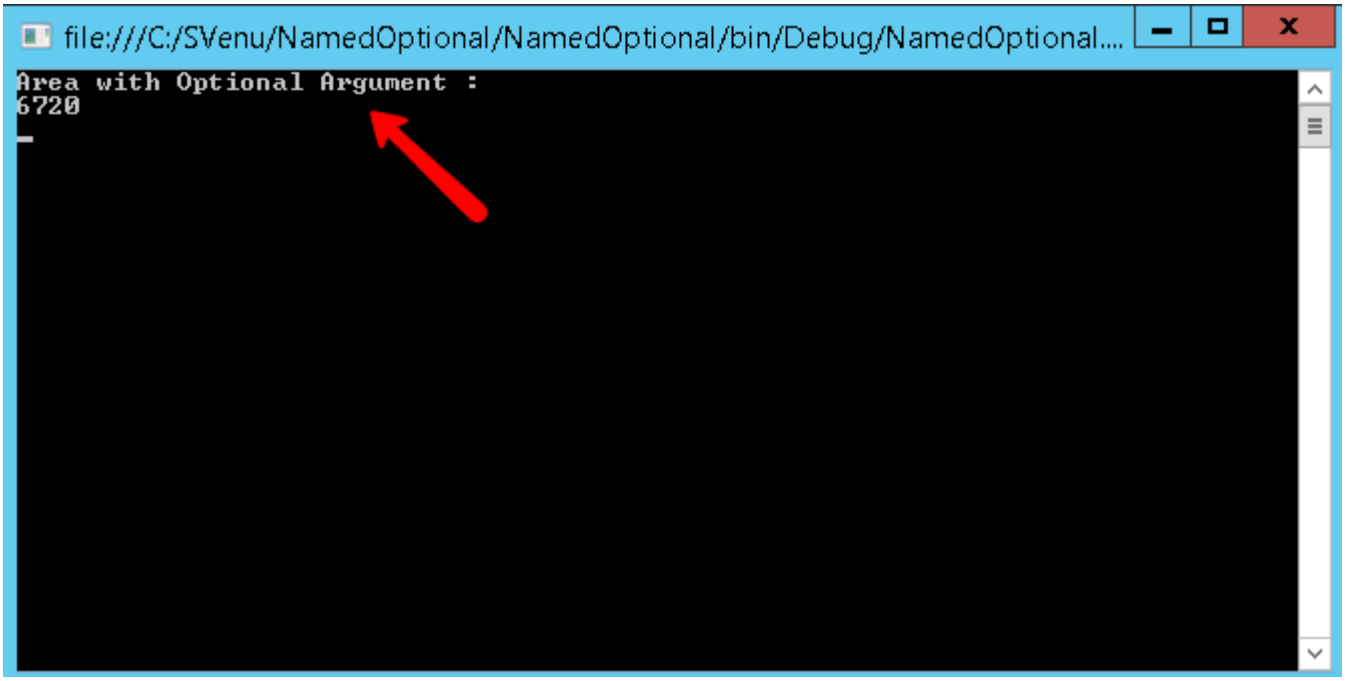
```
private static double FindAreaWithOptional(int length, int width=56)
       {
           try
           {
               return (length * width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Ici, nous avons défini la valeur de largeur comme facultative et nous avons donné la valeur 56. Si vous remarquez, l'IntelliSense lui-même affiche l'argument facultatif, comme indiqué dans l'image ci-dessous.




```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Notez que nous n'avons pas eu d'erreur lors de la compilation et cela vous donnera un résultat comme suit.



Utiliser un attribut facultatif.

Une autre façon d'implémenter l'argument optionnel consiste à utiliser le mot clé `[Optional]` . Si vous ne transmettez pas la valeur de l'argument facultatif, la valeur par défaut de ce type de

données est affectée à cet argument. Le mot clé `Optional` est présent dans l'espace de noms «Runtime.InteropServices».

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
   {
       try
       {
           return (length * width);
       }
       catch (Exception)
       {
           throw new NotImplementedException();
       }
   }

area = FindAreaWithOptional(120);  //area=0
```

Et quand nous appelons la fonction, nous obtenons 0 car le deuxième argument n'est pas passé et la valeur par défaut de int est 0 et donc le produit est 0.

Lire Arguments nommés et facultatifs en ligne:
https://riptutorial.com/fr/csharp/topic/5220/arguments-nommes-et-facultatifs

Chapitre 10: AssemblyInfo.cs Exemples

Remarques

Le nom de fichier `AssemblyInfo.cs` est utilisé par convention comme fichier source où les développeurs placent des attributs de métadonnées qui décrivent l'assemblage complet qu'ils construisent.

Examples

[AssemblyTitle]

Cet attribut est utilisé pour donner un nom à cet assembly particulier.

```
[assembly: AssemblyTitle("MyProduct")]
```

[AssemblyProduct]

Cet attribut est utilisé pour décrire le produit auquel cet assemblage est destiné. Plusieurs assemblys peuvent être des composants du même produit, auquel cas ils peuvent tous partager la même valeur pour cet attribut.

```
[assembly: AssemblyProduct("MyProduct")]
```

Global et local AssemblyInfo

Avoir un global permet une meilleure DRY, il suffit de mettre des valeurs différentes dans AssemblyInfo.cs pour les projets qui ont des écarts. Cette utilisation suppose que votre produit dispose de plusieurs projets de studio visuel.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example

// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)

// Version information for an assembly consists of the following four values:
```

```
// roughly translated from I reckon it is for SO, note that they most likely
// dynamically generate this file
//      Major Version  - Year 6 being 2016
//      Minor Version  - The month
//      Day Number     - Day of month
//      Revision       - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - un pour chaque projet

```
//then the following might be put into a separate Assembly file per project, e.g.
[assembly: AssemblyTitle("Stackoveflow.Redis")]
```

Vous pouvez ajouter le GlobalAssemblyInfo.cs au projet local en utilisant la procédure suivante :

1. Sélectionnez Ajouter / Élément existant ... dans le menu contextuel du projet.
2. Sélectionnez GlobalAssemblyInfo.cs
3. Développez le bouton Ajouter en cliquant sur cette petite flèche en bas à droite
4. Sélectionnez "Ajouter en tant que lien" dans la liste déroulante des boutons.

[AssemblyVersion]

Cet attribut applique une version à l'assembly.

```
[assembly: AssemblyVersion("1.0.*")]
```

Le caractère `*` est utilisé pour incrémenter automatiquement une partie de la version à chaque fois que vous compilez (souvent utilisé pour le numéro "build")

Lecture des attributs d'assemblage

Grâce aux API de réflexion riches de .NET, vous pouvez accéder aux métadonnées d'un assemblage. Par exemple, vous pouvez obtenir l'attribut title de `this` assembly avec le code suivant

```
using System.Linq;
using System.Reflection;

...

Assembly assembly = typeof(this).Assembly;
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();

Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

Contrôle de version automatisé

Votre code dans le contrôle de code source a des numéros de version par défaut (identifiants SVN

ou hachages Git SHA1) ou explicitement (balises Git). Plutôt que de mettre à jour manuellement les versions dans AssemblyInfo.cs, vous pouvez utiliser un processus de génération pour écrire la version de votre système de contrôle de code source dans vos fichiers AssemblyInfo.cs et ainsi sur vos assemblys.

Les packages GitVersionTask ou SemVer.Git.Fody NuGet en sont des exemples. Pour utiliser GitVersionTask, par exemple, après avoir installé le package dans votre projet, supprimez les attributs `Assembly*Version` de vos fichiers AssemblyInfo.cs. Cela met GitVersionTask en charge de la version de vos assemblys.

Notez que le contrôle de version sémantique est de plus en plus la norme *de facto* , ces méthodes recommandent donc d'utiliser des balises de contrôle de source qui suivent SemVer.

Champs communs

Il est recommandé de compléter les champs par défaut de AssemblyInfo. Les informations peuvent être récupérées par les installateurs et apparaîtront lors de l'utilisation de Programmes et fonctionnalités (Windows 10) pour désinstaller ou modifier un programme.

Le minimum devrait être:

- AssemblyTitle - généralement l'espace de noms, *c'est-à - dire* MyCompany.MySolution.MyProject
- AssemblyCompany - le nom complet des entités légales
- AssemblyProduct - le marketing peut avoir une vue ici
- AssemblyCopyright - conservez-le à jour car il a l'air délabré sinon

"AssemblyTitle" devient la "Description du fichier" lors de l'examen de l'onglet Détails de propriétés de la DLL.

[AssemblyConfiguration]

AssemblyConfiguration: l'attribut AssemblyConfiguration doit avoir la configuration utilisée pour générer l'assembly. Utilisez la compilation conditionnelle pour inclure correctement différentes configurations d'assemblage. Utilisez le bloc similaire à l'exemple ci-dessous. Ajoutez autant de configurations différentes que vous utilisez habituellement.

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

[InternalsVisibleTo]

Si vous souhaitez rendre `internal` classes ou fonctions `internal` d'un assembly accessibles à partir d'un autre assembly, vous déclarez ceci par `InternalsVisibleTo` et le nom de l'assembly autorisé à y accéder.

Dans cet exemple de code dans l'assembly `MyAssembly.UnitTests` est autorisé à appeler `internal` éléments `internal` de `MyAssembly` .

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

Ceci est particulièrement utile pour les tests unitaires afin d'éviter les déclarations `public` inutiles.

[AssemblyKeyFile]

Chaque fois que nous voulons que notre assemblage s'installe dans GAC, il faut absolument avoir un nom fort. Pour un assemblage fort, nous devons créer une clé publique. Pour générer le fichier `.snk` .

Pour créer un fichier de clés de nom fort

1. Invite de commandes des développeurs pour VS2015 (avec accès administrateur)
2. À l'invite de commandes, tapez cd C: \ Directory_Name et appuyez sur ENTRÉE.
3. À l'invite de commandes, tapez sn -k KeyFileName.snk, puis appuyez sur ENTRÉE.

une fois le keyFileName.snk créé dans le répertoire spécifié, donnez la référence à votre projet. `snk` à l'attribut `AssemblyKeyFileAttribute` le chemin d'accès au fichier `snk` pour générer la clé lorsque nous construisons notre bibliothèque de classes.

Propriétés -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

Cela va créer un assemblage de noms fort après la construction. Après avoir créé votre assembly de nom fort, vous pouvez l'installer dans GAC

Bonne codage :)

Lire AssemblyInfo.cs Exemples en ligne: https://riptutorial.com/fr/csharp/topic/4264/assemblyinfo-cs-exemples

Chapitre 11: Async / wait, Backgroundworker, Exemples de tâches et de threads

Remarques

Pour exécuter l'un de ces exemples, appelez-les simplement comme ça:

```
static void Main()
{
    new Program().ProcessDataAsync();
    Console.ReadLine();
}
```

Examples

ASP.NET Configure Await

Lorsque ASP.NET gère une demande, un thread est attribué à partir du pool de threads et un **contexte de demande** est créé. Le contexte de la demande contient des informations sur la requête en cours, accessibles via la propriété statique `HttpContext.Current` . Le contexte de demande pour la demande est ensuite affecté au thread qui gère la demande.

Un contexte de requête donné **ne peut être actif que sur un thread à la fois** .

Lorsque l'exécution est en `await` , le thread qui gère une demande est renvoyé au pool de threads pendant que la méthode asynchrone s'exécute et que le contexte de la demande est libre pour qu'un autre thread puisse l'utiliser.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

Lorsque la tâche se termine, le pool de threads assigne un autre thread pour continuer l'exécution de la demande. Le contexte de la demande est ensuite affecté à ce thread. Cela peut être ou ne pas être le fil d'origine.

Blocage

Lorsque le résultat d'un appel de méthode `async` est attendu, **des** blocages **synchrones** peuvent survenir. Par exemple, le code suivant provoquera un blocage lorsque `IndexSync()` est appelé:

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

En effet, par défaut, la tâche attendue, dans ce cas, `db.Products.ToListAsync()` capturera le contexte (dans le cas d'ASP.NET, le contexte de la demande) et tentera de l'utiliser une fois terminé.

Lorsque toute la pile d'appels est asynchrone, il n'y a pas de problème car, une fois que l' `await` est atteinte, le thread d'origine est libéré, libérant le contexte de la requête.

Lorsque nous bloquons de manière synchrone à l'aide de `Task.Result` ou `Task.Wait()` (ou d'autres méthodes de blocage), le thread d'origine est toujours actif et conserve le contexte de la requête. La méthode attendue fonctionne toujours de manière asynchrone et une fois que le rappel tente de s'exécuter, c'est-à-dire une fois la tâche attendue renvoyée, elle tente d'obtenir le contexte de la requête.

Par conséquent, le blocage se produit car, pendant que le thread de blocage avec le contexte de requête attend la fin de l'opération asynchrone, l'opération asynchrone tente d'obtenir le contexte de la requête pour terminer.

ConfigureAwait

Par défaut, les appels à une tâche attendue captureront le contexte actuel et tenteront de reprendre l'exécution sur le contexte une fois terminé.

En utilisant `ConfigureAwait(false)` cela peut être évité et les blocages peuvent être évités.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request
context
    return View(products);
}

public ActionResult IndexSync()
```

```
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Cela peut éviter les blocages quand il est nécessaire de bloquer le code asynchrone, mais cela entraîne la perte du contexte dans la suite (code après l'appel à attendre).

Dans ASP.NET, cela signifie que si votre code suit un appel pour `await someTask.ConfigureAwait(false);` tente d'accéder aux informations du contexte, par exemple `HttpContext.Current.User` alors les informations ont été perdues. Dans ce cas, le `HttpContext.Current` est nul. Par exemple:

```
public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}
```

Si `ConfigureAwait(true)` est utilisé (équivalent à ne pas avoir ConfigureAwait du tout) alors à la fois l' `user` et `user2` sont peuplées avec les mêmes données.

Pour cette raison, il est souvent recommandé d'utiliser `ConfigureAwait(false)` dans le code de la bibliothèque où le contexte n'est plus utilisé.

Async / attente

Vous trouverez ci-dessous un exemple simple d'utilisation de async / waiting pour effectuer des tâches fastidieuses en arrière-plan, tout en conservant la possibilité d'effectuer d'autres tâches qui n'ont pas besoin d'attendre les tâches fastidieuses.

Cependant, si vous avez besoin de travailler avec le résultat de la méthode chronophage ultérieurement, vous pouvez le faire en attendant l'exécution.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");
```

```
    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    // Wait for TimeintensiveMethod to complete and get its result
    int x = await task;
    Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

BackgroundWorker

Voir ci-dessous un exemple simple d'utilisation d'un objet `BackgroundWorker` pour effectuer des opérations chronophages dans un thread d'arrière-plan.

Tu dois:

1. Définissez une méthode de travail qui effectue le travail `DoWork` et appelez-la à partir d'un gestionnaire d'événements pour l'événement `DoWork` d'un `BackgroundWorker` .
2. Commencez l'exécution avec `RunWorkerAsync` . Tout argument requis par la méthode de travailleur attaché à `DoWork` peut être transmis via le `DoWorkEventArgs` paramètre à `RunWorkerAsync` .

Outre l'événement `DoWork` , la classe `BackgroundWorker` définit également deux événements à utiliser pour interagir avec l'interface utilisateur. Ce sont facultatifs.

- L'événement `RunWorkerCompleted` est déclenché lorsque les gestionnaires `DoWork` sont terminés.
- L'événement `ProgressChanged` est déclenché lorsque la méthode `ReportProgress` est appelée.

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
```

```
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}

// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

       for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Tâche

Voir ci-dessous un exemple simple d'utilisation d'une `Task` pour effectuer des `Task` fastidieuses en arrière-plan.

Tout ce que vous avez à faire est d’emballer votre méthode intensive en temps dans un appel `Task.Run()` .

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
```

```
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Fil

Voir ci-dessous un exemple simple d'utilisation d'un `Thread` pour effectuer des tâches fastidieuses en arrière-plan.

```
public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

Comme vous pouvez le constater, nous ne pouvons pas renvoyer une valeur de notre `TimeIntensiveMethod` car `Thread` attend une méthode void comme paramètre.

Pour obtenir une valeur de retour d'un `Thread` utilisez un événement ou le suivant:

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();
```

```
        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.Writeline("Count: " + ret);
```

Tâche "exécuter et oublier" extension

Dans certains cas (par exemple, la journalisation), il peut être utile d'exécuter une tâche et ne pas attendre le résultat. L'extension suivante permet d'exécuter la tâche et de continuer l'exécution du code de repos:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

Le résultat est attendu uniquement dans la méthode d'extension. Comme `async` / `await` est utilisé, il est possible d'attraper une exception et d'appeler une méthode facultative pour la gérer.

Un exemple d'utilisation de l'extension:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
    {
        // Something went wrong, handle it.
    });
```

Lire Async / wait, Backgroundworker, Exemples de tâches et de threads en ligne: https://riptutorial.com/fr/csharp/topic/3824/async---wait--backgroundworker--exemples-de-taches-et-de-threads

Chapitre 12: Async-Await

Introduction

En C #, une méthode déclarée `async` ne sera pas bloquée dans un processus synchrone, dans le cas où vous utiliseriez des opérations basées sur les E / S (par exemple, accès Web, utilisation de fichiers, ...). Le résultat de telles méthodes marquées par async peut être attendu via l'utilisation du mot-clé `await` .

Remarques

Une méthode `async` peut renvoyer `void` , `Task` ou `Task<T>` .

Le type de retour `Task` attendra la fin de la méthode et le résultat sera `void` . `Task<T>` renverra une valeur du type `T` fois la méthode terminée.

`async` méthodes `async` doivent renvoyer `Task` ou `Task<T>` , par opposition à `void` , dans presque toutes les circonstances. `async void` méthodes `async void` ne peuvent pas être `await` , ce qui entraîne divers problèmes. Le seul scénario où un `async` doit retourner `void` est dans le cas d'un gestionnaire d'événements.

`async` / `await` fonctionne en transformant votre méthode `async` en une machine à états. Pour ce faire, il crée une structure en arrière-plan qui stocke l'état actuel et tout contexte (comme les variables locales) et expose une méthode `MoveNext()` pour avancer les états (et exécuter tout code associé) chaque fois qu'une attente attendue se termine.

Examples

Appels consécutifs simples

```
public async Task<JobResult> GetDataFromWebAsync()
{
  var nextJob = await _database.GetNextJobAsync();
  var response = await _httpClient.GetAsync(nextJob.Uri);
  var pageContents = await response.Content.ReadAsStringAsync();
  return await _database.SaveJobResultAsync(pageContents);
}
```

La principale chose à noter ici est que si tous les `await` méthode -ed est appelé de manière asynchrone - et pour le moment de cet appel le contrôle est donné au système - l'écoulement à l' intérieur de la méthode est linéaire et ne nécessite aucun traitement spécial en raison de asynchronie. Si l'une des méthodes appelées échoue, l'exception sera traitée "comme prévu", ce qui dans ce cas signifie que l'exécution de la méthode sera abandonnée et que l'exception passera à la pile.

Try / Catch / Finalement

6,0

A partir de C # 6.0, le mot-clé `await` peut maintenant être utilisé dans un bloc `catch` et `finally` .

```
try {
   var client = new AsyncClient();
   await client.DoSomething();
} catch (MyException ex) {
   await client.LogExceptionAsync();
   throw;
} finally {
   await client.CloseAsync();
}
```

5,0 6,0

Avant C # 6.0, vous deviez faire quelque chose comme suit. Notez que 6.0 a également nettoyé les vérifications NULL avec l' opérateur Null Propagating .

```
AsynClient client;
MyException caughtException;
try {
     client = new AsyncClient();
     await client.DoSomething();
} catch (MyException ex) {
     caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
       await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Notez que si vous attendez une tâche non créée par `async` (par exemple, une tâche créée par `Task.Run`), certains débogueurs peuvent casser des exceptions lancées par la tâche, même si elle est apparemment gérée par la méthode try / catch environnante. Cela se produit car le débogueur considère qu'il n'est pas géré par rapport au code utilisateur. Dans Visual Studio, il existe une option appelée "Just My Code" , qui peut être désactivée pour empêcher le débogueur de se briser dans de telles situations.

Configuration de Web.config sur la cible 4.5 pour un comportement asynchrone correct.

Web.config system.web.httpRuntime doit cibler la version 4.5 pour garantir que le thread loue le contexte de la requête avant de reprendre votre méthode asynchrone.

```
<httpRuntime targetFramework="4.5" />
```

Async et wait ont un comportement indéfini sur ASP.NET avant 4.5. L'async / wait reprendra sur un thread arbitraire qui pourrait ne pas avoir le contexte de la requête. Les applications sous

charge échoueront aléatoirement avec des exceptions de référence NULL accédant au HttpContext après l'attente. Utiliser HttpContext.Current dans WebApi est dangereux à cause de l'async

Appels simultanés

Il est possible d'attendre plusieurs appels en même temps qu'en invoquant d' abord les tâches awaitable et qui les attend.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

Vous pouvez également utiliser `Task.WhenAll` pour regrouper plusieurs tâches en une seule `Task` , qui se termine lorsque toutes les tâches réussies sont terminées.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

Vous pouvez également faire cela dans une boucle, par exemple:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

Pour obtenir les résultats d'une tâche après avoir attendu plusieurs tâches avec Task.WhenAll, attendez simplement la tâche à nouveau. Comme la tâche est déjà terminée, le résultat sera renvoyé

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();

await Task.WhenAll(task1, task2);

var result = await task2;
```

En outre, `Task.WhenAny` peut être utilisé pour exécuter plusieurs tâches en parallèle, telles que `Task.WhenAll` ci-dessus, à la différence que cette méthode se terminera lorsque l' *une* des tâches

fournies sera terminée.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

La `Task` renvoyée par `RunConcurrentTasksWhenAny` se terminera à la fin de `firstTask` , `secondTask` ou `thirdTask` .

Attendez l'opérateur et le mot-clé asynchrone

`await` opérateur et le mot-clé `async` réunis:

> La méthode asynchrone dans laquelle **wait** est utilisée doit être modifiée par le mot-clé **async** .

Le contraire n'est pas toujours vrai: vous pouvez marquer une méthode comme `async` sans utiliser `await` dans son corps.

Ce qui `await` réalité, c'est de suspendre l'exécution du code jusqu'à la fin de la tâche attendue; toute tâche peut être attendue.

Remarque: vous ne pouvez pas attendre la méthode asynchrone qui ne retourne rien (void).

En fait, le mot «suspend» est un peu trompeur car non seulement l'exécution s'arrête, mais le thread peut devenir libre pour exécuter d'autres opérations. Sous le capot, `await` est implémenté par un peu de magie du compilateur: il divise une méthode en deux parties - avant et après l' `await` . La dernière partie est exécutée à la fin de la tâche attendue.

Si nous ignorons certains détails importants, le compilateur le fait grosso modo pour vous:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
    var awaitedResult = await someTask;
    // ... do something more and produce result of type TResult
    return result;
}
```

devient:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
```

```
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
{
    // ...
}
```

Toute méthode habituelle peut être transformée en asynchrone de la manière suivante:

```
await Task.Run(() => YourSyncMethod());
```

Cela peut être avantageux lorsque vous devez exécuter une méthode longue sur le thread d'interface utilisateur sans bloquer l'interface utilisateur.

Mais il y a une remarque très importante ici: **Asynchrone ne signifie pas toujours concurrente (parallèle ou même multithread).** Même sur un seul thread, `async` - `await` permet toujours le code asynchrone. Par exemple, consultez ce planificateur de tâches personnalisé. Un tel planificateur de tâches «fou» peut simplement transformer des tâches en fonctions appelées dans le traitement de la boucle de messages.

Nous devons nous demander: quel thread exécutera la suite de notre méthode `DoIt_Continuation` ?

Par défaut, l'opérateur d' `await` planifie l'exécution de la continuation avec le contexte de synchronisation actuel . Cela signifie que par défaut pour WinForms et la continuation WPF s'exécute dans le thread d'interface utilisateur. Si, pour une raison quelconque, vous devez modifier ce comportement, utilisez la méthode `Task.ConfigureAwait()` :

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

Retourner une tâche sans attendre

Les méthodes qui effectuent des opérations asynchrones ne ont pas besoin d'utiliser `await` si:

- Il n'y a qu'un seul appel asynchrone dans la méthode
- L'appel asynchrone est à la fin de la méthode
- Une exception de capture / traitement pouvant survenir dans la tâche n'est pas nécessaire

Considérez cette méthode qui renvoie une `Task` :

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return await dataStore.GetByKeyAsync(lookupKey);
}
```

Si `GetByKeyAsync` a la même signature que `GetUserAsync` (en retournant une `Task<User>`), la méthode peut être simplifiée:

```
public Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return dataStore.GetByKeyAsync(lookupKey);
}
```

Dans ce cas, la méthode n'a pas besoin d'être marquée `async` , même si elle effectue une opération asynchrone. Le Groupe retourné par `GetByKeyAsync` est transmis directement à la méthode d'appel, où il sera `await` ed.

Important : Le fait de renvoyer la `Task` au lieu de l'attendre modifie le comportement d'exception de la méthode, car elle ne déclenche pas l'exception dans la méthode qui lance la tâche mais dans la méthode qui l'attend.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

Cela améliorera les performances car cela permettra au compilateur d'économiser la génération d'une machine à états **asynchrone** supplémentaire.

Le blocage sur du code asynchrone peut provoquer des blocages

Il est déconseillé de bloquer les appels asynchrones car cela peut provoquer des blocages dans des environnements dotés d'un contexte de synchronisation. La meilleure pratique consiste à utiliser Async / Wait "tout en bas". Par exemple, le code Windows Forms suivant provoque un blocage:

```
private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
```

```
    });

    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}
```

Essentiellement, une fois l'appel asynchrone terminé, il attend que le contexte de synchronisation devienne disponible. Cependant, le gestionnaire d'événement "conserve" le contexte de synchronisation pendant qu'il attend que la méthode `TryThis()` se termine, provoquant ainsi une attente circulaire.

Pour corriger cela, le code doit être modifié pour

```
private async void button1_Click(object sender, EventArgs e)
{
  bool result = await TryThis();
  Trace.TraceInformation("Done with result");
}
```

Remarque: les gestionnaires d'événements sont le seul endroit où `async void` doit être utilisé (car vous ne pouvez pas attendre une méthode `async void`).

Async / wait n'améliorera les performances que si elle permet à la machine d'effectuer des tâches supplémentaires

Considérez le code suivant:

```
public async Task MethodA()
{
     await MethodB();
     // Do other work
}

public async Task MethodB()
{
     await MethodC();
     // Do other work
}

public async Task MethodC()
{
     // Or await some other async work
     await Task.Delay(100);
}
```

Cela ne fonctionnera pas mieux que

```
public void MethodA()
{
     MethodB();
     // Do other work
}

public void MethodB()
{
     MethodC();
     // Do other work
}

public void MethodC()
{
     Thread.Sleep(100);
}
```

Le but principal de async / waiting est de permettre à la machine d'effectuer un travail supplémentaire, par exemple pour permettre au thread appelant d'effectuer d'autres tâches en attendant le résultat d'une opération d'E / S. Dans ce cas, le thread appelant n'est jamais autorisé à faire plus de travail que ce qu'il aurait pu faire autrement, il n'y a donc pas de gain de performance par rapport à l'appel `MethodA()` , `MethodB()` et `MethodC()` .

Lire Async-Await en ligne: https://riptutorial.com/fr/csharp/topic/48/async-await

Chapitre 13: BackgroundWorker

Syntaxe

- `bgWorker.CancellationPending //returns whether the bgWorker was cancelled during its operation`
- `bgWorker.IsBusy //returns true if the bgWorker is in the middle of an operation`
- `bgWorker.ReportProgress(int x) //Reports a change in progress. Raises the "ProgressChanged" event`
- `bgWorker.RunWorkerAsync() //Starts the BackgroundWorker by raising the "DoWork" event`
- `bgWorker.CancelAsync() //instructs the BackgroundWorker to stop after the completion of a task.`

Remarques

L'exécution d'opérations de longue durée au sein du thread d'interface utilisateur peut entraîner le blocage de votre application, apparaissant à l'utilisateur comme ayant cessé de fonctionner. Il est préférable que ces tâches soient exécutées sur un thread d'arrière-plan. Une fois terminée, l'interface utilisateur peut être mise à jour.

Apporter des modifications à l'interface utilisateur pendant l'opération BackgroundWorker nécessite d'appeler les modifications apportées au thread d'interface utilisateur, généralement à l'aide de la méthode Control.Invoke sur le contrôle que vous mettez à jour. Si vous négligez de le faire, votre programme lancera une exception.

Le BackgroundWorker est généralement utilisé uniquement dans les applications Windows Forms. Dans les applications WPF, les tâches permettent de décharger le travail sur les threads d'arrière-plan (éventuellement en combinaison avec async / wait). La mise à jour des mises à jour sur le thread d'interface utilisateur s'effectue généralement automatiquement lorsque la propriété mise à jour implémente INotifyPropertyChanged ou manuellement à l'aide du répartiteur du thread d'interface utilisateur.

Examples

Affectation de gestionnaires d'événements à un BackgroundWorker

Une fois que l'instance de BackgroundWorker a été déclarée, des propriétés et des gestionnaires d'événements doivent lui être attribués pour les tâches exécutées.

```
    /* This is the backgroundworker's "DoWork" event handler. This
       method is what will contain all the work you
       wish to have your program perform without blocking the UI. */

    bgWorker.DoWork += bgWorker_DoWork;
```

```
    /*This is how the DoWork event method signature looks like:*/
    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        // Work to be done here
        // ...
        // To get a reference to the current Backgroundworker:
        BackgroundWorker worker = sender as BackgroundWorker;
        // The reference to the BackgroundWorker is often used to report progress
        worker.ReportProgress(...);
    }

    /*This is the method that will be run once the BackgroundWorker has completed its tasks */

    bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

    /*This is how the RunWorkerCompletedEvent event method signature looks like:*/
    private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
    {
        // Things to be done after the backgroundworker has finished
    }

   /* When you wish to have something occur when a change in progress
     occurs, (like the completion of a specific task) the "ProgressChanged"
     event handler is used. Note that ProgressChanged events may be invoked
     by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress
     is set to true.  */

     bgWorker.ProgressChanged += bgWorker_ProgressChanged;

    /*This is how the ProgressChanged event method signature looks like:*/
    private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        // Things to be done when a progress change has been reported

        /* The ProgressChangedEventArgs gives access to a percentage,
         allowing for easy reporting of how far along a process is*/
        int progress = e.ProgressPercentage;
    }
```

Affectation de propriétés à un BackgroundWorker

Cela permet à BackgroundWorker d'être annulé entre les tâches

```
bgWorker.WorkerSupportsCancellation = true;
```

Cela permet au travailleur de signaler les progrès entre l'achèvement des tâches ...

```
bgWorker.WorkerReportsProgress = true;

//this must also be used in conjunction with the ProgressChanged event
```

Création d'une nouvelle instance BackgroundWorker

Un BackgroundWorker est généralement utilisé pour effectuer des tâches, parfois longues, sans bloquer le thread d'interface utilisateur.

```
// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
     public partial class ExampleForm : Form
     {

      // the following creates an instance of the BackgroundWorker named "bgWorker"
      BackgroundWorker bgWorker = new BackgroundWorker();

      public ExampleForm() { ...
```

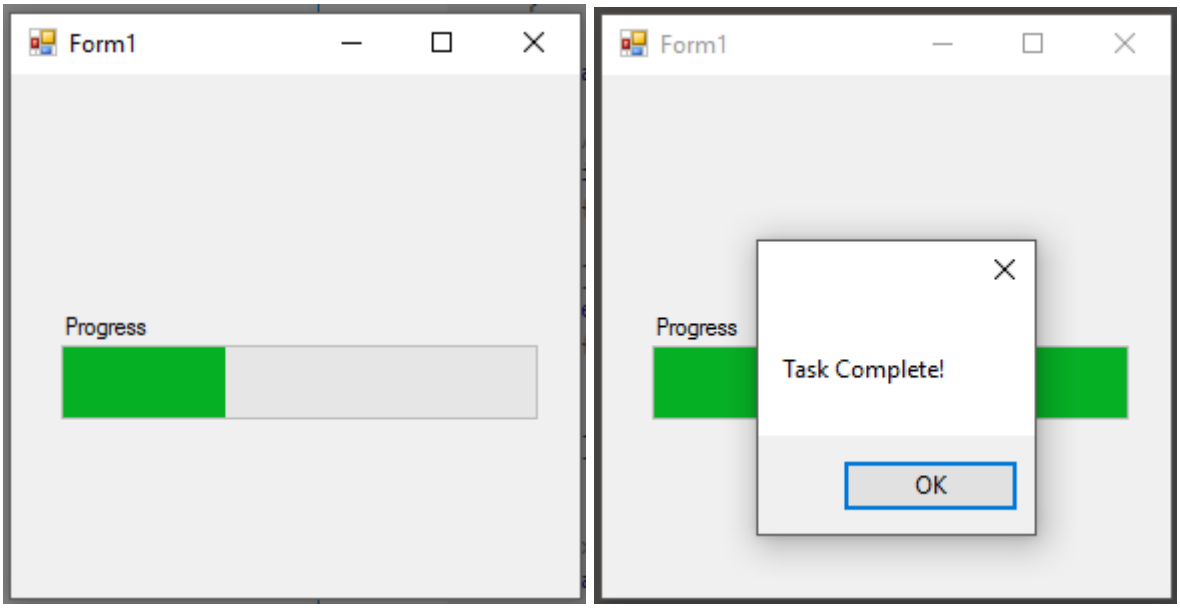
Utiliser un BackgroundWorker pour effectuer une tâche.

L'exemple suivant illustre l'utilisation d'un BackgroundWorker pour mettre à jour un WinForms ProgressBar. Le backgroundWorker mettra à jour la valeur de la barre de progression sans bloquer le thread d'interface utilisateur, affichant ainsi une interface utilisateur réactive pendant le travail en arrière-plan.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
{

    //a new instance of a backgroundWorker is created.
    BackgroundWorker bgWorker = new BackgroundWorker();

    public Form1()
    {
        InitializeComponent();

        prgProgressBar.Step = 1;

        //this assigns event handlers for the backgroundWorker
        bgWorker.DoWork += bgWorker_DoWork;
        bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

        //tell the backgroundWorker to raise the "DoWork" event, thus starting it.
        //Check to make sure the background worker is not already running.
        if(!bgWorker.IsBusy)
            bgWorker.RunWorkerAsync();

    }

    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        //this is the method that the backgroundworker will perform on in the background
thread.
        /* One thing to note! A try catch is not necessary as any exceptions will terminate
the backgroundWorker and report
          the error to the "RunWorkerCompleted" event */
        CountToY();
    }

    private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
    {
        //e.Error will contain any exceptions caught by the backgroundWorker
```

```
        if (e.Error != null)
        {
            MessageBox.Show(e.Error.Message);
        }
        else
        {
            MessageBox.Show("Task Complete!");
            prgProgressBar.Value = 0;
        }
    }

    // example method to perform a "long" running task.
    private void CountToY()
    {
        int x = 0;

        int maxProgress = 100;
        prgProgressBar.Maximum = maxProgress;


        while (x < maxProgress)
        {
            System.Threading.Thread.Sleep(50);
            Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
            x += 1;
        }
    }


}
```

Le résultat est le suivant ...



Lire BackgroundWorker en ligne: https://riptutorial.com/fr/csharp/topic/1588/backgroundworker

Chapitre 14: Bibliothèque parallèle de tâches

Examples

Parallel.ForEach

Un exemple qui utilise la boucle Parallel.ForEach pour exécuter un ping sur un tableau donné d'URL de site Web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

Parallel.For

Un exemple qui utilise Parallel.For en boucle pour exécuter un ping sur un tableau donné d'URL de sites Web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(urls[i]);
```

```
        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[i]));
        }
    });
}
```

Parallel.Invoke

Invoquer des méthodes ou des actions en parallèle (région parallèle)

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}
```

Une interrogation asynchrone annulable Tâche qui attend entre les itérations

```
public class Foo
{
    private const int TASK_ITERATION_DELAY_MS = 1000;
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask_EveryNSeconds, this._cts.Token);
    }
```

```
    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop that runs every N seconds. Good for checking for a heartbeat or
updates.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private async void OwnCodeCancelableTask_EveryNSeconds(object taskState)
    {
        var token = (CancellationToken)taskState;

        while (!token.IsCancellationRequested)
        {
            Console.WriteLine("Do the work that needs to happen every N seconds in this
loop");

            // Passing token here allows the Delay to be cancelled if your task gets
cancelled.
            await Task.Delay(TASK_ITERATION_DELAY_MS, token);
        }
    }
}
```

Une tâche d'interrogation annulable à l'aide de CancellationTokenSource

```
public class Foo
{
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask, this._cts.Token);
    }

    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop with no delays. Writing to a database while pulling from a buffer for
example.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private void OwnCodeCancelableTask(object taskState)
    {
        var token = (CancellationToken) taskState; //Our cancellation token passed from
StartNew();
```

```
        while ( !token.IsCancellationRequested )
        {
            Console.WriteLine("Do your task work in this loop");
        }
    }
}
```

Version asynchrone de PingUrl

```
    static void Main(string[] args)
    {
        string url = "www.stackoverflow.com";
        var pingTask = PingUrlAsync(url);
        Console.WriteLine($"Waiting for response from {url}");
        Task.WaitAll(pingTask);
        Console.WriteLine(pingTask.Result);
    }

    static async Task<string> PingUrlAsync(string url)
    {
        string response = string.Empty;
        var ping = new System.Net.NetworkInformation.Ping();

        var result = await ping.SendPingAsync(url);

        await Task.Delay(5000); //simulate slow internet

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            response = $"{url} is online";
        }

        return response;
    }
```

Lire Bibliothèque parallèle de tâches en ligne:
https://riptutorial.com/fr/csharp/topic/1010/bibliotheque-parallele-de-taches

Chapitre 15: BigInteger

Remarques

Quand utiliser

`BigInteger` objets `BigInteger` sont par nature très lourds en RAM. Par conséquent, ils ne devraient être utilisés qu'en cas d'absolue nécessité, c'est-à-dire pour des nombres à une échelle vraiment astronomique.

De plus, toutes les opérations arithmétiques sur ces objets ont un ordre de grandeur plus lent que leurs homologues primitifs. Il est donc possible pour un `BigInteger` de provoquer un crash en consommant toute la RAM disponible.

Des alternatives

Si la vitesse est impérative pour votre solution, il peut être plus efficace d'implémenter cette fonctionnalité vous-même en utilisant une classe enveloppant un `Byte[]` et en surchargeant vous-même les opérateurs nécessaires. Cependant, cela nécessite un effort supplémentaire considérable.

Examples

Calculez le premier nombre de Fibonacci à 1 000 chiffres

Incluez à l' `using System.Numerics` et ajoutez une référence à `System.Numerics` au projet.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

Cet algorithme simple parcourt les nombres de Fibonacci jusqu'à ce qu'il atteigne une longueur d'au moins 1000 chiffres décimaux, puis l'imprime. Cette valeur est nettement supérieure à ce que pourrait contenir un `ulong` .

Théoriquement, la seule limite à la classe `BigInteger` est la quantité de RAM que votre application peut consommer.

Remarque: `BigInteger` est uniquement disponible dans .NET 4.0 et versions ultérieures.

Lire BigInteger en ligne: https://riptutorial.com/fr/csharp/topic/5654/biginteger

Chapitre 16: BindingList

Examples

Eviter l'itération N * 2

Ceci est placé dans un gestionnaire d'événements Windows Forms

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
```

Cela prend beaucoup de temps pour exécuter, pour corriger, procédez comme suit:

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
nameList.RaiseListChangedEvents = false;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
nameList.RaiseListChangedEvents = true;
nameList.ResetBindings();
```

Ajouter un article à la liste

```
BindingList<string> listOfUIItems = new BindingList<string>();
listOfUIItems.Add("Alice");
listOfUIItems.Add("Bob");
```

Lire BindingList en ligne: https://riptutorial.com/fr/csharp/topic/182/bindinglist--t-

Chapitre 17: C # Script

Examples

Évaluation de code simple

Vous pouvez évaluer n'importe quel code C # valide:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) -
DateTime.Now");
```

Si type n'est pas spécifié, le résultat est `object` :

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Lire C # Script en ligne: https://riptutorial.com/fr/csharp/topic/3780/c-sharp-script

Chapitre 18: Chronomètres

Syntaxe

- stopWatch.Start () - Démarre le chronomètre.
- stopWatch.Stop () - Arrête le chronomètre.
- stopWatch.Elapsed - Obtient le temps total écoulé mesuré par l'intervalle en cours.

Remarques

Les chronomètres sont souvent utilisés dans les programmes de benchmarking au code temporel et permettent de voir comment les différents segments de code sont optimaux pour s'exécuter.

Examples

Créer une instance d'un chronomètre

Une instance Chronomètre peut mesurer le temps écoulé sur plusieurs intervalles, le temps total écoulé étant l'ensemble des intervalles individuels ajoutés. Cela donne une méthode fiable pour mesurer le temps écoulé entre deux événements ou plus.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
    d += 1;
}

stopWatch.Stop();
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

`Stopwach` est dans `System.Diagnostics` , vous devez donc ajouter à l' `using System.Diagnostics;` à votre fichier.

IsHighResolution

- La propriété IsHighResolution indique si le minuteur est basé sur un compteur de performances haute résolution ou basé sur la classe DateTime.
- Ce champ est en lecture seule.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("Operations timed using the system's high-resolution performance
counter.");
```

```
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine("  Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L*1000L*1000L) / frequency;
Console.WriteLine("  Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

https://dotnetfiddle.net/ckrWUo

La minuterie utilisée par la classe Chronomètre dépend du matériel et du système d'exploitation du système. IsHighResolution est vraie si le minuteur Chronomètre est basé sur un compteur de performance haute résolution. Sinon, IsHighResolution est false, ce qui indique que la minuterie du chronomètre est basée sur la minuterie du système.

Les ticks dans Stopwatch dépendent de la machine / du système d'exploitation, vous ne devriez donc jamais compter sur le nombre de ticks du chronomètre pour que les secondes soient identiques entre deux systèmes, voire sur le même système après un redémarrage. Ainsi, vous ne pouvez jamais compter sur les ticks de Chronomètre pour avoir le même intervalle que les ticks DateTime / TimeSpan.

Pour obtenir une heure indépendante du système, veillez à utiliser les propriétés ElapsedMilliseconds ou Stopwatch, qui prennent déjà en compte le nombre de secondes par seconde.

Chronomètre doit toujours être utilisé sur Datetime pour les processus de synchronisation car il est plus léger et utilise Dateime s'il ne peut pas utiliser un compteur de performance haute résolution.

La source

Lire Chronomètres en ligne: https://riptutorial.com/fr/csharp/topic/3676/chronometres

Chapitre 19: Classe partielle et méthodes

Introduction

Les classes partielles nous permettent de diviser les classes en plusieurs parties et en plusieurs fichiers sources. Toutes les pièces sont combinées en une seule classe pendant la compilation. Toutes les parties devraient contenir le mot-clé `partial` , devrait être de la même accessibilité. Toutes les pièces doivent être présentes dans le même assemblage pour pouvoir être incluses lors de la compilation.

Syntaxe

- classe **partielle** publique MyPartialClass {}

Remarques

- Les classes partielles doivent être définies dans le même assembly et dans le même espace de noms que la classe qu'elles étendent.
- Toutes les parties de la classe doivent utiliser le mot-clé `partial` .
- Toutes les parties de la classe doivent avoir la même accessibilité; `public` / `protected` / `private` etc.
- Si une partie utilise le mot-clé `abstract` , le type combiné est considéré comme abstrait.
- Si une partie utilise le mot-clé `sealed` , le type combiné est considéré comme scellé.
- Si une partie utilise le type de base, le type combiné hérite de ce type.
- Le type combiné hérite de toutes les interfaces définies sur toutes les classes partielles.

Examples

Classes partielles

Les classes partielles permettent de diviser la déclaration de classe (généralement en fichiers séparés). Un problème courant qui peut être résolu avec des classes partielles est de permettre aux utilisateurs de modifier le code généré automatiquement sans craindre que leurs modifications ne soient remplacées si le code est régénéré. De plus, plusieurs développeurs peuvent travailler sur la même classe ou les mêmes méthodes.

```
using System;

namespace PartialClassAndMethods
{
```

```
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second
declaration."
        }
    }
}
```

Méthodes partielles

La méthode partielle consiste en la définition dans une déclaration de classe partielle (en tant que scénario commun - dans celui généré automatiquement) et dans l'implémentation dans une autre déclaration de classe partielle.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.PartialMethod(); // outputs "Partial method called."
        }
    }
```

```
}
```

Classes partielles héritant d'une classe de base

Lors de l'héritage d'une classe de base, une seule classe partielle doit avoir la classe de base spécifiée.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

Vous *pouvez* spécifier la *même* classe de base dans plus d'une classe partielle. Il sera signalé comme redondant par certains outils IDE, mais compile correctement.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

Vous *ne pouvez pas* spécifier *différentes* classes de base dans plusieurs classes partielles, cela entraînera une erreur de compilation.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error

// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Lire Classe partielle et méthodes en ligne: https://riptutorial.com/fr/csharp/topic/3674/classe-partielle-et-methodes

Chapitre 20: Classes statiques

Examples

Mot-clé statique

Le mot-clé statique signifie 2 choses:

1. Cette valeur ne change pas d'un objet à un autre mais change plutôt pour une classe dans son ensemble
2. Les propriétés et les méthodes statiques ne nécessitent pas d'instance.

```
public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole
class

    }
}
```

Classes statiques

Le mot-clé "statique" faisant référence à une classe a trois effets:

1. Vous **ne pouvez pas** créer une instance d'une classe statique (cela supprime même le constructeur par défaut)

2. Toutes les propriétés et méthodes de la classe **doivent également** être statiques.
3. Une classe `static` est une classe `sealed` , ce qui signifie qu'elle ne peut pas être héritée.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var foo1 = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

Durée de vie de la classe statique

Une classe `static` est initialisée paresseusement sur l'accès des membres et reste active pendant toute la durée du domaine d'application.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first
accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
    {
        Console.WriteLine("Foo: Hi");
    }
}
```

```
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}
```

Lire Classes statiques en ligne: https://riptutorial.com/fr/csharp/topic/1653/classes-statiques

Chapitre 21: CLSCompliantAttribute

Syntaxe

1. [assembly: CLSCompliant (true)]
2. [CLSCompliant (true)]

Paramètres

Constructeur	Paramètre
CLSCompliantAttribute (booléen)	Initialise une instance de la classe CLSCompliantAttribute avec une valeur booléenne indiquant si l'élément de programme indiqué est conforme à CLS.

Remarques

La spécification CLS (Common Language Specification) est un ensemble de règles de base que tout langage ciblant l'interface CLI (langage qui confirme les spécifications Common Language Infrastructure) doit confirmer afin de pouvoir interagir avec d'autres langages compatibles CLS.

Liste des langues de la CLI

Dans la plupart des cas, vous devez marquer votre assembly en tant que CLSCompliant lorsque vous distribuez des bibliothèques. Cet attribut vous garantira que votre code sera utilisable par tous les langages compatibles CLS. Cela signifie que votre code peut être utilisé par n'importe quel langage pouvant être compilé et exécuté sur CLR (Common Language Runtime)

Lorsque votre assembly est marqué avec `CLSCompliantAttribute` , le compilateur vérifie si votre code enfreint l'une des règles CLS et renvoie un **avertissement** si nécessaire.

Examples

Modificateur d'accès auquel les règles CLS s'appliquent

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVacination = 0;
```

```
        //Warning CS3003  Type of 'Cat.DaysTillVacination' is not CLS-compliant
        protected UInt16 DaysTillVacination
        {
            get { return _daysTillVacination; }
        }

        //Warning    CS3003    Type of 'Cat.Age' is not CLS-compliant
        public UInt16 Age
        { get { return _age; } }

        //valid behaviour by CLS-compliant rules
        public int IncreaseAge()
        {
            int increasedAge = (int)_age + 1;

            return increasedAge;
        }

    }
}
```

Les règles de conformité CLS s'appliquent uniquement aux composants publics / protégés.

Violation de la règle CLS: types non signés / sbyte

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008  Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003  Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;

        //Warning    CS3003    Type of 'Car.YearOfCreation' is not CLS-compliant
        public UInt16 YearOfCreation
        {
            get { return _yearOfCreation; }
        }


        //Warning CS3002  Return type of 'Car.CalculateDistance()' is not CLS-compliant
        public UInt64 CalculateDistance()
        {
            return 0;
        }


        //Warning CS3002  Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-
compliant
        public UIntPtr TestDummyUnsignedPointerMethod()
        {
            int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            UIntPtr ptr = (UIntPtr)arr[0];
```

```
            return ptr;
        }

        //Warning CS3003  Type of 'Car.age' is not CLS-compliant
        public sbyte age = 120;


    }
}
```

Violation de la règle CLS: même dénomination

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning    CS3005    Identifier 'Car.CALCULATEAge()' differing only in case is not
CLS-compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }

    }
}
```

Visual Basic n'est pas sensible à la casse

Violation de la règle CLS: Identifiant _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3008  Identifier '_age' is not CLS-complian
        public int _age = 0;
    }

}
```

Vous ne pouvez pas démarrer la variable avec _

Violation de la règle CLS: hériter de la classe non CLSComplaint

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning    CS3009    'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }

}
```

Lire CLSCompliantAttribute en ligne:
https://riptutorial.com/fr/csharp/topic/7214/clscompliantattribute

Chapitre 22: Code non sécurisé dans .NET

Remarques

- Pour pouvoir utiliser le mot clé `unsafe` dans un projet .Net, vous devez cocher "Autoriser le code non sécurisé" dans Propriétés du projet => Créer
- L'utilisation d'un code non sécurisé peut améliorer les performances, mais au détriment de la sécurité du code (d'où le terme `unsafe`).

Par exemple, lorsque vous utilisez une boucle for, un tableau comme celui-ci:

```
for (int i = 0; i < array.Length; i++)
{
    array[i] = 0;
}
```

.NET Framework garantit que vous ne dépassez pas les limites du tableau, en lançant une `IndexOutOfRangeException` si l'index dépasse les limites.

Toutefois, si vous utilisez un code non sécurisé, vous pouvez dépasser les limites du tableau comme suit:

```
unsafe
{
    fixed (int* ptr = array)
    {
        for (int i = 0; i <= array.Length; i++)
        {
            *(ptr+i) = 0;
        }
    }
}
```

Examples

Index de tableau dangereux

```
void Main()
{
    unsafe
    {
        int[] a = {1, 2, 3};
        fixed(int* b = a)
        {
            Console.WriteLine(b[4]);
        }
    }
}
```

L'exécution de ce code crée un tableau de longueur 3, mais essaie ensuite d'obtenir le 5ème

élément (index 4). Sur ma machine, cette `1910457872` imprimée, mais le comportement n'est pas défini.

Sans le bloc `unsafe` , vous ne pouvez pas utiliser de pointeurs et par conséquent, vous ne pouvez pas accéder aux valeurs après la fin d'un tableau sans provoquer une exception.

Utiliser des tableaux non sécurisés

Lors de l'accès aux tableaux avec des pointeurs, il n'y a pas de vérification des limites et par conséquent, aucune `IndexOutOfRangeException` ne sera lancée. Cela rend le code plus rapide.

Affectation de valeurs à un tableau avec un pointeur:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

Alors que la contrepartie sûre et normale serait:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

La partie dangereuse sera généralement plus rapide et la différence de performance peut varier en fonction de la complexité des éléments de la matrice et de la logique appliquée à chacun. Même s'il peut être plus rapide, il doit être utilisé avec précaution, car il est plus difficile à entretenir et plus facile à briser.

Utilisation de unsafe avec des chaînes

```
var s = "Hello";      // The string referenced by variable 's' is normally immutable, but
                      // since it is memory, we could change it if we can access it in an
                      // unsafe way.

unsafe                // allows writing to memory; methods on System.String don't allow this
{
  fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';     // change data in memory allocated for original string "Hello"
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                      // value in memory, but the contents at that location were
                      // changed by the unsafe write above.
                      // Displays: "aaaaa"
```

Lire Code non sécurisé dans .NET en ligne: https://riptutorial.com/fr/csharp/topic/81/code-non-securise-dans--net

Chapitre 23: Comment utiliser les structures C # pour créer un type d'union (similaire aux unions C)

Remarques

Les types d'union sont utilisés dans plusieurs langues, notamment en langage C, pour contenir plusieurs types différents qui peuvent "se chevaucher" dans le même espace mémoire. En d'autres termes, ils peuvent contenir des champs différents qui démarrent tous au même décalage de mémoire, même s'ils peuvent avoir des longueurs et des types différents. Cela a l'avantage d'économiser de la mémoire et d'effectuer une conversion automatique.

Veuillez noter les commentaires dans le constructeur de la structure. L'ordre dans lequel les champs sont initialisés est extrêmement important. Vous souhaitez d'abord initialiser tous les autres champs, puis définir la valeur que vous souhaitez modifier en tant que dernière instruction. Comme les champs se chevauchent, la configuration de la dernière valeur est celle qui compte.

Examples

Unions de style C en C

Les types d'union sont utilisés dans plusieurs langues, comme le langage C, pour contenir plusieurs types différents qui peuvent "se chevaucher". En d'autres termes, ils peuvent contenir des champs différents qui démarrent tous au même décalage de mémoire, même s'ils peuvent avoir des longueurs et des types différents. Cela a l'avantage d'économiser de la mémoire et d'effectuer une conversion automatique. Pensez à une adresse IP, par exemple. En interne, une adresse IP est représentée sous forme d'entier, mais nous souhaitons parfois accéder au composant Byte différent, comme dans Byte1.Byte2.Byte3.Byte4. Cela fonctionne pour tous les types de valeur, que ce soit des primitives comme Int32 ou long, ou pour d'autres structures que vous définissez vous-même.

Nous pouvons obtenir le même effet en C # en utilisant des structures de disposition explicites.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
```

```
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
    }

    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Ayant défini Struct de cette manière, nous pouvons l'utiliser comme nous utiliserions une Union en C. Par exemple, créons une adresse IP en tant qu'entier aléatoire, puis modifions le premier jeton de l'adresse en «100», en le modifiant. de 'ABCD' à '100.BCD':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Sortie:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

Voir la démo

Les types d'union en C # peuvent également contenir des champs Struct

En dehors des primitives, les structures (Unions) de la structure explicite en C # peuvent également contenir d'autres structures. Tant qu'un champ est un type de valeur et non une référence, il peut être contenu dans une union:

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http, Https, Ftp, Sftp, Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
```

```
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port  = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}
```

Nous pouvons maintenant vérifier que l'ensemble de l'Union de services correspond à la taille d'un long (8 octets).

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;

Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

Voir la démo

Lire Comment utiliser les structures C # pour créer un type d'union (similaire aux unions C) en ligne: https://riptutorial.com/fr/csharp/topic/5626/comment-utiliser-les-structures-c-sharp-pour-creer-un-type-d-union--similaire-aux-unions-c-

Chapitre 24: Commentaires et régions

Examples

commentaires

Utiliser des commentaires dans vos projets est un moyen pratique de laisser des explications sur vos choix de conception et devrait viser à rendre votre vie (ou celle de quelqu'un d'autre) plus facile lors de la maintenance ou de l'ajout au code.

Il y a deux manières d'ajouter un commentaire à votre code.

Commentaires sur une seule ligne

Tout texte placé après `//` sera traité comme un commentaire.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

Multi lignes ou commentaires délimités

Tout texte entre `/*` et `*/` sera traité comme un commentaire.

```
public class Program
{
    public static void Main()
    {
        /*
            This is a multi line comment
            it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
        // although it's rarely used in practice
        System.Console.WriteLine(/* Inline comment */ "Hello, World!");

        System.Console.ReadLine();
    }
```

```
}
```

Les régions

Une région est un bloc de code réductible qui peut aider à la lisibilité et à l'organisation de votre code.

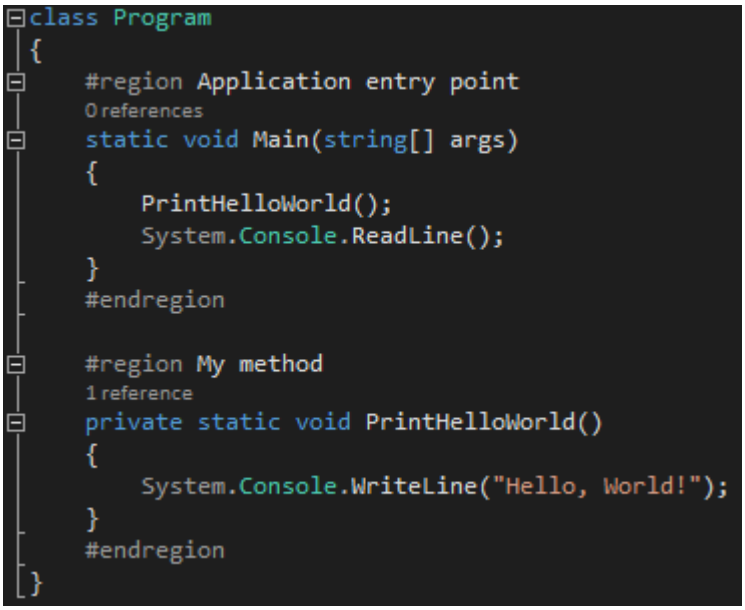
REMARQUE: la règle SA1124 DoNotUseRegions de StyleCop décourage l'utilisation de régions. Ils sont généralement le signe d'un code mal organisé, car C # inclut des classes partielles et d'autres fonctionnalités qui rendent les régions obsolètes.

Vous pouvez utiliser des régions de la manière suivante:

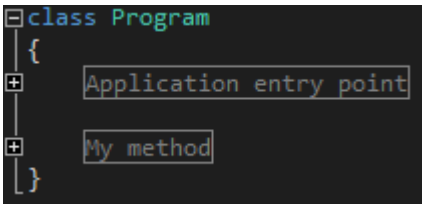
```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

Lorsque le code ci-dessus est affiché dans un IDE, vous pourrez réduire et développer le code à l'aide des symboles + et -.

Étendu



S'est effondré

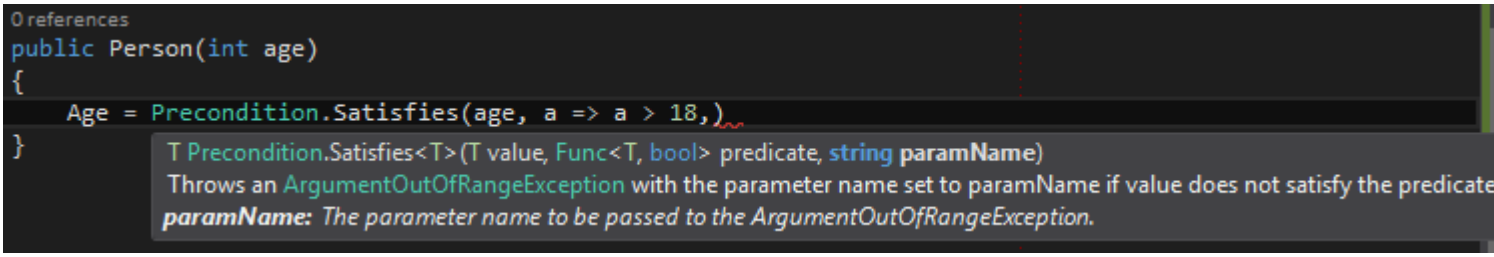


Commentaires sur la documentation

Les commentaires sur la documentation XML peuvent être utilisés pour fournir une documentation API qui peut être facilement traitée par des outils:

```
/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
    ///     name set to <c>paramName</c> if <c>value</c> does not satisfy the
    ///     <c>predicate</c> specified.
    /// </summary>
    /// <typeparam name="T">
    ///     The type of the argument checked
    /// </typeparam>
    /// <param name="value">
    ///     The argument to be checked
    /// </param>
    /// <param name="predicate">
    ///     The predicate the value is required to satisfy
    /// </param>
    /// <param name="paramName">
    ///     The parameter name to be passed to the
    ///     <see cref="ArgumentOutOfRangeException"/>.
    /// </param>
    /// <returns>The value specified</returns>
    public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
    {
        if (!predicate(value))
            throw new ArgumentOutOfRangeException(paramName);

        return value;
    }
}
```

La documentation est instantanément captée par IntelliSense:



Lire Commentaires et régions en ligne: https://riptutorial.com/fr/csharp/topic/5346/commentaires-et-regions

Chapitre 25: Commentaires sur la documentation XML

Remarques

Quelques fois, vous devez **créer une documentation texte étendue à** partir de vos commentaires XML. Malheureusement, ***il n'y a pas de moyen standard pour cela*** .

Mais il existe des projets distincts que vous pouvez utiliser pour ce cas:

- Château de sable
- Docu
- NDoc
- DocFX

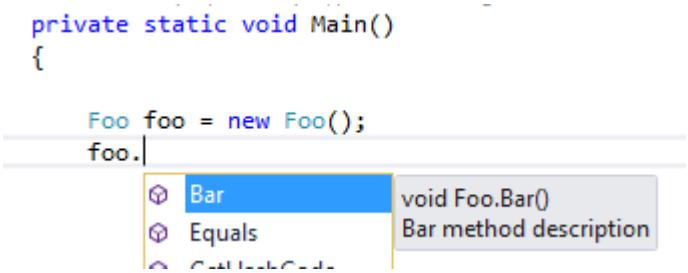
Examples

Annotation de méthode simple

Les commentaires sur la documentation sont placés directement au-dessus de la méthode ou de la classe qu'ils décrivent. Ils commencent par trois barres obliques `///` et permettent de stocker les métadonnées via XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{

}
```

Les informations contenues dans les balises peuvent être utilisées par Visual Studio et d'autres outils pour fournir des services tels que IntelliSense:



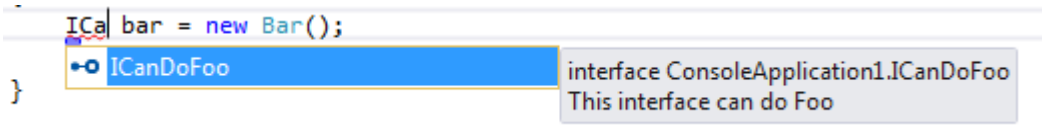
Voir aussi la liste de Microsoft des balises de documentation communes .

Commentaires sur la documentation de l'interface et de la classe

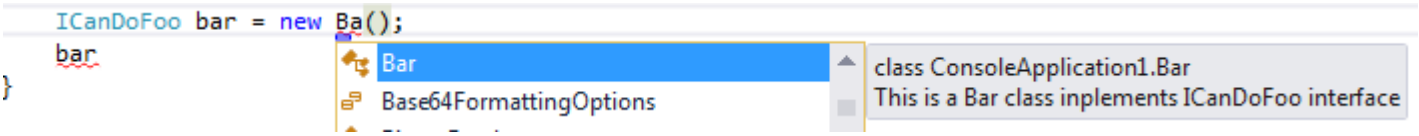
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

Résultat

Résumé de l'interface



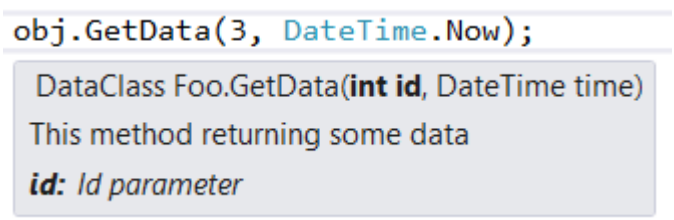
Résumé de classe



Commentaire sur la documentation de la méthode avec les éléments param et return

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
   // ...
}
```

IntelliSense vous montre la description de chaque paramètre:

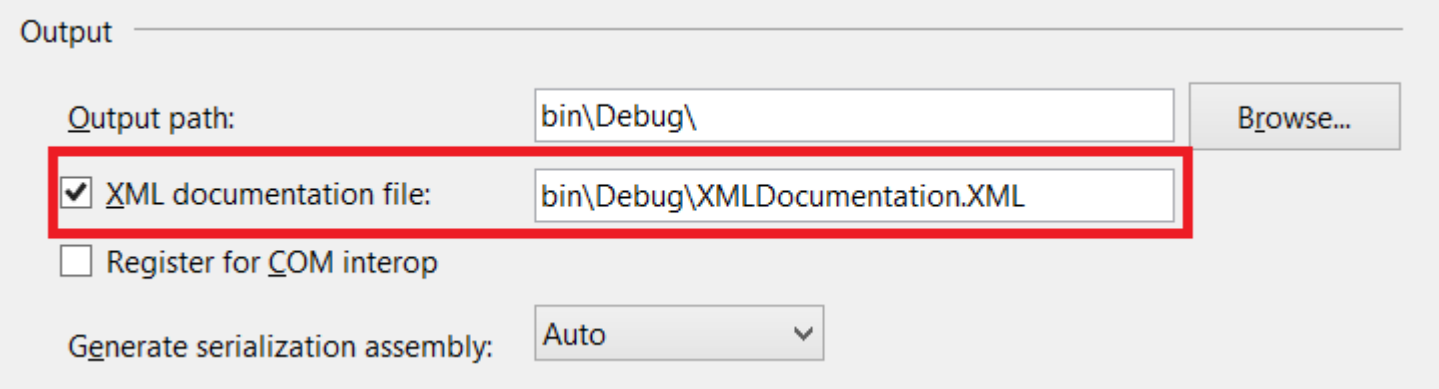


Conseil: Si Intellisense ne s'affiche pas dans Visual Studio, supprimez le premier crochet ou la virgule, puis tapez-le à nouveau.

Générer du code XML à partir de commentaires de documentation

Pour générer un fichier de documentation XML à partir des commentaires de documentation dans le code, utilisez l'option `/doc` avec le compilateur `csc.exe` C #.

Dans Visual Studio 2013/2015, dans **Projet** -> **Propriétés** -> **Construire** -> **Sortie** , cochez la case `XML documentation file` :



Lorsque vous construisez le projet, un fichier XML sera généré par le compilateur avec un nom correspondant au nom du projet (par exemple `XMLDocumentation.dll` -> `XMLDocumentation.xml`).

Lorsque vous utilisez l'assembly dans un autre projet, assurez-vous que le fichier XML se trouve dans le même répertoire que la DLL référencée.

Cet exemple:

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}


/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
```

```
    {
        return new DataClass();
    }
}
```

Produit ce xml sur build:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>
```

Référencement d'une autre classe dans la documentation

La `<see>` peut être utilisée pour créer un lien vers une autre classe. Il contient le membre `cref` qui doit contenir le nom de la classe à référencer. Visual Studio fournira Intellsense lors de l'écriture de cette balise et ces références seront traitées lors du renommage de la classe référencée.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

Dans les fenêtres contextuelles de Visual Studio Intellisense, ces références seront également affichées en couleur dans le texte.

Pour référencer une classe générique, utilisez quelque chose de similaire à ce qui suit:

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
/// </summary>
public class SomeGenericClass<T>
{
}
```

Lire Commentaires sur la documentation XML en ligne:
https://riptutorial.com/fr/csharp/topic/740/commentaires-sur-la-documentation-xml

Chapitre 26: Concaténation de cordes

Remarques

Si vous créez une chaîne dynamique, il est `Concat` d'opter pour la classe `StringBuilder` plutôt que de joindre des chaînes à l'aide de la méthode + ou `Concat` , car chaque + / `Concat` crée un nouvel objet chaîne à chaque exécution.

Examples

+ Opérateur

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Concaténer des chaînes à l'aide de System.Text.StringBuilder

La concaténation de chaînes à l'aide de StringBuilder peut offrir des avantages en termes de performances par rapport à la simple concaténation de chaînes à l'aide de `+` . Cela est dû à la manière dont la mémoire est allouée. Les chaînes sont réallouées à chaque concaténation, StringBuilders alloue de la mémoire en blocs uniquement en les réaffectant lorsque le bloc actuel est épuisé. Cela peut faire une énorme différence lors de nombreuses petites concaténations.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Les appels à `Append()` peuvent être chaînés, car ils renvoient une référence à `StringBuilder` :

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

Concat éléments de tableau de chaînes à l'aide de String.Join

La méthode `String.Join` peut être utilisée pour concaténer plusieurs éléments à partir d'un tableau de chaînes.

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
```

```
string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

Produit la sortie suivante: "orange, grape"

Cet exemple utilise la `String.Join(String, String[], Int32, Int32)` , qui spécifie l'index de démarrage et le décompte au-dessus du séparateur et de la valeur.

Si vous ne souhaitez pas utiliser les surcharges startIndex et count, vous pouvez joindre toutes les chaînes données. Comme ça:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

qui produira;

pomme, orange, raisin, poire

Concaténation de deux chaînes en utilisant $

$ fournit une méthode simple et concise pour concaténer plusieurs chaînes.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

Lire Concaténation de cordes en ligne: https://riptutorial.com/fr/csharp/topic/3616/concatenation-de-cordes

Chapitre 27: Constructeurs et finaliseurs

Introduction

Les constructeurs sont des méthodes d'une classe appelées lorsqu'une instance de cette classe est créée. Leur principale responsabilité est de laisser le nouvel objet dans un état utile et cohérent.

Les destructeurs / finaliseurs sont des méthodes d'une classe appelées lorsqu'une instance de celle-ci est détruite. En C #, ils sont rarement explicitement écrits / utilisés.

Remarques

C # n'a pas réellement de destructeurs, mais plutôt de finaliseurs qui utilisent la syntaxe de destructeur de style C ++. La spécification d'un destructeur remplace la méthode `Object.Finalize()` qui ne peut pas être appelée directement.

Contrairement à d'autres langages ayant une syntaxe similaire, ces méthodes *ne* sont *pas* appelées lorsque des objets sont hors de portée, mais sont appelées lorsque le ramasse-miettes s'exécute, ce qui se produit sous certaines conditions . En tant que tels, ils *ne* sont *pas* garantis pour fonctionner dans un ordre particulier.

Les finaliseurs doivent être responsables du nettoyage des ressources non gérées **uniquement** (pointeurs acquis via la classe Marshal, reçus via p / Invoke (appels système) ou pointeurs bruts utilisés dans des blocs non sécurisés). Pour nettoyer les ressources gérées, consultez IDisposable, le modèle Dispose et l'instruction `using` .

(Autres lectures: Quand devrais-je créer un destructeur?)

Examples

Constructeur par défaut

Lorsqu'un type est défini sans constructeur:

```
public class Animal
{
}
```

alors le compilateur génère un constructeur par défaut équivalent à ce qui suit:

```
public class Animal
{
    public Animal() {}
}
```

La définition de tout constructeur pour le type supprimera la génération de constructeur par défaut. Si le type a été défini comme suit:

```
public class Animal
{
    public Animal(string name) {}
}
```

alors un `Animal` ne peut être créé qu'en appelant le constructeur déclaré.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

Pour le deuxième exemple, le compilateur affichera un message d'erreur:

> 'Animal' ne contient pas de constructeur qui prend 0 arguments

Si vous voulez qu'une classe possède à la fois un constructeur sans paramètre et un constructeur qui prend un paramètre, vous pouvez le faire en implémentant explicitement les deux constructeurs.

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

Le compilateur ne sera pas en mesure de générer un constructeur par défaut si la classe étend une autre classe qui ne possède pas de constructeur sans paramètre. Par exemple, si nous avions une classe `Creature` :

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

alors `Animal` défini comme `class Animal : Creature {}` ne compile pas.

Appeler un constructeur d'un autre constructeur

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
```

```
    {
        Name = name;
    }
}

var dog = new Animal();      // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

Constructeur statique

Un constructeur statique est appelé la première fois qu'un membre d'un type est initialisé, un membre de classe statique est appelé ou une méthode statique. Le constructeur statique est thread-safe. Un constructeur statique est couramment utilisé pour:

- Initialiser l'état statique, c'est-à-dire l'état qui est partagé entre différentes instances de la même classe.
- Créer un singleton

Exemple:

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

Sortie:

Animal initialisé
Animal créé
Animal créé

Voir la démo

Si le premier appel est à une méthode statique, le constructeur statique est appelé sans le

constructeur d'instance. Ceci est correct, car la méthode statique ne peut pas accéder à l'état de l'instance de toute façon.

```
Animal.Yawn();
```

Cela va sortir:

> Animal initialisé
> Bâillement!

Voir aussi Exceptions dans les constructeurs statiques et les constructeurs statiques génériques .

Exemple singleton:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Appel du constructeur de la classe de base

Un constructeur d'une classe de base est appelé avant qu'un constructeur d'une classe dérivée ne soit exécuté. Par exemple, si `Mammal` étend `Animal` , le code contenu dans le constructeur de `Animal` est appelé en premier lors de la création d'une instance de `Mammal` .

Si une classe dérivée ne spécifie pas explicitement quel constructeur de la classe de base doit être appelé, le compilateur assume le constructeur sans paramètre.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Dans ce cas, l'instanciation d'un `Mammal` en appelant un `new Mammal("George the Cat")` imprimera

> Un animal inconnu naît.
> George le chat est un mammifère.

Voir la démo

L'appel d'un constructeur différent de la classe de base se fait en plaçant `: base(args)` entre la signature du constructeur et son corps:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Appeler un `new Mammal("George the Cat")` va maintenant imprimer:

> George le chat naît.
> George le chat est un mammifère.

Voir la démo

Finalisateurs sur les classes dérivées

Lorsqu'un graphe d'objet est finalisé, l'ordre est l'inverse de la construction. Par exemple, le super-type est finalisé avant le type de base, comme le montre le code suivant:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

Modèle de constructeur singleton

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();
```

```
    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

Le constructeur étant privé, aucune nouvelle instance de `SingletonClass` ne peut être créée en consommant du code. Le seul moyen d'accéder à l'instance unique de `SingletonClass` consiste à utiliser la propriété statique `SingletonClass.Instance` .

La propriété `Instance` est affectée par un constructeur statique généré par le compilateur C #. Le runtime .NET garantit que le constructeur statique est exécuté au plus une fois et est exécuté avant la première lecture de l' `Instance` . Par conséquent, toutes les préoccupations relatives à la synchronisation et à l'initialisation sont exécutées par le moteur d'exécution.

Notez que si le constructeur statique échoue, la classe `Singleton` devient définitivement inutilisable pour la vie de AppDomain.

De même, l'exécution du constructeur statique n'est pas garantie au moment du premier accès à `Instance` . Au contraire, il fonctionnera *à un moment donné avant cela* . Cela rend le temps auquel l'initialisation se produit non déterministe. Dans des cas pratiques, JIT appelle souvent le constructeur statique lors de la *compilation* (et non de l'exécution) d'une `Instance` référençant une méthode. Ceci est une optimisation des performances.

Reportez-vous à la page Implémentations Singleton pour savoir comment implémenter le modèle singleton.

Forcer un constructeur statique à être appelé

Alors que les constructeurs statiques sont toujours appelés avant la première utilisation d'un type, il est parfois utile de pouvoir les forcer à être appelés et la classe `RuntimeHelpers` fournit une aide:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

Remarque **:** Toutes les initialisations statiques (les initialiseurs de champs par exemple) seront exécutées, pas seulement le constructeur lui-même.

Utilisations ***potentielles*** **:** forcer l'initialisation pendant l'écran de démarrage dans une application d'interface utilisateur ou s'assurer qu'un constructeur statique n'échoue pas dans un test unitaire.

Appeler des méthodes virtuelles dans un constructeur

Contrairement à C ++ en C #, vous pouvez appeler une méthode virtuelle à partir du constructeur de classe (OK, vous pouvez aussi en C ++, mais le comportement est surprenant au premier abord). Par exemple:

```
abstract class Base
{
    protected Base()
    {
        _obj = CreateAnother();
    }

    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

Si vous venez d'un arrière-plan C ++, c'est surprenant, le constructeur de la classe de base voit déjà la table des méthodes virtuelles de la classe dérivée!

Attention : la classe dérivée n'est peut-être pas encore complètement initialisée (son constructeur sera exécuté après le constructeur de la classe de base) et cette technique est dangereuse (il y a aussi un avertissement StyleCop pour cela). Habituellement, cela est considéré comme une mauvaise pratique.

Constructeurs statiques génériques

Si le type sur lequel le constructeur statique est déclaré est générique, le constructeur statique sera appelé une fois pour chaque combinaison unique d'arguments génériques.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

Cela va sortir:

System.Object
System.String

Voir aussi Comment fonctionnent les constructeurs statiques pour les types génériques?

Exceptions dans les constructeurs statiques

Si un constructeur statique lève une exception, il ne sera jamais réessayé. Le type est inutilisable pour la durée de vie de AppDomain. Toute autre utilisation du type `TypeInitializationException` une `TypeInitializationException` autour de l'exception d'origine.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() {}
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

Cela va sortir:

> Stator statique
>
> System.TypeInitializationException: L'initialiseur de type pour 'Animal' a généré une exception. ---> System.Exception: Une exception de type 'System.Exception' a été levée.

[...]

> System.TypeInitializationException: L'initialiseur de type pour 'Animal' a généré une exception. ---> System.Exception: Une exception de type 'System.Exception' a été levée.

où vous pouvez voir que le constructeur actuel n'est exécuté qu'une seule fois et que l'exception

est réutilisée.

Initialisation du constructeur et de la propriété

L'affectation de la valeur de la propriété doit-elle être exécutée *avant* ou *après* le constructeur de la classe?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

Dans l'exemple ci-dessus, la valeur `TestProperty` doit- `TestProperty` être `1` dans le constructeur de la classe ou après le constructeur de classe?

Affectez des valeurs de propriété dans la création d'instance comme ceci:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Sera exécuté ***après*** l'exécution du constructeur. Toutefois, l'initialisation de la propriété dans la propriété de la classe dans C # 6.0 ressemble à ceci:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

sera fait ***avant que*** le constructeur ne soit exécuté.

Combinant les deux concepts ci-dessus dans un seul exemple:

```
public class TestClass
{
```

```
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

static void Main(string[] args)
{
    var testInstance = new TestClass() { TestProperty = 1 };
    Console.WriteLine(testInstance.TestProperty); //resulting in 1
}
```

Résultat final:

```
"Or shall this be executed"
"1"
```

Explication:

La valeur `TestProperty` sera d'abord attribuée en tant que `2` , puis le constructeur `TestClass` sera exécuté, entraînant l'impression de

```
"Or shall this be executed"
```

Et puis, la `TestProperty` sera affectée à `1` raison de la `new TestClass() { TestProperty = 1 }` , ce qui rend la valeur finale de `TestProperty` imprimée par `Console.WriteLine(testInstance.TestProperty)`

```
"1"
```

Lire Constructeurs et finaliseurs en ligne: https://riptutorial.com/fr/csharp/topic/25/constructeurs-et-finaliseurs

Chapitre 28: Constructions de flux de données TPL (Task Parallel Library)

Examples

JoinBlock

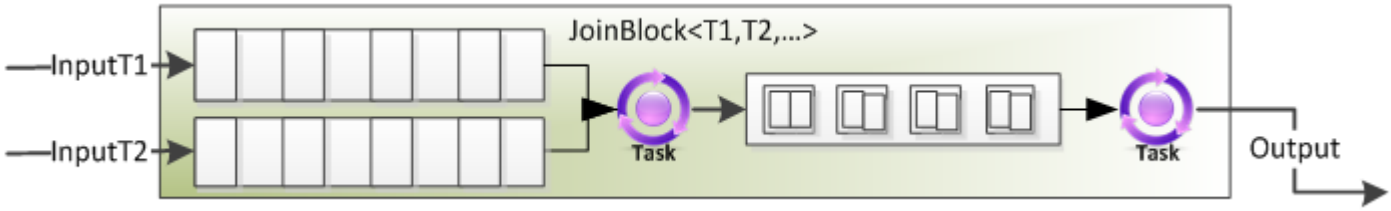
(Recueille 2-3 entrées et les combine dans un tuple)

Comme BatchBlock, JoinBlock <T1, T2,…> peut regrouper des données provenant de plusieurs sources de données. En fait, c'est l'objectif principal de JoinBlock <T1, T2,…>.

Par exemple, un JoinBlock <string, double, int> est un ISourceBlock <Tuple <string, double, int >>.

Comme avec BatchBlock, JoinBlock <T1, T2,…> est capable de fonctionner en mode gourmand et non gourmand.

- Dans le mode gourmand par défaut, toutes les données proposées aux cibles sont acceptées, même si l'autre cible ne dispose pas des données nécessaires pour former un tuple.
- En mode non gourmand, les cibles du bloc reporteront les données jusqu'à ce que toutes les cibles se soient vues proposer les données nécessaires à la création d'un tuple. À ce moment, le bloc s'engagera dans un protocole de validation en deux phases. Ce report permet à une autre entité de consommer les données entre-temps afin de permettre au système global d'avancer.



Traitement des demandes avec un nombre limité d'objets groupés

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);
```

```
    return resource;
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

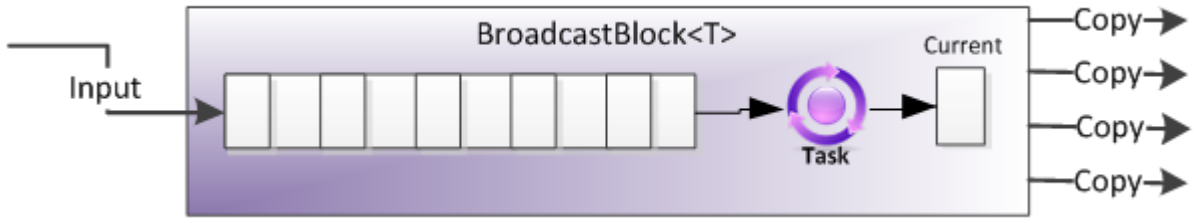
Introduction à TPL Dataflow par Stephen Toub

BroadcastBlock

(Copier un article et envoyer les copies à chaque bloc auquel il est lié)

Contrairement à BufferBlock, la mission de BroadcastBlock dans la vie est de permettre à toutes les cibles liées à partir du bloc d'obtenir une copie de chaque élément publié, en écrasant continuellement la valeur «actuelle» avec celles qui lui sont propagées.

En outre, contrairement à BufferBlock, BroadcastBlock ne conserve pas inutilement les données. Après qu'une donnée particulière a été offerte à toutes les cibles, cet élément sera écrasé par n'importe quelle donnée suivante (comme pour tous les blocs de flux de données, les messages sont traités dans l'ordre FIFO). Cet élément sera offert à toutes les cibles, et ainsi de suite.



Producteur / consommateur asynchrone avec un producteur étranglé

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

Statut d'exposition d'un agent

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }
```

```
    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        …
        Status.Post("Done");
    }
}
```

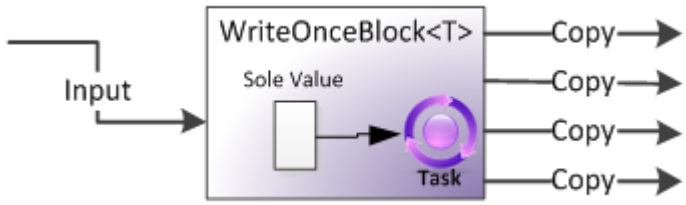
Introduction à TPL Dataflow par Stephen Toub

WriteOnceBlock

(Variable en lecture seule: Mémorise son premier élément de données et en distribue des copies en tant que sortie. Ignore tous les autres éléments de données)

Si BufferBlock est le bloc le plus fondamental dans TPL Dataflow, WriteOnceBlock est le plus simple.
Il stocke au plus une valeur et, une fois cette valeur définie, elle ne sera jamais remplacée ou remplacée.

Vous pouvez penser à WriteOnceBlock comme étant similaire à une variable membre readonly en C #, sauf qu'au lieu d'être uniquement paramétrable dans un constructeur et ensuite immuable, il est seulement définissable une fois et est alors immuable.



Fractionnement des sorties potentielles d'une tâche

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try
    {
        result.Post(await task);
    }
    catch(Exception ex)
    {
```

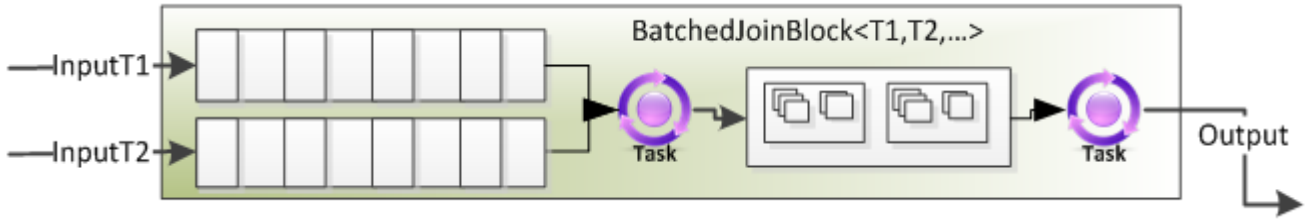
```
        exception.Post(ex);
    }
}
```

Introduction à TPL Dataflow par Stephen Toub

BatchedJoinBlock

(Recueille un certain nombre d'éléments totaux de 2-3 entrées et les regroupe dans un Tuple de collections d'éléments de données)

BatchedJoinBlock <T1, T2,…> est en quelque sorte une combinaison de BatchBlock et de JoinBlock <T1, T2,…>.
Alors que JoinBlock <T1, T2,…> est utilisé pour agréger une entrée de chaque cible dans un tuple et que BatchBlock est utilisé pour agréger N entrées dans une collection, BatchedJoinBlock <T1, T2,…> est utilisé pour collecter N entrées toutes les cibles en tuples de collections.



Scatter / Gather

Considérons un problème de dispersion / rassemblement où N opérations sont lancées, dont certaines peuvent réussir et produire des sorties de chaînes, et d'autres peuvent échouer et produire des exceptions.

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}
```

Introduction à TPL Dataflow par Stephen Toub

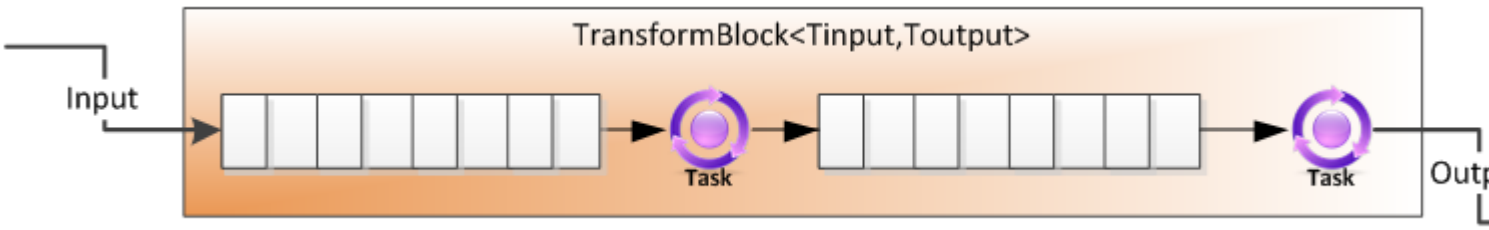
TransformBlock

(Sélectionnez un à un)

Comme avec ActionBlock, TransformBlock <TInput, TOutput> permet l'exécution d'un délégué pour effectuer certaines actions pour chaque donnée d'entrée; **contrairement à ActionBlock, ce traitement a une sortie.** Ce délégué peut être un Func <TInput, TOutput>, auquel cas le traitement de cet élément est considéré comme terminé lorsque le délégué revient, ou il peut s'agir d'un Func <TInput, Task>, auquel cas le traitement de cet élément est considéré comme terminé. lorsque le délégué renvoie, mais lorsque la tâche renvoyée est terminée. Pour ceux qui sont familiers avec LINQ, c'est un peu similaire à Select () dans la mesure où il prend une entrée, transforme cette entrée d'une certaine manière, puis produit une sortie.

Par défaut, TransformBlock <TInput, TOutput> traite ses données séquentiellement avec un MaxDegreeOfParallelism égal à 1. En plus de recevoir une entrée en mémoire tampon et de le traiter, ce bloc prend également toute sa sortie et son tampon traités (données qui n'ont pas été traitées et les données traitées).

Il comporte deux tâches: une pour traiter les données et une pour transmettre les données au bloc suivant.



Un pipeline concomitant

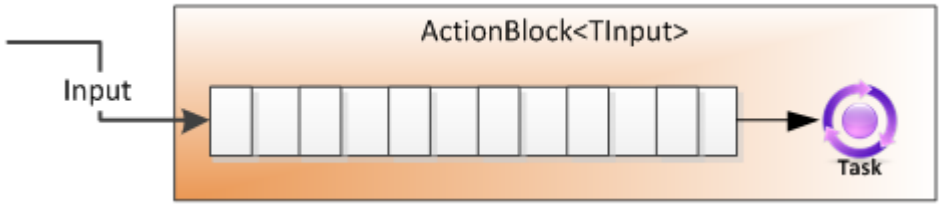
```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

Introduction à TPL Dataflow par Stephen Toub

ActionBlock

(pour chaque)

Cette classe peut être considérée logiquement comme un tampon pour les données à traiter combinées avec des tâches pour traiter ces données, avec le «bloc de flux de données» gérant les deux. Dans son utilisation la plus élémentaire, nous pouvons instancier un ActionBlock et y «publier» des données; le délégué fourni lors de la construction du ActionBlock sera exécuté de manière asynchrone pour chaque donnée envoyée.



Calcul synchrone

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Limiter les téléchargements asynchrones à 5 au maximum simultanément

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

Introduction à TPL Dataflow par Stephen Toub

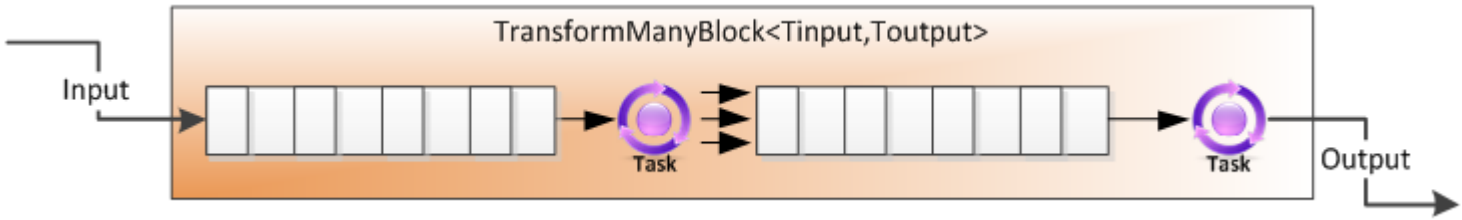
TransformManyBlock

(SelectMany, 1-m: les résultats de ce mappage sont «aplatis», tout comme le SelectMany de LINQ)

TransformManyBlock <TInput, TOutput> est très similaire à TransformBlock <TInput, TOutput>. La principale différence est qu'un TransformBlock <TInput, TOutput> produit une et une seule sortie pour chaque entrée, TransformManyBlock <TInput, TOutput> produit un nombre quelconque (zéro ou plus) de sorties pour chaque entrée. Comme avec ActionBlock et TransformBlock <TInput, TOutput>, ce traitement peut être spécifié à l'aide de délégués, à la fois pour le traitement synchrone et asynchrone.

Un Func <TInput, IEnumerable> est utilisé pour synchrone et un Func <TInput, Task <IEnumerable >> est utilisé pour asynchrone. Comme avec ActionBlock et TransformBlock <TInput, TOutput>, TransformManyBlock <TInput, TOutput> utilise par défaut un traitement séquentiel, mais peut être configuré autrement.

Le délégué de mappage exécute une collection d'éléments qui sont insérés individuellement dans le tampon de sortie.



Asynchrone Web Crawler

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Élargir un Enumerable dans ses éléments constitutifs

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Filtrage en passant de 1 à 0 ou 1 éléments

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Introduction à TPL Dataflow par Stephen Toub

BatchBlock

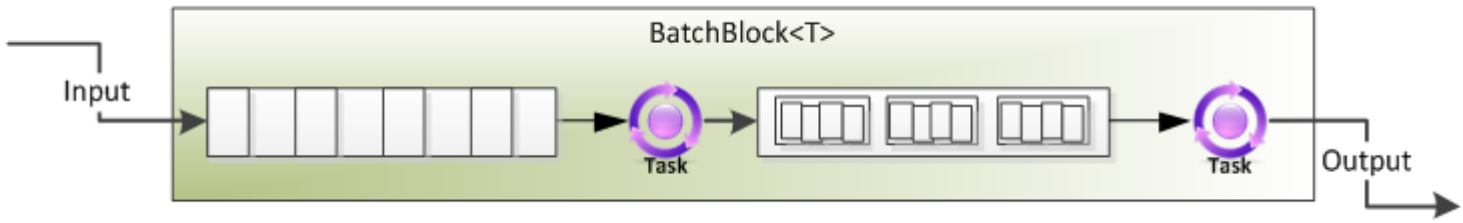
(Regroupe un certain nombre d'éléments de données séquentiels dans des ensembles d'éléments de données)

BatchBlock combine N éléments uniques en un seul lot, représenté sous la forme d'un tableau d'éléments. Une instance est créée avec une taille de lot spécifique, et le bloc crée ensuite un lot dès qu'il reçoit ce nombre d'éléments, produisant de manière asynchrone le lot dans le tampon de sortie.

BatchBlock est capable de s'exécuter dans des modes gourmands et non gourmands.

- Dans le mode gourmand par défaut, tous les messages proposés au bloc depuis un nombre quelconque de sources sont acceptés et mis en mémoire tampon pour être convertis en lots.
- ◦ En mode non gourmand, tous les messages sont reportés des sources jusqu'à ce que

suffisamment de sources aient offert des messages au bloc pour créer un lot. Ainsi, un BatchBlock peut être utilisé pour recevoir 1 élément de chacune des N sources, N éléments provenant de 1 source et une myriade d'options entre les deux.



Classement des demandes en groupes de 100 pour les soumettre à une base de données

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Créer un lot une fois par seconde

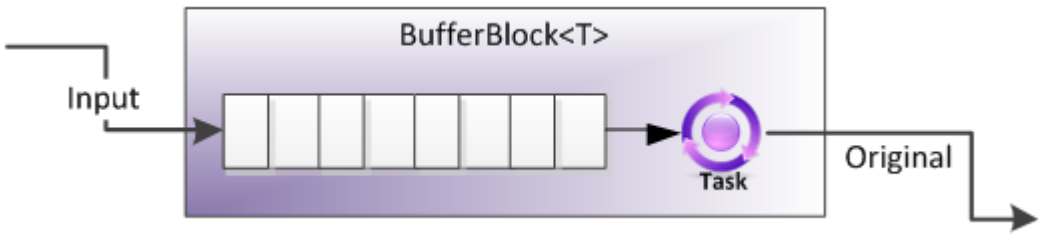
```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Introduction à TPL Dataflow par Stephen Toub

BufferBlock

(FIFO Queue: Les données qui entrent sont les données qui sortent)

En bref, BufferBlock fournit un tampon non lié ou limité pour stocker les instances de T.
Vous pouvez «publier» des instances de T sur le bloc, ce qui entraîne le stockage des données à enregistrer dans un ordre FIFO (first-in-first-out) par le bloc.
Vous pouvez «recevoir» du bloc, ce qui vous permet d'obtenir de manière synchrone ou asynchrone des instances de T précédemment stockées ou disponibles (encore une fois, FIFO).



Producteur / consommateur asynchrone avec un producteur étranglé

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
```

```
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

Introduction à TPL Dataflow par Stephen Toub

Lire Constructions de flux de données TPL (Task Parallel Library) en ligne: https://riptutorial.com/fr/csharp/topic/3110/constructions-de-flux-de-donnees-tpl--task-parallel-library-

Chapitre 29: Contexte de synchronisation dans Async-Await

Examples

Pseudocode pour asynchrone / en attente de mots-clés

Considérons une méthode asynchrone simple:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

En simplifiant, on peut dire que ce code signifie en réalité ce qui suit:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

Cela signifie que les mots-clés `async` / d' `await` utilisent le contexte de synchronisation actuel s'il existe. C'est-à-dire que vous pouvez écrire du code de bibliothèque qui fonctionnerait correctement dans les applications d'interface utilisateur, Web et console.

Article source

Désactivation du contexte de synchronisation

Pour désactiver le contexte de synchronisation, vous devez appeler la méthode `ConfigureAwait` :

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

. . .
```

```
Foo().ConfigureAwait(false);
```

> ConfigureAwait fournit un moyen d'éviter le comportement de capture par défaut de SynchronizationContext; Le fait de transmettre false pour le paramètre flowContext empêche le SynchronizationContext d'être utilisé pour reprendre l'exécution après l'attente.

Citation de It's All About le SynchronizationContext .

Pourquoi SynchronizationContext est-il si important?

Considérez cet exemple:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

Cette méthode gèle l'application de l'interface utilisateur jusqu'à ce que `RunTooLong` soit terminé. L'application ne répondra plus.

Vous pouvez essayer d'exécuter le code interne de manière asynchrone:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

Mais ce code ne s'exécutera pas car le corps interne peut être exécuté sur un thread non-interface utilisateur et il ne devrait pas modifier directement les propriétés de l'interface utilisateur :

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            lable1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Maintenant, n'oubliez pas de toujours utiliser ce modèle. Ou, essayez `SynchronizationContext.Post` qui le fera pour vous:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
```

```
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
        }, null);
    });
}
```

Lire Contexte de synchronisation dans Async-Await en ligne:
https://riptutorial.com/fr/csharp/topic/7381/contexte-de-synchronisation-dans-async-await

Chapitre 30: Contrats de code

Syntaxe

1. Contract.Requires (Condition, userMessage)

   Contract.Requires (Condition, userMessage)

   Contract.Result <T>

   Contract.Ensures ()

   Contract.Invariants ()

Remarques

.NET prend en charge l'idée de Design by Contract via sa classe Contracts trouvée dans l'espace de noms System.Diagnostics et introduite dans .NET 4.0. L'API de contrats de code inclut des classes pour les vérifications statiques et d'exécution du code et vous permet de définir des conditions préalables, des postconditions et des invariants au sein d'une méthode. Les conditions préalables spécifient les conditions que les paramètres doivent remplir avant d'exécuter une méthode, les conditions postérieures vérifiées à la fin d'une méthode et les invariants définissent les conditions qui ne changent pas pendant l'exécution d'une méthode.

Pourquoi les contrats de code sont-ils nécessaires?

Le suivi des problèmes d'une application lorsque votre application est en cours d'exécution est l'une des préoccupations majeures de tous les développeurs et administrateurs. Le suivi peut être effectué de plusieurs manières. Par exemple -

- Vous pouvez appliquer le suivi sur notre application et obtenir les détails d'une application lorsque l'application est en cours d'exécution
- Vous pouvez utiliser le mécanisme de journalisation des événements lorsque vous exécutez l'application. Les messages peuvent être vus à l'aide de l'Observateur d'événements
- Vous pouvez appliquer l'analyse des performances après un intervalle de temps spécifique et écrire des données en temps réel à partir de votre application.

Les contrats de code utilisent une approche différente pour le suivi et la gestion des problèmes au sein d'une application. Au lieu de valider tout ce qui est renvoyé par un appel de méthode, Contrats de code à l'aide de conditions préalables, de postconditions et d'invariants sur les méthodes, assurez-vous que tout ce qui entre et sort de vos méthodes est correct.

Examples

Conditions préalables

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

Postconditions

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

Invariants

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

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

        public Point()
        {
        }
```

```
        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }

            Console.WriteLine("X = {0}", this.X);
            Console.WriteLine("Y = {0}", this.Y);
        }

        [ContractInvariantMethod]
        private void ValidateCoordinates()
        {
            Contract.Invariant(this.X >= 0);
            Contract.Invariant(this.Y >= 0);
        }
    }
}
```

Définition de contrats sur l'interface

```
[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
```

```
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer
ID cannot be null!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password
cannot be null!!");
        }
    }
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID),"Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID),
"Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }

    }

    private string m_custID, m_PWD;

    public string CustomerID
    {
        get
        {
            return m_custID;
        }
        set
        {
            m_custID = value;
        }
    }

    public string Password
    {
        get
        {
            return m_PWD;
        }
        set
```

```
        {
            m_PWD = value;
        }
    }
}
```

Dans le code ci-dessus, nous avons défini une interface appelée `IValidation` avec un attribut `[ContractClass]` . Cet attribut prend l'adresse d'une classe où nous avons implémenté un contrat pour une interface. La classe `ValidationContract` utilise les propriétés définies dans l'interface et vérifie les valeurs NULL à l'aide de `Contract.Requires<T>` . `T` est une classe d'exception.

Nous avons également marqué l'accesseur get avec un attribut `[Pure]` . L'attribut pure garantit que la méthode ou une propriété ne modifie pas l'état d'instance d'une classe dans laquelle l'interface `IValidation` est implémentée.

Lire Contrats de code en ligne: https://riptutorial.com/fr/csharp/topic/4241/contrats-de-code

Chapitre 31: Contrats de code et assertions

Examples

Les affirmations pour vérifier la logique doivent toujours être vraies

Les assertions ne sont pas utilisées pour tester les paramètres d'entrée, mais pour vérifier que le flux de programme est conforme, c'est-à-dire que vous pouvez faire certaines hypothèses sur votre code à un moment donné. En d'autres termes: un test effectué avec `Debug.Assert` devrait *toujours* supposer que la valeur testée est `true` .

Debug.Assert ne s'exécute que dans les versions DEBUG; il est filtré des constructions RELEASE. En plus des tests unitaires, il doit être considéré comme un outil de débogage et non comme un remplacement des contrats de code ou des méthodes de validation des entrées.

Par exemple, c'est une bonne assertion:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Ici, assert est un bon choix car nous pouvons supposer que RetrieveSystemConfiguration () retournera une valeur valide et ne renverra jamais null.

Voici un autre bon exemple:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

Tout d'abord, nous pouvons supposer que RetrieveUserData () renverra une valeur valide. Ensuite, avant d'utiliser la propriété Age, nous vérifions l'hypothèse (qui devrait toujours être vraie) que l'âge de l'utilisateur est strictement positif.

Ceci est un mauvais exemple d'affirmation:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert n'est pas pour la validation des entrées car il est incorrect de supposer que cette assertion sera toujours vraie. Vous devez utiliser des méthodes de validation des entrées pour cela. Dans le cas ci-dessus, vous devez également vérifier que la valeur d'entrée est un nombre en premier lieu.

Lire Contrats de code et assertions en ligne: https://riptutorial.com/fr/csharp/topic/4349/contrats-

de-code-et-assertions

Chapitre 32: Conventions de nommage

Introduction

Cette rubrique présente certaines conventions de nommage de base utilisées lors de l'écriture dans le langage C #. Comme toutes les conventions, elles ne sont pas appliquées par le compilateur, mais assureront la lisibilité entre les développeurs.

Pour obtenir des directives complètes sur la conception de .NET Framework, consultez docs.microsoft.com/dotnet/standard/design-guidelines .

Remarques

Choisissez des noms d'identifiant facilement lisibles

Par exemple, une propriété nommée HorizontalAlignment est plus lisible en anglais que AlignmentHorizontal.

Favoriser la lisibilité sur la brièveté

Le nom de la propriété `CanScrollHorizontally` est meilleur que `ScrollableX` (une référence obscure à l'axe X).

Évitez d'utiliser des traits de soulignement, des tirets ou tout autre caractère non alphanumérique.

N'utilisez pas de notation hongroise

La notation hongroise est la pratique consistant à inclure un préfixe dans les identificateurs pour coder certaines métadonnées relatives au paramètre, telles que le type de données de l'identificateur, par exemple `string strName` .

Evitez également d'utiliser des identifiants en conflit avec les mots-clés déjà utilisés dans C #.

Abréviations et acronymes

En général, vous ne devez pas utiliser d’abréviations ou d’acronymes; ceux-ci rendent vos noms moins lisibles. De même, il est difficile de savoir quand on peut supposer qu'un acronyme est largement reconnu.

Examples

Conventions de capitalisation

Les termes suivants décrivent différentes manières d’identifier les cas.

Pascal Casing

La première lettre de l'identifiant et la première lettre de chaque mot concaténé ultérieur sont en majuscules. Vous pouvez utiliser la casse Pascal pour les identificateurs de trois caractères ou plus. Par exemple: `BackColor`

Camel Casing

La première lettre d'un identifiant est en minuscule et la première lettre de chaque mot concaténé ultérieur est en majuscule. Par exemple: `backColor`

Majuscule

Toutes les lettres de l'identifiant sont en majuscules. Par exemple: `IO`

Règles

Lorsqu'un identifiant est constitué de plusieurs mots, n'utilisez pas de séparateurs, tels que des traits de soulignement ("_") ou des tirets ("-"), entre les mots. Au lieu de cela, utilisez le boîtier pour indiquer le début de chaque mot.

Le tableau suivant résume les règles de capitalisation pour les identificateurs et fournit des exemples pour les différents types d'identificateurs:

Identifiant	Cas	Exemple
Variable locale	chameau	carName
Classe	Pascal	AppDomain
Type d'énumération	Pascal	ErrorLevel
Valeurs d'énumération	Pascal	Erreur fatale
un événement	Pascal	ValueChanged
Classe d'exception	Pascal	WebException
Champ statique en lecture seule	Pascal	RedValue
Interface	Pascal	IDisposable
Méthode	Pascal	ToString

Identifiant	Cas	Exemple
Espace de noms	Pascal	System.Drawing
Paramètre	chameau	typeName
Propriété	Pascal	Couleur de fond

Plus d'informations peuvent être trouvées sur MSDN .

Interfaces

Les interfaces doivent être nommées avec des noms ou des syntagmes nominaux, ou des adjectifs décrivant le comportement. Par exemple, `IComponent` utilise un nom descriptif, `ICustomAttributeProvider` utilise une expression nominale et `IPersistable` utilise un adjectif.

Les noms d'interface doivent être précédés de la lettre `I` pour indiquer que le type est une interface et que le cas Pascal doit être utilisé.

Vous trouverez ci-dessous des interfaces correctement nommées:

```
public interface IServiceProvider
public interface IFormatable
```

Champs privés

Il existe deux conventions communes pour les champs privés: `camelCase` et `_camelCaseWithLeadingUnderscore` .

Affaire de chameau

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

Housse camel avec soulignement

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;
```

```
    public Rational(int numerator, int denominator)
    {
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Espaces de noms

Le format général des espaces de noms est le suivant:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

Les exemples comprennent:

```
Fabrikam.Math
Litware.Security
```

Le préfixe des noms d'espaces de noms avec un nom de société empêche les espaces de noms de différentes sociétés de porter le même nom.

Enums

Utilisez un nom singulier pour la plupart des énumérations

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

Utilisez un nom pluriel pour les types Enum qui sont des champs de bits

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Remarque: Ajoutez toujours le `FlagsAttribute` *à un type Enum de champ de bits.*

Ne pas ajouter 'enum' comme suffixe

```
public enum VolumeEnum // Incorrect
```

N'utilisez pas le nom enum dans chaque entrée

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

Des exceptions

Ajouter 'exception' comme suffixe

Les noms d'exception personnalisés doivent être suffixés avec "-Exception".

Les exceptions ci-dessous sont correctement nommées:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Lire Conventions de nommage en ligne: https://riptutorial.com/fr/csharp/topic/2330/conventions-de-nommage

Chapitre 33: Conversion de type

Remarques

La conversion de type convertit un type de données en un autre type. Il est également connu sous le nom de Casting de type. En C #, le transtypage a deux formes:

Conversion de type implicite - Ces conversions sont effectuées par C # de manière sécurisée. Par exemple, les conversions de types intégraux plus petits à plus grands et les conversions de classes dérivées en classes de base.

Conversion de type explicite - Ces conversions sont effectuées explicitement par les utilisateurs utilisant les fonctions prédéfinies. Les conversions explicites nécessitent un opérateur de distribution.

Examples

Exemple d'opérateur implicite MSDN

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implict conversion called");
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implict conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

Sortie:

Chiffre pour doubler la conversion implicite appelée
double to Digit implique la conversion appelée
num = 7 dig2 = 12

Démo en direct sur .NET Fiddle

Conversion de type explicite

```
using System;
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

Lire Conversion de type en ligne: https://riptutorial.com/fr/csharp/topic/3489/conversion-de-type

Chapitre 34: Cordes Verbatim

Syntaxe

- @ "Les chaînes verbatim sont des chaînes dont le contenu n'est pas échappé. Dans ce cas, \ n ne représente pas le caractère de nouvelle ligne mais deux caractères individuels: \ et n. Les chaînes de caractères sont créées avec le caractère @"
- @ "Pour échapper aux guillemets," "les guillemets doubles" "sont utilisés."

Remarques

Pour concaténer des littéraux de chaîne, utilisez le symbole @ au début de chaque chaîne.

```
var combinedString = @"\t means a tab" + @" and \n means a newline";
```

Examples

Cordes multilignes

```
var multiLine = @"This is a

multiline paragraph";
```

Sortie:

C'est un

paragraphe multiligne

Démo en direct sur .NET Fiddle

Les chaînes multi-lignes contenant des guillemets doubles peuvent également être échappées comme sur une seule ligne, car elles sont textuelles.

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

Démo en direct sur .NET Fiddle

Il convient de noter que les espaces / tabulations au début des lignes 2 et 3 sont effectivement présents dans la valeur de la variable; vérifier cette question pour des solutions possibles.

Échapper à des citations doubles

Les doubles guillemets dans les chaînes verbatim peuvent être échappés en utilisant 2 doubles guillemets séquentiels `""` pour représenter un guillemet double `"` dans la chaîne résultante.

```
var str = @"""I don't think so,"" he said.";
Console.WriteLine(str);
```

Sortie:

> "Je ne pense pas", a-t-il dit.

Démo en direct sur .NET Fiddle

Cordes Verbatim Interpolées

Les chaînes verbatim peuvent être combinées avec les nouvelles fonctions d' interpolation String trouvées dans C # 6.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

Sortie:

> Test \ n 1 2 3
> Nouvelle ligne

Démo en direct sur .NET Fiddle

Comme prévu dans une chaîne verbatim, les barres obliques inverses sont ignorées en tant que caractères d'échappement. Et comme prévu d'une chaîne interpolée, toute expression entre accolades est évaluée avant d'être insérée dans la chaîne à cette position.

Les chaînes verbatim indiquent au compilateur de ne pas utiliser les caractères d'échappement

Dans une chaîne normale, la barre oblique inverse est le caractère d'échappement qui indique au compilateur de rechercher le ou les caractères suivants pour déterminer le caractère réel de la chaîne. (Liste complète des caractères échappés)

Dans les chaînes verbatim, il n'y a pas de caractère échappé (sauf pour `""` qui est transformé en un `"`). Pour utiliser une chaîne verbatim, ajoutez simplement un `@` avant les guillemets de départ.

Cette chaîne verbatim

```
var filename = @"c:\temp\newfile.txt"
```

Sortie:

c: \ temp \ newfile.txt

Par opposition à l'utilisation d'une chaîne ordinaire (non verbatim):

```
var filename = "c:\temp\newfile.txt"
```

ça va sortir:

```
c:    emp
ewfile.txt
```

en utilisant un personnage en train de s'échapper. (Le `\t` est remplacé par un caractère de tabulation et le `\n` est remplacé par une nouvelle ligne.)

Démo en direct sur .NET Fiddle

Lire Cordes Verbatim en ligne: https://riptutorial.com/fr/csharp/topic/16/cordes-verbatim

Chapitre 35: Courant

Examples

Utiliser des flux

Un flux est un objet qui fournit un moyen de transfert de données de bas niveau. Ils n'agissent pas eux-mêmes comme des conteneurs de données.

Les données que nous traitons sont sous forme de tableau d' `byte []` (`byte []`). Les fonctions de lecture et d'écriture sont toutes orientées octets, par exemple `WriteByte()` .

Il n'y a pas de fonctions pour traiter les entiers, les chaînes de caractères, etc. Cela rend le flux très général, mais il est moins simple de travailler si, par exemple, vous voulez simplement transférer du texte. Les flux peuvent être particulièrement utiles lorsque vous manipulez de grandes quantités de données.

Nous devrons utiliser différents types de flux sur la base desquels il doit être écrit / lu (c.-à-d. Le magasin de sauvegarde). Par exemple, si la source est un fichier, nous devons utiliser `FileStream` :

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

De même, `MemoryStream` est utilisé si le magasin de sauvegarde est en mémoire:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes(“C:\\file.txt”);

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
   // do stuff here...
}
```

De même, `System.Net.Sockets.NetworkStream` est utilisé pour l'accès au réseau.

Tous les flux sont dérivés de la classe générique `System.IO.Stream` . Les données ne peuvent pas être directement lues ou écrites à partir de flux. Le .NET Framework fournit des classes d'aide tels que `StreamReader` , `StreamWriter` , `BinaryReader` et `BinaryWriter` qui convertissent les types natifs et l'interface de flux de bas niveau, et transférer les données vers ou à partir du flux pour vous.

Lecture et écriture aux cours d' eau peuvent se faire via `StreamReader` et `StreamWriter` . Il faut être prudent lors de la fermeture de ceux-ci. Par défaut, la fermeture ferme également le flux contenu

et le rend inutilisable pour d'autres utilisations. Ce comportement par défaut peut être modifié en utilisant un constructeur ayant le paramètre `bool leaveOpen` et en définissant sa valeur sur `true` .

`StreamWriter` :

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.Write(NextLine);
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

`StreamReader` :

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();     This will close ms and when we try to use ms later it will cause an
exception
    sw.Flush();     //You can send the remaining data to stream. Closing will do this
automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Puisque les classes `Stream` , `StreamReader` , `StreamWriter` , etc. mettre en œuvre la `IDisposable` interface, nous pouvons appeler la `Dispose()` méthode sur des objets de ces classes.

Lire Courant en ligne: https://riptutorial.com/fr/csharp/topic/3114/courant

Chapitre 36: Création d'une application console à l'aide d'un éditeur de texte brut et du compilateur C # (csc.exe)

Examples

Création d'une application console à l'aide d'un éditeur de texte brut et du compilateur C

Pour utiliser un éditeur de texte brut afin de créer une application console écrite en C #, vous aurez besoin du compilateur C #. Le compilateur C # (csc.exe) se trouve à l'emplacement suivant: `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

NB Selon la version de .NET Framework installée sur votre système, vous devrez peut-être modifier le chemin ci-dessus en conséquence.

Sauvegarder le code

Le but de cette rubrique n'est pas de vous *apprendre* à écrire une application console, mais de vous apprendre à en *compiler* une [pour produire un seul fichier exécutable], avec uniquement le compilateur C # et tout éditeur de texte brut (tel que Bloc-notes).

1. Ouvrez la boîte de dialogue Exécuter en utilisant le raccourci clavier `Windows Key` + `R`
2. Tapez `notepad` , puis appuyez sur `Entrée`
3. Collez l'exemple de code ci-dessous dans le Bloc-notes
4. Enregistrez le fichier en tant que `ConsoleApp.cs` en accédant à **Fichier** → **Enregistrer sous ...** , puis en entrant `ConsoleApp.cs` dans le champ de texte "Nom du fichier", puis en sélectionnant `All Files` comme type de fichier.
5. Cliquez sur `Save`

Compiler le code source

1. Ouvrez la boîte de dialogue Exécuter en utilisant `Windows Key` + `R`
2. Entrez:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Maintenant, revenez à l'endroit où vous avez enregistré votre fichier `ConsoleApp.cs` . Vous devriez

maintenant voir un fichier exécutable (`ConsoleApp.exe`). Double-cliquez sur `ConsoleApp.exe` pour l'ouvrir.

C'est tout! Votre application console a été compilée. Un fichier exécutable a été créé et vous disposez maintenant d'une application de console fonctionnelle.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;

            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

Lire Création d'une application console à l'aide d'un éditeur de texte brut et du compilateur C #

(csc.exe) en ligne: https://riptutorial.com/fr/csharp/topic/6676/creation-d-une-application-console-a-l-aide-d-un-editeur-de-texte-brut-et-du-compilateur-c-sharp--csc-exe-

Chapitre 37: Créer son propre MessageBox dans l'application Windows Form

Introduction

Nous devons d'abord savoir ce qu'est une MessageBox ...

Le contrôle MessageBox affiche un message avec le texte spécifié et peut être personnalisé en spécifiant une image personnalisée, des jeux de titres et de boutons (ces jeux de boutons permettent à l'utilisateur de choisir plus d'une réponse de base oui / non).

En créant notre propre MessageBox, nous pouvons réutiliser ce contrôle MessageBox dans toutes les nouvelles applications en utilisant simplement le dll généré ou en copiant le fichier contenant la classe.

Syntaxe

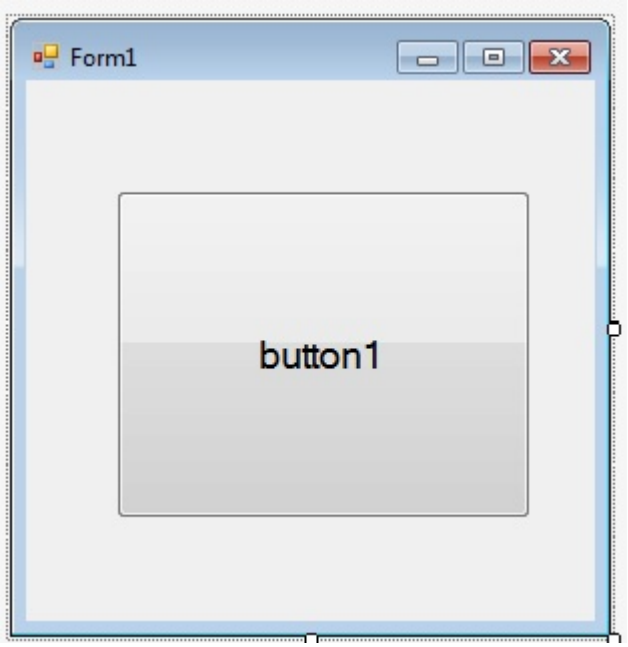
- 'statique DialogResult result = DialogResult.No; // DialogResult est renvoyé par des dialogues après le renvoi. '

Examples

Création du propre contrôle MessageBox.

Pour créer notre propre contrôle MessageBox, suivez simplement le guide ci-dessous ...

1. Ouvrez votre instance de Visual Studio (VS 2008/2010/2012/2015/2017)
2. Allez dans la barre d'outils en haut et cliquez sur Fichier -> Nouveau projet -> Application Windows Forms -> Donnez un nom au projet, puis cliquez sur OK.
3. Une fois chargé, faites glisser un contrôle de bouton depuis la Boîte à outils (situé à gauche) sur le formulaire (comme indiqué ci-dessous).



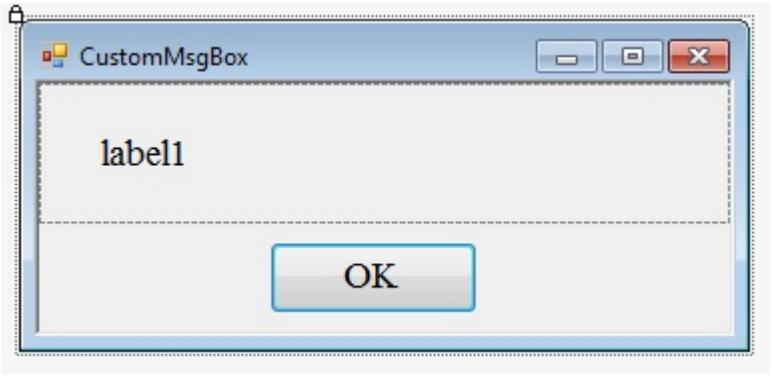
4. Double-cliquez sur le bouton et l'environnement de développement intégré générera automatiquement le gestionnaire d'événements click pour vous.

5. Modifiez le code du formulaire pour qu'il ressemble à ce qui suit (vous pouvez cliquer avec le bouton droit sur le formulaire et cliquer sur Modifier le code):

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
           CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. Explorateur de solutions -> Clic droit sur votre projet -> Ajouter -> Windows Form et définissez le nom comme "CustomMsgBox.cs"

7. Faites glisser un contrôle de bouton et d'étiquette de la boîte à outils vers le formulaire (cela ressemblera au formulaire ci-dessous après l'avoir fait):



8. Maintenant, écrivez le code ci-dessous dans le nouveau formulaire créé:

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}
```

9. Maintenant, lancez le programme en appuyant simplement sur la touche F5. Félicitations, vous avez fait un contrôle réutilisable.

Comment utiliser son propre contrôle MessageBox créé dans une autre application Windows Form.

Pour trouver vos fichiers .cs existants, cliquez avec le bouton droit sur le projet dans votre instance de Visual Studio, puis cliquez sur Ouvrir un dossier dans l'explorateur de fichiers.

1. Visual Studio -> Votre projet actuel (Windows Form) -> Explorateur de solutions -> Nom du projet -> Clic droit -> Ajouter -> Elément existant - Recherchez ensuite votre fichier .cs existant.

2. Maintenant, il y a une dernière chose à faire pour utiliser le contrôle. Ajoutez une instruction using à votre code pour que votre assembly connaisse ses dépendances.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
.
```

```
.
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. Pour afficher la boîte de message, utilisez simplement les éléments suivants ...

   CustomMsgBox.Show ("Votre message pour la boîte de message ...", "MSG", "OK");

Lire Créer son propre MessageBox dans l'application Windows Form en ligne: https://riptutorial.com/fr/csharp/topic/9788/creer-son-propre-messagebox-dans-l-application-windows-form

Chapitre 38: Cryptographie (System.Security.Cryptography)

Examples

Exemples modernes de chiffrement authentifié symétrique d'une chaîne

La cryptographie est quelque chose de très difficile et après avoir passé beaucoup de temps à lire différents exemples et à voir comment il est facile d'introduire une forme de vulnérabilité, j'ai trouvé une réponse écrite par @jbtule qui, à mon avis, est très bonne. Bonne lecture:

« La meilleure pratique générale pour le chiffrement symétrique est d'utiliser le chiffrement avec assermentée Associated données (AEAD), mais cela ne fait pas partie des standards .net bibliothèques crypto. Ainsi , le premier exemple utilise AES256 puis HMAC256 , deux pas Chiffrer puis MAC , qui nécessite plus de frais généraux et plus de clés.

Le deuxième exemple utilise la pratique plus simple de AES256- GCM en utilisant le Bouncy Castle open source (via nuget).

Les deux exemples ont une fonction principale qui prend la chaîne de message secrète, la ou les clé (s) et une charge utile et un retour non-secret facultatifs et une chaîne chiffrée authentifiée éventuellement complétée par les données non-secrètes. Idéalement, vous devriez les utiliser avec une ou des clés de 256 bits générées aléatoirement, voir `NewKey()` .

Les deux exemples ont également une méthode d'assistance qui utilise un mot de passe de chaîne pour générer les clés. Ces méthodes d'assistance sont fournies pour faciliter la comparaison avec d'autres exemples, mais elles sont *beaucoup moins sûres* car la force du mot de passe sera *beaucoup plus faible qu'une clé de 256 bits* .

Mise à jour: Ajout `byte[]` surcharges `byte[]` , et seul le Gist a le formatage complet avec 4 espaces indent et api docs en raison des limites de réponse StackOverflow. "

Cryptage intégré .NET (AES) -Then-MAC (HMAC) [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
```

```
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
HMac-Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
```

```
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
                        int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");
```

```
      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }

      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
```

```
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!",
KeyBitSize), "cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
```

```
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
              binaryWriter.Write(
                encryptedMessage,
                nonSecretPayloadLength + iv.Length,
                encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
              );
            }
            //Return Plain Text
            return plainTextStream.ToArray();
          }
        }
      }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length ==0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      int payloadIndex = nonSecretPayload.Length;

      byte[] cryptKey;
      byte[] authKey;
      //Use Random Salt to prevent pre-generated weak password attacks.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);
```

```
        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
      }

      //Deriving separate key, might be less efficient than using HKDF,
      //but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      }

      return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
authSalt.Length + nonSecretPayloadLength);
    }
  }
}
```

Château gonflable AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33
Base64
```

```
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
```

```
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
```

```
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
```

```
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +
nonSecretPayloadLength);
    }
  }
}
```

Introduction au chiffrement symétrique et asymétrique

Vous pouvez améliorer la sécurité du transit ou du stockage des données en mettant en œuvre des techniques de cryptage. Fondamentalement, il existe deux approches lors de l'utilisation de *System.Security.Cryptography* : **symétrique** et **asymétrique.**

Chiffrement symétrique

Cette méthode utilise une clé privée pour effectuer la transformation des données.

Avantages:

- Les algorithmes symétriques consomment moins de ressources et sont plus rapides que les asymétriques.
- La quantité de données que vous pouvez chiffrer est illimitée.

Les inconvénients:

- Le cryptage et le décryptage utilisent la même clé. Quelqu'un pourra déchiffrer vos données si la clé est compromise.
- Vous pourriez vous retrouver avec plusieurs clés secrètes à gérer si vous choisissez d'utiliser une clé secrète différente pour différentes données.

Sous System.Security.Cryptography, vous avez différentes classes qui effectuent un chiffrement symétrique, elles sont appelées chiffrement par bloc :

- AesManaged (algorithme AES).
- AesCryptoServiceProvider (plainte de l' algorithme AES FIPS 140-2).
- DESCryptoServiceProvider (algorithme DES).
- RC2CryptoServiceProvider (algorithme Rivest Cipher 2).
- RijndaelManaged (algorithme AES). *Remarque* : RijndaelManaged n'est **pas une** plainte FIPS-197 .
- TripleDES (algorithme TripleDES).

Chiffrement asymétrique

Cette méthode utilise une combinaison de clés publiques et privées pour effectuer la transformation des données.

Avantages:

- Il utilise des clés plus grandes que les algorithmes symétriques, ce qui les rend moins susceptibles d’être fissurés en utilisant la force brute.
- Il est plus facile de garantir qui est capable de chiffrer et de déchiffrer les données car elles reposent sur deux clés (publique et privée).

Les inconvénients:

- La quantité de données que vous pouvez crypter est limitée. La limite est différente pour chaque algorithme et est généralement proportionnelle à la taille de la clé de l'algorithme. Par exemple, un objet RSACryptoServiceProvider avec une longueur de clé de 1 024 bits ne peut chiffrer qu'un message inférieur à 128 octets.
- Les algorithmes asymétriques sont très lents par rapport aux algorithmes symétriques.

Sous System.Security.Cryptography, vous avez accès à différentes classes effectuant un chiffrement asymétrique:

- DSACryptoServiceProvider (algorithme d' algorithme de signature numérique)
- RSACryptoServiceProvider (algorithme d' algorithme RSA)

Hachage de mot de passe

Les mots de passe ne doivent jamais être stockés en texte brut! Ils doivent être hachés avec un sel généré aléatoirement (pour se défendre contre les attaques par table arc-en-ciel) en utilisant un algorithme de hachage à mot de passe lent. Un nombre élevé d'itérations (> 10k) peut être utilisé pour ralentir les attaques par force brute. Un délai de ~ 100 ms est acceptable pour un utilisateur qui se connecte, mais rend difficile la suppression d'un long mot de passe. Lorsque vous choisissez un nombre d'itérations, vous devez utiliser la valeur maximale tolérable pour votre application et l'augmenter à mesure que les performances de l'ordinateur s'améliorent. Vous devrez également envisager d'arrêter les requêtes répétées qui pourraient être utilisées comme une attaque par déni de service.

Lors du premier hachage, un sel peut être généré pour vous, le hachage et le sel obtenus peuvent alors être stockés dans un fichier.

```
private void firstHash(string userName, string userPassword, int numberOfItterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberOfItterations);
//Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberOfItterations); //Store the hashed
password with the salt and number of itterations to check against future password entries
}
```

Vérifier le mot de passe d'un utilisateur existant, lire son hash et son sel dans un fichier et le comparer au hachage du mot de passe saisi

```
private bool checkPassword(string userName, string userPassword, int numberOfItterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
numberOfItterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMach = comparePasswords(usersHash, hashedPassword);    //Compares byte
arrays
    return passwordsMach;
}
```

Cryptage de fichiers symétrique simple

L'exemple de code suivant illustre un moyen simple et rapide de chiffrer et de déchiffrer des fichiers à l'aide de l'algorithme de chiffrement symétrique AES.

Le code génère aléatoirement les vecteurs Salt et Initialization à chaque fois qu'un fichier est crypté, ce qui signifie que le cryptage du même fichier avec le même mot de passe entraînera toujours une sortie différente. Les valeurs salt et IV sont écrites dans le fichier de sortie afin que seul le mot de passe soit requis pour le déchiffrer.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);

        // Encrypt or decrypt the file
        using (var cryptoTransform = encryptMode
            ? cypher.CreateEncryptor(key, iv)
            : cypher.CreateDecryptor(key, iv))
        using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}
```

Données aléatoires sécurisées par cryptographie

Il y a des moments où la classe Random () du framework peut ne pas être considérée comme

suffisamment aléatoire, étant donné qu'elle est basée sur un générateur de nombres pseudo-aléatoires. Les classes Crypto du framework fournissent cependant quelque chose de plus robuste sous la forme de RNGCryptoServiceProvider.

Les exemples de code suivants montrent comment générer des tableaux, des chaînes et des nombres d'octets cryptographiquement sécurisés.

Tableau des octets aléatoires

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

Nombre entier aléatoire (avec distribution uniforme)

```
public static int GenerateRandomInt(int minVal=0, int maxVal=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxVal - minVal + 1) + minVal);
}
```

Chaîne aléatoire

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);

    // Generate the output string
    var allowable = allowableChars.ToCharArray();
    var l = allowable.Length;
    var chars = new char[length];
    for (var i = 0; i < length; i++)
        chars[i] = allowable[rnd[i] % l];

    return new string(chars);
}
```

Chiffrement de fichiers asymétrique rapide

Le chiffrement asymétrique est souvent considéré comme préférable au chiffrement symétrique pour transférer des messages à d'autres parties. Cela est principalement dû au fait qu'elle annule

de nombreux risques liés à l'échange d'une clé partagée et garantit que, même si une personne disposant de la clé publique peut chiffrer un message pour le destinataire, seul ce destinataire peut le déchiffrer. Malheureusement, le principal inconvénient des algorithmes de chiffrement asymétrique est qu'ils sont nettement plus lents que leurs cousins symétriques. En tant que tel, le cryptage asymétrique des fichiers, en particulier des fichiers de grande taille, peut souvent constituer un processus très complexe sur le plan informatique.

Afin de fournir à la fois sécurité et performance, une approche hybride peut être adoptée. Cela implique la génération aléatoire d'une clé et d'un vecteur d'initialisation pour le chiffrement *symétrique* . Ces valeurs sont ensuite chiffrées à l'aide d'un algorithme *asymétrique* et écrites dans le fichier de sortie avant d'être utilisées pour chiffrer les données source de manière *symétrique* et les ajouter à la sortie.

Cette approche fournit un haut niveau de performance et de sécurité, dans la mesure où les données sont chiffrées à l'aide d'un algorithme symétrique (rapide) et la clé et iv, tous deux générés de manière aléatoire (sécurisés) sont chiffrés par un algorithme asymétrique (sécurisé). Il a également l'avantage supplémentaire que la même charge utile chiffrée à différentes occasions aura un texte chiffré très différent, car les clés symétriques sont générées de manière aléatoire à chaque fois.

La classe suivante illustre le chiffrement asymétrique des chaînes et des tableaux d'octets, ainsi que le chiffrement des fichiers hybrides.

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more
robust but takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }
```

```
    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }

    public static string DecryptString(string value, string privateKey)
    {
        return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value),
privateKey));
    }

    #endregion

    #region Hybrid File Encryption and Decription

    public static void EncryptFile(string inputFilePath, string outputFilePath, string
publicKey)
    {
        using (var symmetricCypher = new AesManaged())
        {
            // Generate random key and IV for symmetric encryption
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(key);
                rng.GetBytes(iv);
            }

            // Encrypt the symmetric key and IV
            var buf = new byte[key.Length + iv.Length];
            Array.Copy(key, buf, key.Length);
            Array.Copy(iv, 0, buf, key.Length, iv.Length);
            buf = EncryptData(buf, publicKey);

            var bufLen = BitConverter.GetBytes(buf.Length);

            // Symmetrically encrypt the data and write it to the file, along with the
encrypted key and iv
            using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
            using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsOut.Write(bufLen,0, bufLen.Length);
                fsOut.Write(buf, 0, buf.Length);
                fsIn.CopyTo(cs);
            }
```

```
        }
    }

    public static void DecryptFile(string inputFilePath, string outputFilePath, string
privateKey)
    {
        using (var symmetricCypher = new AesManaged())
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        {
            // Determine the length of the encrypted key and IV
            var buf = new byte[sizeof(int)];
            fsIn.Read(buf, 0, buf.Length);
            var bufLen = BitConverter.ToInt32(buf, 0);

            // Read the encrypted key and IV data from the file and decrypt using the
asymmetric algorithm
            buf = new byte[bufLen];
            fsIn.Read(buf, 0, buf.Length);
            buf = DecryptData(buf, privateKey);

            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            Array.Copy(buf, key, key.Length);
            Array.Copy(buf, key.Length, iv, 0, iv.Length);

            // Decript the file data using the symmetric algorithm
            using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsIn.CopyTo(cs);
            }
        }
    }

    #endregion

    #region Key Storage

    public static void WritePublicKey(string publicKeyFilePath, string publicKey)
    {
        File.WriteAllText(publicKeyFilePath, publicKey);
    }
    public static string ReadPublicKey(string publicKeyFilePath)
    {
        return File.ReadAllText(publicKeyFilePath);
    }

    private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

    public static string ReadPrivateKey(string privateKeyFilePath, string password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        var cypherText = File.ReadAllBytes(privateKeyFilePath);

        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);
```

```
            using (var decryptor = cypher.CreateDecryptor(key, iv))
            using (var msDecrypt = new MemoryStream(cypherText))
            using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
                return srDecrypt.ReadToEnd();
            }
        }
    }

    public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string
password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var encryptor = cypher.CreateEncryptor(key, iv))
            using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
            using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor,
CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(privateKey);
            }
        }
    }

    #endregion
}
```

Exemple d'utilisation:

```
private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey,
privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);
```

```
    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}
```

Lire Cryptographie (System.Security.Cryptography) en ligne:
https://riptutorial.com/fr/csharp/topic/2988/cryptographie--system-security-cryptography-

Chapitre 39: Débordement

Examples

Débordement d'entier

Il existe une capacité maximale qu'un entier peut stocker. Et lorsque vous dépassez cette limite, elle retournera au côté négatif. Pour `int` , c'est `2147483647`

```
int x = int.MaxValue;                //MaxValue is 2147483647
x = unchecked(x + 1);                //make operation explicitly unchecked so that the example
also works when the check for arithmetic overflow/underflow is enabled in the project settings

Console.WriteLine(x);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

Pour tous les nombres entiers hors de cette plage, utilisez namespace System.Numerics qui a le type de données BigInteger. Consultez le lien ci-dessous pour plus d'informations
https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx

Débordement pendant le fonctionnement

Le débordement se produit également pendant l'opération. Dans l'exemple suivant, x est un `int` , 1 est un `int` par défaut. Par conséquent , l' addition est un `int` addition. Et le résultat sera un `int` . Et ça va déborder.

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1;                     //It will be overflown
Console.WriteLine(y);               //Will print -2147483648
Console.WriteLine(int.MinValue);    //Same as Min value
```

Vous pouvez empêcher cela en utilisant 1L. Maintenant, 1 sera un `long` et l'addition sera un `long` ajout

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1L;                    //It will be OK
Console.WriteLine(y);               //Will print 2147483648
```

Questions de commande

Il y a un débordement dans le code suivant

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

Considérant que dans le code suivant il n'y a pas de débordement

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

Cela est dû à l'ordre de gauche à droite des opérations. Dans le premier code, le fragment `x + x` déborde et devient ensuite `long` . Par contre, `x + 1L` devient `long` et après cela, `x` est ajouté à cette valeur.

Lire Débordement en ligne: https://riptutorial.com/fr/csharp/topic/3303/debordement

Chapitre 40: Déclaration de verrouillage

Syntaxe

- verrouiller (obj) {}

Remarques

À l'aide de l'instruction de `lock` , vous pouvez contrôler l'accès des différents threads au code dans le bloc de code. Il est couramment utilisé pour empêcher les conditions de concurrence, par exemple, la lecture et la suppression d'éléments multiples dans une collection. Comme le verrouillage oblige les threads à attendre que d'autres threads quittent un bloc de code, cela peut entraîner des retards qui pourraient être résolus avec d'autres méthodes de synchronisation.

MSDN

> Le mot-clé lock marque un bloc d'instructions en tant que section critique en obtenant le verrou d'exclusion mutuelle pour un objet donné, en exécutant une instruction, puis en libérant le verrou.
>
> Le mot-clé lock garantit qu'un thread n'entre pas dans une section critique du code alors qu'un autre thread se trouve dans la section critique. Si un autre thread tente d'entrer un code verrouillé, il attendra, bloquera, jusqu'à ce que l'objet soit libéré.
>
> La meilleure pratique consiste à définir un objet **privé** à verrouiller ou une variable d'objet **statique privée** pour protéger les données communes à toutes les instances.

Dans C # 5.0 et versions ultérieures, la déclaration de `lock` est équivalente à:

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

Pour C # 4.0 et les versions antérieures, l'instruction de `lock` est équivalente à:

```
System.Threading.Monitor.Enter(refObject);
try
{
    // code
}
```

```
finally
{
     System.Threading.Monitor.Exit(refObject);
}
```

Examples

Usage simple

L'utilisation commune du `lock` est une section critique.

Dans l'exemple suivant, `ReserveRoom` doit être appelé à partir de différents threads. La synchronisation avec le `lock` est le moyen le plus simple de prévenir les conditions de course. Le corps de la méthode est entouré d'un `lock` qui garantit que deux threads ou plus ne peuvent pas l'exécuter simultanément.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

Si un thread atteint `lock` bloc verrouillé pendant qu'un autre thread s'exécute dans celui-ci, le premier attendra un autre pour quitter le bloc.

> La meilleure pratique consiste à définir un objet privé à verrouiller ou une variable d'objet statique privée pour protéger les données communes à toutes les instances.

Lancer une exception dans une déclaration de verrouillage

Le code suivant libère le verrou. Il n'y aura pas de problème. L'énoncé de verrouillage des coulisses fonctionne comme `try finally`

```
lock(locker)
{
    throw new Exception();
}
```

Plus peut être vu dans la spécification C # 5.0 :

Une déclaration de `lock` du formulaire

```
lock (x) ...
```

où `x` est une expression d'un *type de référence* , est précisément équivalent à

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

sauf que `x` n'est évalué qu'une seule fois.

Retourner dans une déclaration de verrouillage

Le code suivant libère le verrou.

```
lock(locker)
{
    return 5;
}
```

Pour une explication détaillée, cette réponse SO est recommandée.

Utilisation d'instances d'objet pour le verrouillage

Lorsque vous utilisez la déclaration de `lock` intégrée à C #, une instance de quelque type est nécessaire, mais son état importe peu. Une instance d' `object` est parfaite pour cela:

```
public class ThreadSafe {
  private static readonly object locker = new object();


  public void SomeThreadSafeMethod() {
    lock (locker) {
      // Only one thread can be here at a time.
    }
  }
}
```

NB les instances de `Type` ne doivent pas être utilisées pour cela (dans le code au-dessus de `typeof(ThreadSafe)`) car les instances de `Type` sont partagées entre AppDomains et l'étendue du verrou peut donc inclure du code (par exemple, si `ThreadSafe` est chargé dans deux AppDomains dans le même processus puis verrouiller sur son instance `Type` se verrouilleraient mutuellement).

Anti-Patterns et Gotchas

Verrouillage sur une variable allouée / locale

L'une des erreurs lors de l'utilisation de `lock` est l'utilisation d'objets locaux comme casier dans une fonction. Étant donné que ces instances d'objet local différeront à chaque appel de la fonction, le `lock` ne fonctionnera pas comme prévu.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

En supposant que le verrouillage restreint l'accès à l'objet de synchronisation lui-même

Si un thread appelle: `lock(obj)` et un autre thread appelle `obj.ToString()` deuxième thread ne sera pas bloqué.

```
object obj = new Object();

public void SomeMethod()
{
     lock(obj)
    {
       //do dangerous stuff
    }
 }

 //Meanwhile on other tread
 public void SomeOtherMethod()
 {
   var objInString = obj.ToString(); //this does not block
 }
```

Attend des sous-classes pour savoir quand verrouiller

Parfois, les classes de base sont conçues de telle sorte que leurs sous-classes doivent utiliser un verrou lors de l'accès à certains champs protégés:

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

Il est beaucoup plus sûr d' *encapsuler le verrouillage* en utilisant une méthode de modèle :

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
```

```
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

Le verrouillage sur une variable ValueType en boîte ne se synchronise pas

Dans l'exemple suivant, une variable privée est implicitement encadrée car elle est fournie en tant qu'argument d' `object` à une fonction, s'attendant à ce qu'une ressource de moniteur se verrouille. La boxe se produit juste avant d'appeler la fonction IncInSync, donc l'instance encadrée correspond à un objet de segment différent à chaque appel de la fonction.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

La boxe se produit dans la fonction `Inc` :

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld       UserQuery+BulemicCounter.counterLock
IL_0008:  box         System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

Cela ne signifie pas qu'un ValueType en boîte ne peut pas être utilisé pour le verrouillage du

moniteur:

```
private readonly object counterLock = 1;
```

Maintenant, la boxe se produit dans le constructeur, ce qui est bien pour le verrouillage:

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

Utiliser des verrous inutilement lorsqu'une alternative plus sûre existe

Un modèle très courant consiste à utiliser une `List` ou un `Dictionary` privé dans une classe de thread thread et à verrouiller chaque fois qu'il est accessible:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

S'il y a plusieurs méthodes qui accèdent au dictionnaire de `values` , le code peut devenir très long et, plus important encore, le verrouillage tout le temps en cache l' *intention* . Le verrouillage est également très facile à oublier et le manque de verrouillage peut rendre très difficile la recherche de bogues.

En utilisant un `ConcurrentDictionary` , nous pouvons éviter de verrouiller complètement:

```
public class Cache
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
```

```
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

L'utilisation de collections simultanées améliore également les performances car toutes utilisent des techniques sans verrouillage dans une certaine mesure.

Lire Déclaration de verrouillage en ligne: https://riptutorial.com/fr/csharp/topic/1495/declaration-de-verrouillage

Chapitre 41: Délégués Func

Syntaxe

- `public delegate TResult Func<in T, out TResult>(T arg)`
- `public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)`
- `public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)`
- `public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)`

Paramètres

Paramètre	Détails
`arg` ou `arg1`	le (premier) paramètre de la méthode
`arg2`	le second paramètre de la méthode
`arg3`	le troisième paramètre de la méthode
`arg4`	le quatrième paramètre de la méthode
`T` ou `T1`	le type du (premier) paramètre de la méthode
`T2`	le type du deuxième paramètre de la méthode
`T3`	le type du troisième paramètre de la méthode
`T4`	le type du quatrième paramètre de la méthode
`TResult`	le type de retour de la méthode

Examples

Sans paramètres

Cet exemple montre comment créer un délégué qui encapsule la méthode qui renvoie l'heure actuelle

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}
```

```
static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}
```

Avec plusieurs variables

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}
```

Méthodes Lambda & Anonymous

Une méthode anonyme peut être affectée partout où un délégué est attendu:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Les expressions lambda peuvent être utilisées pour exprimer la même chose:

```
Func<int, int> square = x => x * x;
```

Dans les deux cas, nous pouvons maintenant appeler la méthode stockée dans un `square` comme ceci:

```
var sq = square.Invoke(2);
```

Ou comme un raccourci:

```
var sq = square(2);
```

Notez que pour que l'affectation soit de type sécurisé, les types de paramètre et le type de retour de la méthode anonyme doivent correspondre à ceux du type délégué:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

Paramètres de type covariant et contre-polarisant

`Func` prend également en charge Covariant & Contravariant

```
// 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;

    }
}
```

Lire Délégués Func en ligne: https://riptutorial.com/fr/csharp/topic/2769/delegues-func

Chapitre 42: Des minuteries

Syntaxe

- `myTimer.Interval` - définit la fréquence à laquelle l'événement "Tick" est appelé (en millisecondes)
- `myTimer.Enabled` - valeur booléenne qui définit le temporisateur à `myTimer.Enabled` / désactiver
- `myTimer.Start()` - Démarre la minuterie.
- `myTimer.Stop()` - Arrête le minuteur.

Remarques

Si vous utilisez Visual Studio, les minuteurs peuvent être ajoutés en tant que contrôle directement à votre formulaire à partir de la boîte à outils.

Examples

Temporisateurs Multithread

`System.Threading.Timer` - Minuteur multithread le plus simple. Contient deux méthodes et un constructeur.

Exemple: une minuterie appelle la méthode DataWrite, qui écrit "multithread execute ..." au bout de cinq secondes, puis toutes les secondes jusqu'à ce que l'utilisateur appuie sur Enter:

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
    Console.ReadLine();
    timer.Dispose(); // This both stops the timer and cleans up.
  }

  static void DataWrite (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data); // Writes "multithread executed..."
  }
}
```

Remarque: Publiera une section distincte pour disposer de minuteries multithread.

`Change` - Cette méthode peut être appelée lorsque vous souhaitez modifier l'intervalle du minuteur.

`Timeout.Infinite` - Si vous souhaitez tirer une seule fois. Spécifiez ceci dans le dernier argument

du constructeur.

`System.Timers` - Une autre classe de temporisateur fournie par .NET Framework. Il enveloppe le `System.Threading.Timer` .

Caractéristiques:

- `IComponent` - `IComponent` dans le plateau de composants du concepteur de Visual Studio
- Propriété `Interval` au lieu d'une méthode `Change`
- `event Elapsed` au lieu d'un `delegate` rappel
- Propriété `Enabled` pour démarrer et arrêter le minuteur (`default value = false`)
- Méthodes `Start` & `Stop` en cas de confusion avec la propriété `Enabled` (au-dessus du point)
- `AutoReset` - pour indiquer un événement récurrent (`default value = true`)
- Propriété `SynchronizingObject` avec les méthodes `Invoke` et `BeginInvoke` pour appeler des méthodes en toute sécurité sur des éléments WPF et des contrôles Windows Forms

Exemple représentant toutes les fonctionnalités ci-dessus:

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
  static void Main()
  {
    Timer timer = new Timer(); // Doesn't require any args
    timer.Interval = 500;
    timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
    Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
    timer.Start(); // Restart the timer
    Console.ReadLine();
    timer.Dispose(); // Permanently stop the timer
 }

 static void timer_Elapsed(object sender, EventArgs e)
 {
   Console.WriteLine ("Tick");
 }
}
```

`Multithreaded timers` - utilisez le pool de threads pour permettre à quelques threads de servir plusieurs minuteries. Cela signifie que la méthode de rappel ou l'événement `Elapsed` peut se déclencher sur un thread différent à chaque appel.

`Elapsed` - Cet événement se déclenche toujours à l'heure - que l'événement `Elapsed` précédent ait ou non été exécuté. Pour cette raison, les rappels ou les gestionnaires d'événements doivent être sécurisés pour les threads. La précision des temporisateurs multithread dépend du système d'exploitation et se situe généralement entre 10 et 20 ms.

`interop` - quand vous avez besoin d'une plus grande précision, utilisez ceci et appelez le minuteur

multimédia Windows. Cela a une précision de 1 ms et est défini dans `winmm.dll` .

`timeBeginPeriod` - Appelez ceci en premier pour informer OS que vous avez besoin d'une précision de synchronisation élevée

`timeSetEvent` - appelle cela après `timeBeginPeriod` pour démarrer une minuterie multimédia.

`timeKillEvent` - appelez cela lorsque vous avez terminé, cela arrête le minuteur

`timeEndPeriod` - Appelez ceci pour informer le système d'exploitation que vous n'avez plus besoin d'une précision de synchronisation élevée.

Vous pouvez trouver des exemples complets sur Internet qui utilisent la minuterie multimédia en recherchant les mots clés `dllimport winmm.dll timesetevent` .

Créer une instance d'un temporisateur

Les temporisateurs permettent d'effectuer des tâches à des intervalles de temps spécifiques (Do X toutes les Y secondes). Vous trouverez ci-dessous un exemple de création d'une nouvelle instance de Timer.

REMARQUE : Ceci s'applique aux minuteries utilisant WinForms. Si vous utilisez WPF, vous voudrez peut-être regarder dans `DispatcherTimer`

```
    using System.Windows.Forms; //Timers use the Windows.Forms namespace

    public partial class Form1 : Form
    {

        Timer myTimer = new Timer(); //create an instance of Timer named myTimer


        public Form1()
        {
            InitializeComponent();
        }

    }
```

Affectation du gestionnaire d'événements "Tick" à une minuterie

Toutes les actions effectuées dans une minuterie sont traitées dans l'événement "Tick".

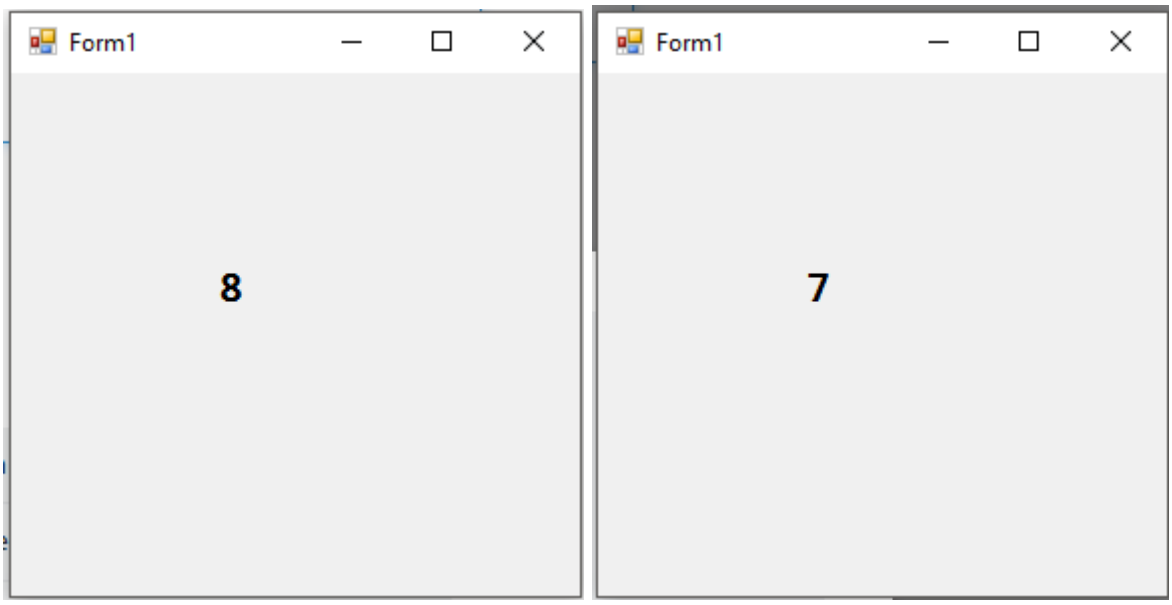
```
public partial class Form1 : Form
{

    Timer myTimer = new Timer();


    public Form1()
    {
        InitializeComponent();
```

```
        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

Exemple: utiliser une minuterie pour effectuer un simple compte à rebours.

```
    public partial class Form1 : Form
    {

    Timer myTimer = new Timer();
    int timeLeft = 10;

        public Form1()
        {
            InitializeComponent();

            //set properties for the Timer
            myTimer.Interval = 1000;
            myTimer.Enabled = true;

            //Set the event handler for the timer, named "myTimer_Tick"
            myTimer.Tick += myTimer_Tick;

            //Start the timer as soon as the form is loaded
            myTimer.Start();

            //Show the time set in the "timeLeft" variable
            lblCountDown.Text = timeLeft.ToString();

        }

        private void myTimer_Tick(object sender, EventArgs e)
        {
            //perform these actions at the interval set in the properties.
            lblCountDown.Text = timeLeft.ToString();
            timeLeft -= 1;

            if (timeLeft < 0)
            {
                myTimer.Stop();
            }
        }
    }
```

Résulte en...



Etc...

Lire Des minuteries en ligne: https://riptutorial.com/fr/csharp/topic/3829/des-minuteries

Chapitre 43: Diagnostic

Examples

Debug.WriteLine

Ecrit dans les écouteurs de trace de la collection Listeners lorsque l'application est compilée dans la configuration de débogage.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

Dans Visual Studio ou Xamarin Studio, cela apparaîtra dans la fenêtre Sortie d'application. Cela est dû à la présence du programme d' écoute de trace par défaut dans TraceListenerCollection.

Redirection de la sortie du journal avec TraceListeners

Vous pouvez rediriger la sortie de débogage vers un fichier texte en ajoutant un TextWriterTraceListener à la collection Debug.Listeners.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

Vous pouvez rediriger la sortie de débogage vers le flux de sortie d'une application de console à l'aide de ConsoleTraceListener.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Lire Diagnostic en ligne: https://riptutorial.com/fr/csharp/topic/2147/diagnostic

Chapitre 44: Directives du préprocesseur

Syntaxe

- #define *[symbole]* // Définit un symbole de compilation.
- #undef *[symbole]* // Annule un symbole du compilateur.
- #warning *[message d'avertissement]* // Génère un avertissement de compilation. Utile avec #if.
- # *erreur [message d'erreur]* // Génère une erreur de compilation. Utile avec #if.
- #line *[numéro de ligne] (nom du fichier)* // Remplace le numéro de ligne du compilateur (et éventuellement le nom du fichier source). Utilisé avec les modèles de texte T4 .
- #pragma warning [disable | restore] *[numéros d'avertissement]* // Désactive / restaure les avertissements du compilateur.
- #pragma checksum " *[nomfichier]* " " *[guid]* " " *[somme de contrôle]* " // Valide le contenu du fichier source.
- #region *[nom de la région]* // Définit une région de code réductible.
- #endregion // Termine un bloc de région de code.
- #if *[condition]* // Exécute le code ci-dessous si la condition est vraie.
- #else // Utilisé après un #if.
- #elif *[condition]* // Utilisé après un #if.
- #endif // Termine un bloc conditionnel démarré avec #if.

Remarques

Les directives de préprocesseur sont généralement utilisées pour faciliter la modification des programmes sources et leur compilation dans différents environnements d'exécution. Les directives du fichier source indiquent au préprocesseur d'effectuer des actions spécifiques. Par exemple, le préprocesseur peut remplacer des jetons dans le texte, insérer le contenu d'autres fichiers dans le fichier source ou supprimer la compilation d'une partie du fichier en supprimant des sections de texte. Les lignes de préprocesseur sont reconnues et exécutées avant l'expansion de la macro. Par conséquent, si une macro se développe en quelque chose qui ressemble à une commande de préprocesseur, cette commande n'est pas reconnue par le préprocesseur.

Les instructions de préprocesseur utilisent le même jeu de caractères que les instructions de fichier source, sauf que les séquences d'échappement ne sont pas prises en charge. Le jeu de caractères utilisé dans les instructions de préprocesseur est identique au jeu de caractères d'exécution. Le préprocesseur reconnaît également les valeurs de caractères négatives.

Expressions conditionnelles

Les expressions conditionnelles (`#if` , `#elif` , etc.) prennent en charge un sous-ensemble limité d'opérateurs booléens. Elles sont:

- `==` et `!=` . Celles-ci ne peuvent être utilisées que pour tester si le symbole est vrai (défini) ou

faux (non défini)

- `&&` , `||` , `!`
- `()`

Par exemple:

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

compilerait le code qui imprime "OK!" à la console si `DEBUG` n'est pas défini, `SOME_SYMBOL` ou `SOME_OTHER_SYMBOL` est défini et `RELEASE` est défini.

Remarque: Ces substitutions sont effectuées *au moment de la compilation* et ne sont donc pas disponibles pour inspection au moment de l'exécution. Le code éliminé par l'utilisation de `#if` ne fait pas partie de la sortie du compilateur.

Voir aussi: Directives de préprocesseur C # sur MSDN.

Examples

Expressions conditionnelles

Lorsque ce qui suit est compilé, il renverra une valeur différente selon les directives définies.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

Les expressions conditionnelles sont généralement utilisées pour enregistrer des informations supplémentaires pour les versions de débogage.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
```

```
    }
}
```

Génération des avertissements et des erreurs du compilateur

Les avertissements du compilateur peuvent être générés à l'aide de la directive `#warning` , et des erreurs peuvent également être générées à l'aide de la directive `#error` .

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

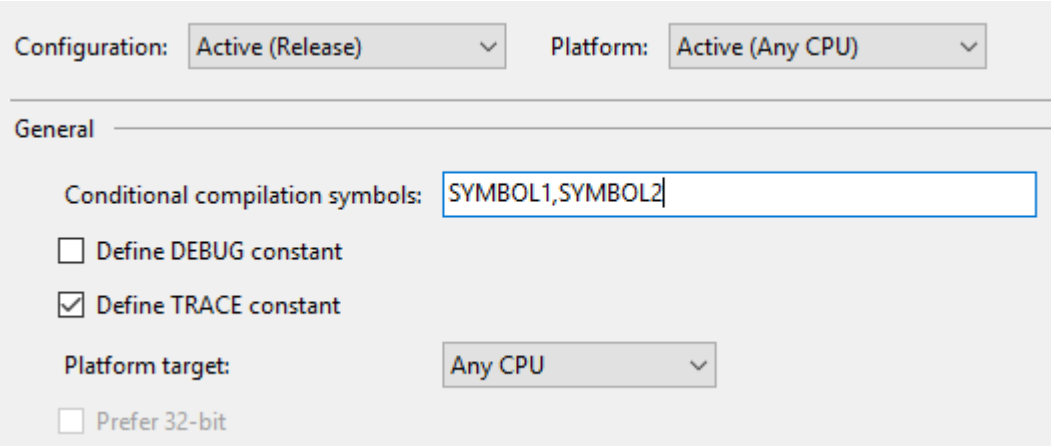
Définition et suppression de symboles

Un symbole de compilateur est un mot clé défini au moment de la compilation, qui peut être vérifié pour exécuter certaines sections de code de manière conditionnelle.

Il existe trois façons de définir un symbole de compilateur. Ils peuvent être définis via le code:

```
#define MYSYMBOL
```

Ils peuvent être définis dans Visual Studio, sous Propriétés du projet> Créer> Symboles de compilation conditionnels:



(Notez que `DEBUG` et `TRACE` ont leurs propres cases à cocher et qu'il n'est pas nécessaire de les spécifier explicitement.)

Ou ils peuvent être définis à la compilation en utilisant le commutateur `/define:[name]` sur le compilateur C #, `csc.exe` .

Vous pouvez également `#undefine` symboles non définis à l'aide de la directive `#undefine` .

L'exemple le plus courant est le symbole `DEBUG` , défini par Visual Studio lorsqu'une application est compilée en mode Debug (en mode Release).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
#if DEBUG
        System.Diagnostics.Trace.WriteLine("Unhandled exception!");
        System.Diagnostics.Trace.WriteLine(ex);
        throw;
#else
        LoggingFramework.LogError(ex);
        DisplayFriendlyErrorMessage();
#endif
    }
}
```

Dans l'exemple ci-dessus, lorsqu'une erreur survient dans la logique métier de l'application, si l'application est compilée en mode débogage (et que le symbole `DEBUG` est défini), l'erreur sera consignée dans le journal de suivi et l'exception sera supprimée. -pour le débogage. Toutefois, si l'application est compilée en mode Release (et qu'aucun symbole `DEBUG` n'est défini), une structure de journalisation est utilisée pour consigner discrètement l'erreur et un message d'erreur convivial s'affiche pour l'utilisateur final.

Blocs de région

Utilisez `#region` et `#endregion` pour définir une région de code `#endregion` .

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

Ces directives ne sont utiles que lorsqu'un IDE prenant en charge des régions réductibles (telles que Visual Studio) est utilisé pour modifier le code.

Autres instructions du compilateur

Ligne

`#line` contrôle le numéro de ligne et le nom de fichier signalés par le compilateur lors de la sortie des avertissements et des erreurs.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

Somme de contrôle Pragma

`#pragma checksum` permet de spécifier une somme de contrôle spécifique pour une base de données de programme (PDB) générée pour le débogage.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

Utilisation de l'attribut conditionnel

L'ajout d'un attribut `Conditional` de l'espace de noms `System.Diagnostics` à une méthode est un moyen efficace de contrôler les méthodes appelées dans vos builds et celles qui ne le sont pas.

```
#define EXAMPLE_A

using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
    static void ExampleA() {...}

    [Conditional("EXAMPLE_B")]
    static void ExampleB() {...}
}
```

Désactiver et restaurer les avertissements du compilateur

Vous pouvez désactiver les avertissements du compilateur en utilisant l'avertissement `#pragma warning disable` et les restaurer à l'aide de la `#pragma warning restore` :

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;
```

```
#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Les numéros d'avertissement séparés par des virgules sont autorisés:

```
#pragma warning disable CS0168, CS0219
```

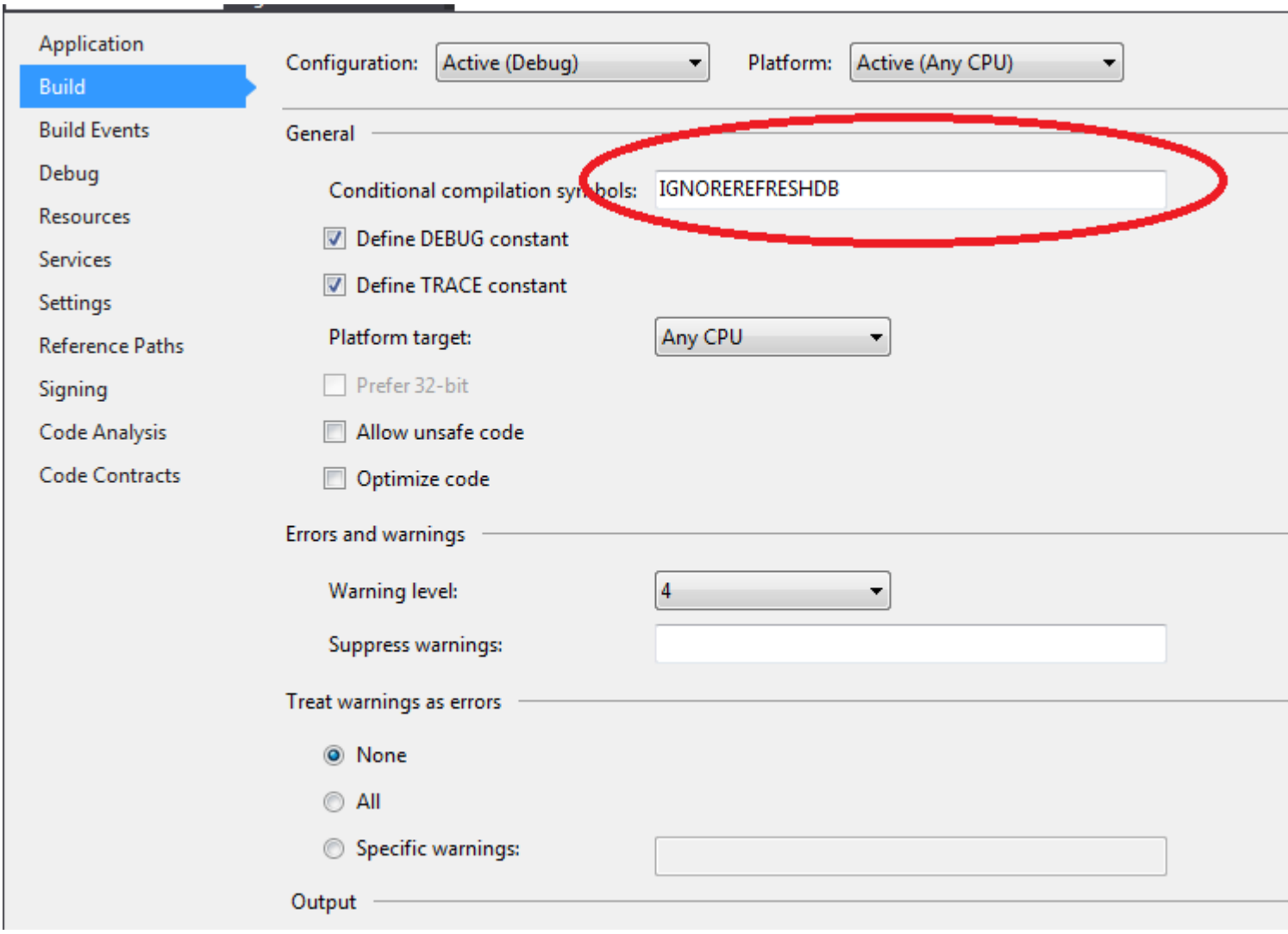
Le préfixe `CS` est facultatif et peut même être mélangé (bien que ce ne soit pas une bonne pratique):

```
#pragma warning disable 0168, 0219, CS0414
```

Préprocesseurs personnalisés au niveau du projet

Il est pratique de définir un prétraitement conditionnel personnalisé au niveau du projet lorsque certaines actions doivent être ignorées, disons pour les tests.

Accédez à l' `Solution Explorer` -> Cliquez avec le `bouton droit de la souris` sur le projet que vous souhaitez définir comme variable -> `Properties` -> `Build` -> En général Rechercher champ `Conditional compilation symbols` et entrez votre variable conditionnelle ici



Exemple de code qui ignorera du code:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

Lire Directives du préprocesseur en ligne: https://riptutorial.com/fr/csharp/topic/755/directives-du-preprocesseur

Chapitre 45: En boucle

Examples

Styles de boucle

Tandis que

Le type de boucle le plus trivial. Le seul inconvénient est qu'il n'y a pas d'indice intrinsèque pour savoir où vous êtes dans la boucle.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Faire

Semblable à `while` , mais la condition est évaluée à la fin de la boucle au lieu du début. Cela se traduit par l'exécution des boucles au moins une fois.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

Pour

Un autre style de boucle trivial. En boucle, un index (`i`) est augmenté et vous pouvez l'utiliser. Il est généralement utilisé pour manipuler des tableaux.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Pour chaque

`IEnumarable` modernisée de bouclage à travers les objets `IEnumarable` . Heureusement, vous n'avez pas à penser à l'index de l'élément ou au nombre d'éléments de la liste.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

Méthode Foreach

Alors que les autres styles sont utilisés pour sélectionner ou mettre à jour les éléments dans les collections, ce style est généralement utilisé pour *appeler une méthode* immédiatement pour tous les éléments d'une collection.

```
list.ForEach(item => item.DoSomething());

// or
list.ForEach(item => DoSomething(item));

// or using a method group
list.ForEach(Console.WriteLine);

// using an array
Array.ForEach(myArray, Console.WriteLine);
```

Il est important de noter que cette méthode est uniquement disponible sur les instances `List<T>` et comme méthode statique sur `Array` - elle **ne** fait **pas** partie de Linq.

Linq Parallel Foreach

Tout comme Linq Foreach, sauf que celui-ci fait le travail en parallèle. Cela signifie que tous les éléments de la collection exécuteront l'action donnée simultanément, simultanément.

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

Pause

Parfois, la condition de la boucle doit être vérifiée au milieu de la boucle. Le premier est sans doute plus élégant que le second:

```
for (;;)
{
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

Alternative:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
```

```
    }
}
```

Remarque: dans les boucles imbriquées et / ou le `switch` doit utiliser plus qu'une simple `break` .

Boucle Foreach

foreach itérera sur tout objet d'une classe qui implémente `IEnumerable` (notez que `IEnumerable<T>` hérite). De tels objets incluent des objets intégrés, mais ne se limitent pas à: `List<T>` , `T[]` (tableaux de tout type), `Dictionary<TKey, TSource>` , ainsi que des interfaces comme `IQueryable` et `ICollection` , etc.

syntaxe

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

remarques

1. Le type `ItemType` n'a pas besoin de correspondre au type précis des éléments, il doit simplement être assignable à partir du type des éléments
2. Au lieu de `ItemType` , `var` peut aussi être utilisé pour déduire le type d'éléments de l'énumerableObject en inspectant l'argument générique de l'implémentation `IEnumerable`
3. L'instruction peut être un bloc, une seule instruction ou même une instruction vide (`;`)
4. Si `enumerableObject` pas `IEnumerable` , le code ne sera pas compilé
5. Au cours de chaque itération, l'élément en cours est `ItemType` en `ItemType` (même s'il n'est pas spécifié, mais via le `var` via le compilateur) et si l'élément ne peut pas être `InvalidCastException` une `InvalidCastException` sera lancée.

Considérez cet exemple:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

est équivalent à:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
```

```
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

En boucle

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Sortie:

0
1
2
3
4

Les IEnumerators peuvent être itérés avec une boucle while:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Sortie de l'échantillon:

Tokyo / Yokohama
Métro de New York
Sao Paulo
Séoul / Incheon

Pour boucle

Une boucle For est idéale pour faire les choses un certain temps. C'est comme une boucle While mais l'incrément est inclus dans la condition.

Une boucle For est configurée comme ceci:

```
for (Initialization; Condition; Increment)
```

```
{
    // Code
}
```

Initialisation - Crée une nouvelle variable locale qui ne peut être utilisée que dans la boucle.
Condition - La boucle ne s'exécute que lorsque la condition est vraie.
Incrément - Comment la variable change à chaque exécution de la boucle.

Un exemple:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Sortie:

0
1
2
3
4

Vous pouvez également laisser des espaces dans la boucle For, mais vous devez avoir tous les points-virgules pour que cela fonctionne.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Sortie pour 3:

3
5
7
9
11

Boucle Do-While

Il est semblable à un `while` en boucle, à l' exception qu'il teste la condition à la *fin* du corps de la boucle. La boucle Do-While exécute la boucle une seule fois, que la condition soit vraie ou non.

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
```

```
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

Boucles imbriquées

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
}
```

continuer

En plus de `break` , il y a aussi le mot clé `continue` . Au lieu de casser complètement la boucle, il faudra simplement ignorer l'itération en cours. Cela pourrait être utile si vous ne voulez pas qu'un code soit exécuté si une valeur particulière est définie.

Voici un exemple simple:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

Aura pour résultat:

```
9
10
```

Remarque: `Continue` est souvent utile dans les boucles while ou do-while. Les boucles For, avec des conditions de sortie bien définies, pourraient ne pas en bénéficier autant.

Lire En boucle en ligne: https://riptutorial.com/fr/csharp/topic/2064/en-boucle

Chapitre 46: Enum

Introduction

Un enum peut dériver de l'un des types suivants: octet, sbyte, short, ushort, int, uint, long, ulong. La valeur par défaut est int et peut être modifiée en spécifiant le type dans la définition enum:

énumération publique Jour de la semaine: octet {lundi = 1, mardi = 2, mercredi = 3, jeudi = 4, vendredi = 5}

Ceci est utile lorsque P / Invoking au code natif, mappage aux sources de données et circonstances similaires. En général, le int par défaut doit être utilisé, car la plupart des développeurs s'attendent à ce qu'un énum soit un int.

Syntaxe

- Enum Colors {Rouge, Vert, Bleu} // Déclaration Enum
- enum Couleurs: octet {Rouge, Vert, Bleu} // Déclaration avec un type spécifique
- Enum Couleurs {Rouge = 23, Vert = 45, Bleu = 12} // Déclaration avec des valeurs définies
- Colors.Red // Accéder à un élément d'un Enum
- int value = (int) Colors.Red // Récupère la valeur int d'un élément enum
- Couleurs color = (Couleurs) intValue // Récupère un élément enum depuis int

Remarques

Un Enum (abréviation de "type énuméré") est un type constitué d'un ensemble de constantes nommées, représentées par un identificateur spécifique au type.

Les énumérations sont les plus utiles pour représenter des concepts ayant un nombre (généralement faible) de valeurs discrètes possibles. Par exemple, ils peuvent être utilisés pour représenter un jour de la semaine ou un mois de l'année. Ils peuvent également être utilisés comme indicateurs pouvant être combinés ou vérifiés à l'aide d'opérations binaires.

Examples

Obtenez toutes les valeurs de membres d'un enum

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

Cela va imprimer:

```
One
Two
Three
```

Enum comme drapeaux

Le `FlagsAttribute` peut être appliqué à un enum modifiant le comportement du `ToString()` pour correspondre à la nature de l'énumération:

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

Étant `FlagsAttribute` que `FlagsAttribute` s'appuie sur les constantes d'énumération pour avoir deux puissances (ou leurs combinaisons) et que les valeurs d'énumération sont finalement des valeurs numériques, vous êtes limité par la taille du type numérique sous-jacent. Le plus grand type numérique disponible que vous pouvez utiliser est `UInt64` , qui vous permet de spécifier 64 constantes enum distinctes (non combinées). Le mot clé `enum` utilise par défaut le type sous-jacent `int` , à savoir `Int32` . Le compilateur autorisera la déclaration de valeurs supérieures à 32 bits. Ceux-ci se dérouleront sans avertissement et aboutiront à deux ou plusieurs membres de la même valeur. Par conséquent, si un enum est destiné à accueillir un ensemble de bits de plus de 32 indicateurs, vous devez spécifier explicitement un type plus grand:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Bien que les drapeaux ne représentent souvent qu'un seul bit, ils peuvent être combinés en «ensembles» nommés pour une utilisation plus facile.

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
```

```
    Option3 = 4,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Pour éviter d'épeler les valeurs décimales des puissances de deux, l' opérateur de décalage à gauche (<<) peut également être utilisé pour déclarer le même enum

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

A partir de C # 7.0, les littéraux binaires peuvent aussi être utilisés.

Pour vérifier si la valeur de la variable enum a un certain drapeau, la méthode `HasFlag` peut être utilisée. Disons que nous avons

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

Et une `value`

```
var value = MyEnum.One | MyEnum.Two;
```

Avec `HasFlag` nous pouvons vérifier si l'un des indicateurs est défini

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
    Console.WriteLine("Enum has Two");

if(value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

Nous pouvons également parcourir toutes les valeurs de enum pour obtenir tous les indicateurs définis

```
var type = typeof(MyEnum);
```

```
var names = Enum.GetNames(type);

foreach (var name in names)
{
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

Ou

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

Les trois exemples vont imprimer:

```
Enum has One
Enum has Two
```

Tester les valeurs d'énumération de style avec la logique binaire

Une valeur d'énumération de style drapeaux doit être testée avec une logique bit par bit car elle ne peut correspondre à aucune valeur unique.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

La valeur `Default` est en fait une combinaison de deux autres *fusionnés* avec un OU au niveau du bit. Par conséquent, pour tester la présence d'un indicateur, nous devons utiliser un ET au niveau du bit.

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

Enum pour enchaîner et revenir

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}


// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"


// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET
versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive
match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)


// String to enum (all .NET versions)
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Thows
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

Valeur par défaut pour enum == ZERO

La valeur par défaut d'un enum est zéro . Si une énumération ne définit pas un élément avec une valeur de zéro, sa valeur par défaut sera zéro.

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);
```

```
        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
        else
            Console.WriteLine("Unknown");
    }
}
```

Exemple: https://dotnetfiddle.net/l5Rwie

Bases d'énum

De MSDN :

> Un type d'énumération (également appelé énumération ou énumération) fournit un moyen efficace de définir un ensemble de **constantes intégrales** nommées pouvant être **affectées à une variable** .

Essentiellement, un enum est un type qui n'autorise qu'un ensemble d'options finies et chaque option correspond à un nombre. Par défaut, ces nombres augmentent dans l'ordre dans lequel les valeurs sont déclarées, à partir de zéro. Par exemple, on pourrait déclarer une énumération pour les jours de la semaine:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Cette énumération pourrait être utilisée comme ceci:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

Par défaut, le type sous-jacent de chaque élément de l' `enum` est `int` , mais les `byte` , `byte` , `sbyte` `short` , `ushort` , `uint` , `long` et `ulong` peuvent également être utilisés. Si vous utilisez un type autre que `int` , vous devez spécifier le type en utilisant deux-points après le nom enum:

```
public enum Day : byte
{
```

```
    // same as before
}
```

Les nombres après le nom sont maintenant des octets au lieu d'entiers. Vous pouvez obtenir le type sous-jacent de l'énumération comme suit:

```
Enum.GetUnderlyingType(typeof(Days));
```

Sortie:

```
System.Byte
```

Démo: violon .NET

Manipulation binaire utilisant des énumérations

FlagsAttribute doit être utilisé chaque fois que l'énumérateur représente une collection d'indicateurs, plutôt qu'une valeur unique. La valeur numérique attribuée à chaque valeur enum aide lors de la manipulation des énumérations à l'aide d'opérateurs binaires.

Exemple 1: Avec [Drapeaux]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

imprime rouge, bleu

Exemple 2: sans [drapeaux]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

imprime 3

Utiliser la notation pour les drapeaux

L'opérateur de décalage à gauche (`<<`) peut être utilisé dans les déclarations d'énumération afin de s'assurer que chaque indicateur a exactement `1` en représentation binaire, comme le devraient les indicateurs.

Cela aide également à améliorer la lisibilité des grands enums avec beaucoup de drapeaux.

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

Il est maintenant évident que `MyEnum` contient que des indicateurs corrects et non des éléments désordonnés comme `Flag30 = 1073741822` (ou 11111111111111111111111111110 en binaire), ce qui est inapproprié.

Ajout d'informations de description supplémentaires à une valeur enum

Dans certains cas, vous pouvez vouloir ajouter une description supplémentaire à une valeur enum, par exemple lorsque la valeur enum elle-même est moins lisible que ce que vous pourriez souhaiter afficher pour l'utilisateur. Dans ce cas, vous pouvez utiliser la classe `System.ComponentModel.DescriptionAttribute` .

Par exemple:

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

Maintenant, si vous souhaitez renvoyer la description d'une valeur d'énumération spécifique, vous pouvez effectuer les opérations suivantes:

```
public static string GetDescriptionAttribute(PossibleResults result)
{
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
```

```
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}
```

Cela peut également être facilement transformé en une méthode d'extension pour toutes les énumérations:

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())
typeof(DescriptionAttribute))).Description;
    }
}
```

Et ensuite facilement utilisé comme ceci: `Console.WriteLine(result.GetDescription());`

Ajouter et supprimer des valeurs de l'énumération marquée

Ce code consiste à ajouter et à supprimer une valeur d'une instance d'énumération marquée:

```
[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// set additional value
value |= MyEnum.Flag2;  //value is now Flag1, Flag2
value |= MyEnum.Flag3;  //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

Enums peuvent avoir des valeurs inattendues

Comme un enum peut être converti en et à partir de son type intégral sous-jacent, la valeur peut tomber en dehors de la plage de valeurs donnée dans la définition du type enum.

Bien que le type `DaysOfWeek` ci-dessous `DaysOfWeek` que 7 valeurs définies, il peut toujours contenir une valeur `int` .

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
```

```
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOFWeek s = DaysOfWeek.Sunday;
s++; // No error
```

Il n'y a actuellement aucun moyen de définir une énumération qui n'a pas ce comportement.

Toutefois, les valeurs d'énumération non définies peuvent être détectées à l'aide de la méthode `Enum.IsDefined` . Par exemple,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

Lire Enum en ligne: https://riptutorial.com/fr/csharp/topic/931/enum

Chapitre 47: Est égal à et GetHashCode

Remarques

Chaque implémentation de `Equals` doit répondre aux exigences suivantes:

- **Réfléchissant** : Un objet doit être égal à lui-même.
  `x.Equals(x)` renvoie `true` .
- **Symétrique** : Il n'y a pas de différence si je compare x à y ou y à x - le résultat est le même.
  `x.Equals(y)` renvoie la même valeur que `y.Equals(x)` .
- **Transitif** : Si un objet est égal à un autre objet et que celui-ci est égal à un troisième, le premier doit être égal au troisième.
  if `(x.Equals(y) && y.Equals(z))` renvoie `true` , alors `x.Equals(z)` renvoie `true` .
- **Cohérent** : Si vous comparez un objet à plusieurs fois, le résultat est toujours le même. Les invocations successives de `x.Equals(y)` renvoient la même valeur tant que les objets référencés par x et y ne sont pas modifiés.
- **Comparaison à null** : Aucun objet n'est égal à `null` .
  `x.Equals(null)` renvoie `false` .

Implémentations de `GetHashCode` :

- **Compatible avec `Equals`** : Si deux objets sont égaux (ce qui signifie que `Equals` renvoie true), `GetHashCode` **doit** renvoyer la même valeur pour chacun d'eux.
- **Large range** : Si deux objets ne sont pas égaux (`Equals` dit false), il devrait y avoir une **forte probabilité que** leurs codes de hachage soient distincts. *Un* hachage *parfait* n'est souvent pas possible car il y a un nombre limité de valeurs à choisir.
- **Bon marché** : le calcul du code de hachage dans tous les cas devrait être peu coûteux.

Voir: Directives pour la surcharge d'Equals () et de l'opérateur ==

Examples

Comportement par défaut égal.

`Equals` est déclaré dans la classe `Object` elle-même.

```
public virtual bool Equals(Object obj);
```

Par défaut, `Equals` a le comportement suivant:

- Si l'instance est un type de référence, alors `Equals` renvoie true uniquement si les références

sont identiques.

- Si l'instance est un type de valeur, alors `Equals` renvoie true uniquement si le type et la valeur sont identiques.

- `string` est un cas particulier. Il se comporte comme un type de valeur.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

Écrire un bon remplacement GetHashCode

`GetHashCode` a des effets de performance majeurs sur Dictionary <> et HashTable.

Bonnes méthodes `GetHashCode`

- devrait avoir une distribution uniforme
    - chaque entier devrait avoir une chance à peu près égale de retourner pour une instance aléatoire
    - si votre méthode retourne le même entier (par exemple, la constante '999') pour chaque instance, vous aurez de mauvaises performances

- devrait être rapide
    - Ce ne sont PAS des hachages cryptographiques, où la lenteur est une caractéristique
    - plus votre fonction de hachage est lente, plus votre dictionnaire est lent
- doit renvoyer le même HashCode sur deux instances évaluées par `Equals` à true
    - S'ils ne le font pas (par exemple parce que `GetHashCode` renvoie un nombre aléatoire), les éléments ne peuvent pas être trouvés dans une `List` , un `Dictionary` ou un élément similaire.

Une bonne méthode pour implémenter `GetHashCode` consiste à utiliser un nombre premier comme valeur de départ et à ajouter les codes de hachage des champs du type multiplié par d'autres nombres premiers:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Seuls les champs utilisés dans la méthode `Equals` doivent être utilisés pour la fonction de hachage.

Si vous avez besoin de traiter le même type de différentes manières pour Dictionary / HashTables, vous pouvez utiliser IEqualityComparer.

Remplacer Equals et GetHashCode sur les types personnalisés

Pour une classe `Person` comme:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

Mais définir `Equals` et `GetHashCode` comme suit:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
```

```
    public string Clothes { get; set; }

    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

L'utilisation de LINQ pour effectuer des requêtes différentes sur des personnes vérifie également `Equals` et `GetHashCode` :

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

Est égal à et GetHashCode dans IEqualityComparator

Pour le type donné `Person` :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

Mais définir `Equals` et `GetHashCode` dans un `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
```

```
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when
comparing two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Notez que pour cette requête, deux objets ont été considérés égaux si les deux `Equals` renvoyés sont true et que `GetHashCode` a renvoyé le même code de hachage pour les deux personnes.

Lire Est égal à et GetHashCode en ligne: https://riptutorial.com/fr/csharp/topic/3429/est-egal-a-et-gethashcode

Chapitre 48: Événements

Introduction

Un événement est une notification indiquant que quelque chose s'est produit (tel qu'un clic de souris) ou, dans certains cas, est sur le point de se produire (par exemple, une modification de prix).

Les classes peuvent définir des événements et leurs instances (objets) peuvent déclencher ces événements. Par exemple, un bouton peut contenir un événement Click qui est déclenché lorsqu'un utilisateur a cliqué dessus.

Les gestionnaires d'événements sont alors des méthodes appelées lorsque leur événement correspondant est déclenché. Un formulaire peut contenir un gestionnaire d'événements Clicked pour chaque bouton qu'il contient, par exemple.

Paramètres

Paramètre	Détails
EventArgsT	Le type qui dérive de EventArgs et contient les paramètres de l'événement.
Nom de l'événement	Le nom de l'évènement.
HandlerName	Le nom du gestionnaire d'événements.
SenderObject	L'objet qui appelle l'événement.
EventArguments	Une instance du type EventArgsT qui contient les paramètres d'événement.

Remarques

En levant un événement:

- Vérifiez toujours si le délégué est `null` . Un délégué nul signifie que l'événement n'a pas d'abonnés. Augmenter un événement sans abonnés entraînera une `NullReferenceException` .

6,0

- Copiez le délégué (par exemple, `EventName`) dans une variable locale (par exemple, `eventName`) avant de rechercher la nullité / la `eventName` l'événement. Cela évite les conditions de concurrence dans les environnements multithread:

Faux :

```
    if(Changed != null)      // Changed has 1 subscriber at this point
                             // In another thread, that one subscriber decided to unsubscribe
        Changed(this, args); // `Changed` is now null, `NullReferenceException` is thrown.
```

À droite :

```
    // Cache the "Changed" event as a local. If it is not null, then use
    // the LOCAL variable (handler) to raise the event, NOT the event itself.
    var handler = Changed;
    if(handler != null)
        handler(this, args);
```

6,0

- Utilisez l'opérateur null-conditionnel (?.) Pour générer la méthode au lieu de vérifier par null le délégué des abonnés dans une instruction `if` : `EventName?.Invoke(SenderObject, new EventArgsT());`
- Lorsque vous utilisez Action <> pour déclarer des types de délégué, la signature du gestionnaire de méthode / événement anonyme doit être identique au type de délégué anonyme déclaré dans la déclaration d'événement.

Examples

Déclarer et soulever des événements

Déclarer un événement

Vous pouvez déclarer un événement sur une `class` ou une `struct` utilisant la syntaxe suivante:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

Il existe une syntaxe étendue pour la déclaration des événements, dans laquelle vous détenez une instance privée de l'événement et définissez une instance publique à l'aide de la fonction `add` et `set` accessors. La syntaxe est très similaire aux propriétés C #. Dans tous les cas, la syntaxe illustrée ci-dessus doit être préférée, car le compilateur émet du code pour garantir que plusieurs threads peuvent ajouter et supprimer en toute sécurité des gestionnaires d'événements pour l'événement de votre classe.

Élever l'événement

6,0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

6,0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;

    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);

}
```

Notez que les événements ne peuvent être déclenchés que par le type déclarant. Les clients ne peuvent s'inscrire / se désinscrire

Pour les versions C # antérieures à 6.0, où `EventName?.Invoke` n'est pas pris en charge, il est `EventName?.Invoke` d'attribuer l'événement à une variable temporaire avant l'appel, comme illustré dans l'exemple, qui garantit la sécurité des threads. code. Si vous ne le faites pas, une `NullReferenceException` peut être `NullReferenceException` dans certains cas où plusieurs threads utilisent la même instance d'objet. Dans C # 6.0, le compilateur émet un code similaire à celui montré dans l'exemple de code pour C # 6.

Déclaration d'événement standard

Déclaration d'événement:

```
public event EventHandler<EventArgsT> EventName;
```

Déclaration du gestionnaire d'événement:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

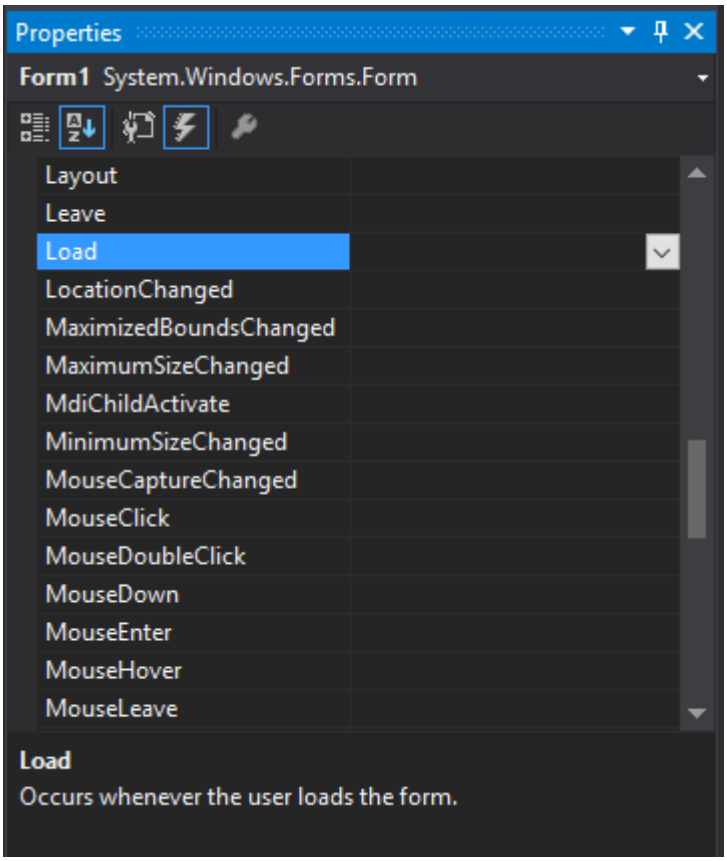
S'abonner à l'événement:

Dynamiquement:

```
EventName += HandlerName;
```

Par le concepteur:

1. Cliquez sur le bouton Événements dans la fenêtre des propriétés du contrôle (Éclair)
2. Double-cliquez sur le nom de l'événement:



3. Visual Studio générera le code de l'événement:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Invoquer la méthode:

```
EventName(SenderObject, EventArguments);
```

Déclaration de gestionnaire d'événement anonyme

Déclaration d'événement:

```
public event EventHandler<EventArgsType> EventName;
```

Déclaration du gestionnaire d'événement utilisant l' opérateur lambda => et s'abonnant à l'événement:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Déclaration du gestionnaire d'événement à l'aide de la syntaxe de la méthode de délégué anonyme:

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Déclaration et souscription d'un gestionnaire d'événement qui n'utilise pas le paramètre de l'événement et peut donc utiliser la syntaxe ci-dessus sans avoir à spécifier de paramètres:

```
EventName += delegate { /* Handler Logic */ }
```

Invoquer l'événement:

```
EventName?.Invoke(SenderObject, EventArguments);
```

Déclaration d'événement non standard

Les événements peuvent être de tout type délégué, pas seulement `EventHandler` et `EventHandler<T>` . Par exemple:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

Ceci est utilisé de manière similaire aux `EventHandler` standard:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

Il est possible de déclarer plusieurs événements du même type dans une seule instruction, similaire aux champs et aux variables locales (bien que cela puisse souvent être une mauvaise idée):

```
public event EventHandler Event1, Event2, Event3;
```

Cela déclare trois événements distincts (`Event1` , `Event2` et `Event3`) tous de type `EventHandler` . *Remarque: Bien que certains compilateurs puissent accepter cette syntaxe dans les interfaces ainsi que dans les classes, la spécification C # (v5.0 §13.2.3) fournit une grammaire pour les interfaces qui ne l'autorise pas.*

Création de EventArgs personnalisés contenant des données

supplémentaires

Les événements personnalisés nécessitent généralement des arguments d'événement personnalisés contenant des informations sur l'événement. Par exemple, `MouseEventArgs` qui est utilisé par les événements de la souris tels que les événements `MouseDown` ou `MouseUp` , contient des informations sur l' `Location` ou les `Buttons` utilisés pour générer l'événement.

Lors de la création de nouveaux événements, pour créer un argument d'événement personnalisé:

- Créez une classe dérivée de `EventArgs` et définissez les propriétés des données nécessaires.
- Par convention, le nom de la classe doit se terminer par `EventArgs` .

Exemple

Dans l'exemple ci-dessous, nous créons un événement `PriceChangingEventArgs` pour la propriété `Price` d'une classe. La classe de données d'événement contient un `CurrentPrice` et un `NewPrice` . L'événement se déclenche lorsque vous attribuez une nouvelle valeur à la propriété `Price` et informe le consommateur que la valeur change et lui permet de connaître le prix actuel et le nouveau prix:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Produit

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
```

```
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
}
```

Vous pouvez améliorer l'exemple en permettant au consommateur de modifier la nouvelle valeur, puis la valeur sera utilisée pour la propriété. Pour ce faire, il suffit d'appliquer ces changements de classes.

Changez la définition de `NewPrice` pour qu'elle soit `NewPrice` :

```
public int NewPrice { get; set; }
```

Modifiez la définition de `Price` pour utiliser `e.NewPrice` comme valeur de propriété, après avoir appelé `OnPriceChanging` :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

Créer un événement annulable

Un événement peut être annulé par une classe lorsqu'il est sur le point d'effectuer une action pouvant être annulée, telle que l'événement `FormClosing` d'un `Form` .

Pour créer un tel événement:

- Créez un nouvel argument d'événement dérivant de `CancelEventArgs` et ajoutez des propriétés supplémentaires pour les données d'événement.
- Créez un événement à l'aide de `EventHandler<T>` et utilisez la nouvelle classe d'argument event cancel que vous avez créée.

Exemple

Dans l'exemple ci-dessous, nous créons un événement `PriceChangingEventArgs` pour la propriété `Price` d'une classe. La classe de données d'événement contient une `Value` qui permet au consommateur de connaître le nouveau. L'événement se déclenche lorsque vous affectez une nouvelle valeur à la propriété `Price` et informe le consommateur que la valeur change et lui permet d'annuler l'événement. Si le consommateur annule l'événement, la valeur précédente de `Price` sera utilisée:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Produit

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

Propriétés de l'événement

Si une classe génère un grand nombre d'événements, le coût de stockage d'un champ par délégué peut ne pas être acceptable. Le .NET Framework fournit des propriétés d'événement pour ces cas. De cette façon, vous pouvez utiliser une autre structure de données comme `EventHandlerList` pour stocker les délégués d'événement:

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
```

```
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

Cette approche est largement utilisée dans les frameworks d'interface graphique comme WinForms, où les contrôles peuvent comporter des dizaines voire des centaines d'événements.

Notez que `EventHandlerList` n'est pas thread-safe, donc si vous prévoyez d'utiliser votre classe à partir de plusieurs threads, vous devrez ajouter des instructions de verrouillage ou un autre mécanisme de synchronisation (ou utiliser un stockage assurant la sécurité des threads).

Lire Événements en ligne: https://riptutorial.com/fr/csharp/topic/64/evenements

Chapitre 49: Exécution de requêtes HTTP

Examples

Création et envoi d'une requête HTTP POST

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

Création et envoi d'une requête HTTP GET

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
string responseBodyFromRemoteServer;
```

```
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

Gestion des erreurs de codes de réponse HTTP spécifiques (tels que 404 introuvable)

```
using System.Net;

...

string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as
WebExceptionStatus.ConnectFailure.
        // ...
    }
}
```

Envoi d'une requête HTTP POST asynchrone avec un corps JSON

```
public static async Task PostAsync(this Uri uri, object value)
{
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());

    using (var client = new HttpClient())
    {
        return await client.PostAsync(uri, content);
```

```
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

Envoi de requête HTTP GET asynchrone et lecture de requête JSON

```
public static async Task<TResult> GetAnsync<TResult>(this Uri uri)
{
    using (var client = new HttpClient())
    {
        var message = await client.GetAsync(uri);

        if (!message.IsSuccessStatusCode)
            throw new Exception();

        return message.ReadAsAsync<TResult>();
    }
}

. . .

public class Result
{
    public double foo { get; set; }

    public string bar { get; set; }
}

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
var result = await uri.GetAsync<Result>();
```

Récupérer le code HTML pour la page Web (simple)

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

Lire Exécution de requêtes HTTP en ligne: https://riptutorial.com/fr/csharp/topic/1971/execution-de-requetes-http

Chapitre 50: Expressions conditionnelles

Examples

Déclaration If-Else

La programmation en général nécessite souvent une `decision` ou une `branch` dans le code pour rendre compte de la manière dont le code fonctionne sous différentes entrées ou conditions. Dans le langage de programmation C # (et la plupart des langages de programmation en la matière), le moyen le plus simple et le plus utile de créer une branche dans votre programme est d'utiliser une instruction `If-Else` .

Supposons que nous ayons une méthode (alias une fonction) qui prend un paramètre int qui représentera un score allant jusqu'à 100, et la méthode imprimera un message indiquant si nous réussissons ou non.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

Lorsque vous examinez cette méthode, vous pouvez remarquer cette ligne de code (`score >= 50`) dans l'instruction `If` . Cela peut être vu comme une condition `boolean` , où si la condition est évaluée à égale à `true` , le code qui se trouve entre le `if { }` est exécuté.

Par exemple, si cette méthode a été appelée comme ceci: `PrintPassOrFail(60);` , la sortie de la méthode serait une impression de console disant ***pass!*** la valeur du paramètre de 60 étant supérieure ou égale à 50.

Cependant, si la méthode a été appelée comme: `PrintPassOrFail(30);` , la sortie de la méthode imprimera en disant ***Fail!*** . En effet, la valeur 30 n'est pas supérieure ou égale à 50, donc le code entre the `else { }` est exécuté à la place de l'instruction `If` .

Dans cet exemple, nous avons dit que le *score* devrait aller jusqu'à 100, ce qui n'a pas du tout été pris en compte. Pour prendre en compte le *score qui* ne dépasse pas 100, ou peut-être en dessous de 0, reportez-vous à l'exemple **If-Else If-Else Statement** .

Déclaration If-Else If-Else

Suite à l'exemple **If-Else Statement** , il est maintenant temps d'introduire l'instruction `Else If` . L'instruction `Else If` suit directement l'instruction `If` dans la structure **If-Else If-Else** , mais

possède intrinsèquement une syntaxe similaire à celle de l'instruction `If` . Il est utilisé pour ajouter plus de branches au code que ce qu'une simple déclaration **If-Else** peut faire.

Dans l'exemple de la **déclaration If-Else** , l'exemple spécifiait que le score montait à 100; Cependant, il n'y a jamais eu de vérification contre cela. Pour résoudre ce problème, modifions la méthode de l' **instruction If-Else** pour qu'elle ressemble à ceci:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

Toutes ces instructions seront exécutées dans l'ordre, du début à la fin, jusqu'à ce qu'une condition soit remplie. Dans cette nouvelle mise à jour de la méthode, nous avons ajouté deux nouvelles branches pour tenir compte du dépassement *des limites* .

Par exemple, si nous appelons maintenant la méthode dans notre code comme `PrintPassOFail(110);` , la sortie serait une impression de console disant ***Erreur: le score est supérieur à 100!*** ; et si nous avons appelé la méthode dans notre code comme `PrintPassOrFail(-20);` , la sortie dirait ***Erreur: le score est inférieur à 0!*** .

Changer les déclarations

Une instruction switch permet de tester l'égalité d'une variable par rapport à une liste de valeurs. Chaque valeur est appelée un cas et la variable activée est vérifiée pour chaque cas de commutation.

Une instruction `switch` est souvent plus concise et compréhensible que `if...else if... else..` lors du test de plusieurs valeurs possibles pour une seule variable.

La syntaxe est la suivante

```
switch(expression) {
   case constant-expression:
      statement(s);
      break;
   case constant-expression:
      statement(s);
```

```
      break;

   // you can have any number of case statements
   default : // Optional
      statement(s);
      break;
}
```

il y a plusieurs choses à considérer lors de l'utilisation de l'instruction switch

- L'expression utilisée dans une instruction switch doit avoir un type intégral ou énuméré, ou être d'un type de classe dans lequel la classe a une fonction de conversion unique pour un type intégral ou énuméré.
- Vous pouvez avoir un nombre quelconque de déclarations de cas dans un commutateur. Chaque cas est suivi de la valeur à comparer et d'un deux-points. Les valeurs à comparer doivent être uniques dans chaque instruction de commutateur.
- Une instruction switch peut avoir un cas par défaut facultatif. Le cas par défaut peut être utilisé pour effectuer une tâche lorsque aucun des cas n'est vrai.
- Chaque cas doit se terminer par une déclaration de `break` , sauf s'il s'agit d'une instruction vide. Dans ce cas, l'exécution se poursuivra au cas par cas. L'instruction break peut également être omise lorsque vous utilisez une `goto case` `return` , `throw` ou `goto case` .

Exemple peut être donné avec les notes sage

```
char grade = 'B';

switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent!");
        break;
    case 'B':
    case 'C':
        Console.WriteLine("Well done");
        break;
    case 'D':
        Console.WriteLine("You passed");
        break;
    case 'F':
        Console.WriteLine("Better try again");
        break;
    default:
        Console.WriteLine("Invalid grade");
        break;
}
```

Si les conditions de déclaration sont des expressions et des valeurs booléennes standard

La déclaration suivante

```
if (conditionA && conditionB && conditionC) //...
```

est exactement équivalent à

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

en d'autres termes, les conditions à l'intérieur de l'instruction "if" forment simplement une expression booléenne ordinaire.

Une erreur courante lors de l'écriture d'instructions conditionnelles est de comparer explicitement à `true` et `false` :

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

Cela peut être réécrit comme

```
if (conditionA && !conditionB && conditionC)
```

Lire Expressions conditionnelles en ligne: https://riptutorial.com/fr/csharp/topic/3144/expressions-conditionnelles

Chapitre 51: Expressions lambda

Remarques

Une expression lambda est une syntaxe permettant de créer des fonctions anonymes en ligne. Plus formellement, à partir du Guide de programmation C # :

> Une expression lambda est une fonction anonyme que vous pouvez utiliser pour créer des délégués ou des types d'arborescence d'expression. En utilisant des expressions lambda, vous pouvez écrire des fonctions locales pouvant être transmises en tant qu'arguments ou renvoyées en tant que valeur des appels de fonction.

Une expression lambda est créée à l'aide de l'opérateur `=>` . Mettez tous les paramètres sur le côté gauche de l'opérateur. Du côté droit, mettez une expression qui peut utiliser ces paramètres; cette expression résoudra la valeur de retour de la fonction. Plus rarement, si nécessaire, un `{code block}` complet peut être utilisé à droite. Si le type de retour n'est pas nul, le bloc contiendra une déclaration de retour.

Examples

Passer une expression Lambda en tant que paramètre à une méthode

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Ici `x => x > 6` est une expression lambda agissant comme un prédicat qui garantit que seuls les éléments supérieurs à 6 sont renvoyés.

Expressions Lambda comme raccourci pour l'initialisation des délégués

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

La syntaxe d'expression Lambda ci-dessus est équivalente au code explicite suivant:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

Lambdas pour `Func` et` Action`

Les lambdas sont généralement utilisés pour définir des *fonctions* simples (généralement dans le contexte d'une expression linq):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Ici, le `return` est implicite.

Cependant, il est également possible de passer des *actions en* tant que lambdas:

```
myObservable.Do(x => Console.WriteLine(x));
```

Expressions lambda avec plusieurs paramètres ou aucun paramètre

Utilisez des parenthèses autour de l'expression à gauche de l'opérateur `=>` pour indiquer plusieurs paramètres.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

De même, un ensemble vide de parenthèses indique que la fonction n'accepte pas les paramètres.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

Mettre plusieurs instructions dans une déclaration Lambda

Contrairement à une expression lambda, une instruction lambda peut contenir plusieurs instructions séparées par des points-virgules.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Notez que les instructions sont placées entre accolades `{}` .

N'oubliez pas que l'instruction lambdas ne peut pas être utilisée pour créer des arborescences d'expression.

Les Lambdas peuvent être émises à la fois comme `Func` et` Expression`

En supposant la classe `Person` suivante:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Le lambda suivant:

```
p => p.Age > 18
```

Peut être passé en argument aux deux méthodes:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Parce que le compilateur est capable de transformer des lambdas à la fois en délégués et en `Expression` s.

De toute évidence, les fournisseurs LINQ s'appuient fortement sur les `Expression` (exposées principalement via l'interface `IQueryable<T>`) pour pouvoir analyser les requêtes et les traduire en magasin.

Expression Lambda en tant que gestionnaire d'événements

Les expressions Lambda peuvent être utilisées pour gérer des événements, ce qui est utile lorsque:

- Le gestionnaire est court.
- Le gestionnaire n'a jamais besoin d'être désabonné.

Une bonne situation dans laquelle un gestionnaire d'événements lambda peut être utilisé est donnée ci-dessous:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

Si le désabonnement d'un gestionnaire d'événement enregistré à un moment ultérieur du code est nécessaire, l'expression du gestionnaire d'événement doit être enregistrée dans une variable et l'inscription / désinscription doit être effectuée via cette variable:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

La raison pour laquelle cela est fait plutôt que de simplement ressaisir l'expression lambda textuellement pour la désabonner (`-=`) est que le compilateur C # ne considérera pas nécessairement les deux expressions égales:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Notez que si des instructions supplémentaires sont ajoutées à l'expression lambda, les accolades environnantes requises peuvent être accidentellement omises, sans provoquer d'erreur de compilation. Par exemple:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

Cela compilera, mais entraînera l'ajout de l'expression lambda `(sender, args) => Console.WriteLine("Email sent");` en tant que gestionnaire d'événements et en exécutant l'instruction `emailSendButton.Enabled = true;` immédiatement. Pour corriger cela, le contenu du lambda doit être entouré d'accolades. Cela peut être évité en utilisant des accolades dès le début, en étant prudent lors de l'ajout d'instructions supplémentaires à un gestionnaire d'événements lambda, ou en entourant le lambda entre parenthèses dès le début:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Lire Expressions lambda en ligne: https://riptutorial.com/fr/csharp/topic/46/expressions-lambda

Chapitre 52: Expressions lambda

Remarques

Fermetures

Les expressions Lambda captureront implicitement les variables utilisées et créeront une fermeture . Une fermeture est une fonction associée à un contexte d'état. Le compilateur générera une fermeture chaque fois qu'une expression lambda «enfermera» une valeur de son contexte environnant.

Par exemple, lorsque ce qui suit est exécuté

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

`safeApplyFilterPredicate` fait référence à un objet nouvellement créé qui a une référence privée à la valeur actuelle du `filterer` et dont la méthode `Invoke` se comporte comme

```
o => (o != null) && filterer.Predicate(i);
```

Cela peut être important, car tant que la référence à la valeur maintenant dans `safeApplyFilterPredicate` est conservée, il y aura une référence à l'objet auquel le `filterer` fait actuellement référence. Cela a un effet sur la récupération de place et peut provoquer un comportement inattendu si l'objet auquel le `filterer` fait actuellement référence est muté.

D'un autre côté, les fermetures peuvent être utilisées pour limiter délibérément un comportement impliquant des références à d'autres objets.

Par exemple

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

Les fermetures peuvent également être utilisées pour modéliser des machines à états:

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

Examples

Expressions lambda de base

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;

// Behaviourally equivalent to:

int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

Expressions lambda de base avec LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Utiliser la syntaxe lambda pour créer une fermeture

Voir les remarques pour la discussion des fermetures. Supposons que nous ayons une interface:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

et puis ce qui suit est exécuté:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

Maintenant, `machineClosure` fait référence à une fonction de `int` à `int` , qui derrière les scènes utilise l'instance `IMachine` laquelle la `machine` fait référence pour effectuer le calcul. Même si la référence `machine` est hors de portée, dans la mesure où le `machineClosure` objet est maintenu,

l'original `IMachine` instance est conservé en tant que partie d'une « fermeture », définie automatiquement par le compilateur.

Attention: cela peut signifier que le même appel de fonction renvoie des valeurs différentes à des moments différents (par exemple, dans cet exemple, si la machine conserve une somme de ses entrées). Dans de nombreux cas, cela peut être inattendu et doit être évité pour tout code dans un style fonctionnel - les fermetures accidentelles et inattendues peuvent être une source de bogues.

Syntaxe Lambda avec corps de bloc d'instructions

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Expressions lambda avec System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Lire Expressions lambda en ligne: https://riptutorial.com/fr/csharp/topic/7057/expressions-lambda

Chapitre 53: Extensions Réactives (Rx)

Examples

Observation de l'événement TextChanged sur une zone de texte

Une observable est créée à partir de l'événement TextChanged de la zone de texte. De plus, toute entrée est sélectionnée uniquement si elle est différente de la dernière entrée et si aucune entrée n'a été effectuée dans les 0,5 secondes. La sortie dans cet exemple est envoyée à la console.

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

Diffusion de données à partir d'une base de données avec Observable

Supposons qu'une méthode retourne `IEnumerable<T>` , fe

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Crée une observable et démarre une méthode de manière asynchrone. `SelectMany` aplatit la collection et l'abonnement est déclenché tous les 200 éléments via `Buffer` .

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

Lire Extensions Réactives (Rx) en ligne: https://riptutorial.com/fr/csharp/topic/5770/extensions-reactives--rx-

Chapitre 54: Fichier et flux I / O

Introduction

Gère les fichiers.

Syntaxe

- `new System.IO.StreamWriter(string path)`
- `new System.IO.StreamWriter(string path, bool append)`
- `System.IO.StreamWriter.WriteLine(string text)`
- `System.IO.StreamWriter.WriteAsync(string text)`
- `System.IO.Stream.Close()`
- `System.IO.File.ReadAllText(string path)`
- `System.IO.File.ReadAllLines(string path)`
- `System.IO.File.ReadLines(string path)`
- `System.IO.File.WriteAllText(string path, string text)`
- `System.IO.File.WriteAllLines(string path, IEnumerable<string> contents)`
- `System.IO.File.Copy(string source, string dest)`
- `System.IO.File.Create(string path)`
- `System.IO.File.Delete(string path)`
- `System.IO.File.Move(string source, string dest)`
- `System.IO.Directory.GetFiles(string path)`

Paramètres

Paramètre	Détails
chemin	L'emplacement du fichier.
ajouter	Si le fichier existe, true ajoutera des données à la fin du fichier (append), false remplacera le fichier.
texte	Texte à écrire ou à stocker.
Contenu	Une collection de chaînes à écrire.
la source	L'emplacement du fichier que vous souhaitez utiliser.
dest	L'emplacement auquel vous voulez qu'un fichier accède.

Remarques

- Veillez toujours à fermer les objets `Stream` . Cela peut être fait avec un en `using` le bloc comme indiqué ci - dessus ou en appelant manuellement `myStream.Close()` .
- Assurez-vous que l'utilisateur actuel dispose des autorisations nécessaires sur le chemin

que vous essayez de créer.
- Les chaînes verbatim doivent être utilisées lors de la déclaration d'une chaîne de chemin comprenant des barres obliques inverses, comme ceci: `@"C:\MyFolder\MyFile.txt"`

Examples

Lecture d'un fichier à l'aide de la classe System.IO.File

Vous pouvez utiliser la fonction **System.IO.File.ReadAllText** pour lire le contenu entier d'un fichier dans une chaîne.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

Vous pouvez également lire un fichier sous la forme d'un tableau de lignes à l'aide de la fonction **System.IO.File.ReadAllLines** :

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Écriture de lignes dans un fichier à l'aide de la classe System.IO.StreamWriter

La classe **System.IO.StreamWriter** :

> Implémente un TextWriter pour écrire des caractères dans un flux dans un codage particulier.

En utilisant la méthode `WriteLine` , vous pouvez écrire du contenu ligne par ligne dans un fichier.

Notez l'utilisation du mot-clé `using` qui garantit que l'objet StreamWriter est éliminé dès qu'il est hors de portée et donc le fichier est fermé.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Notez que StreamWriter peut recevoir un second paramètre `bool` dans son constructeur, ce qui permet d' `Append` un fichier au lieu de remplacer le fichier:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

Écriture dans un fichier à l'aide de la classe System.IO.File

Vous pouvez utiliser la fonction **System.IO.File.WriteAllText** pour écrire une chaîne dans un fichier.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

Vous pouvez également utiliser la fonction **System.IO.File.WriteAllLines** qui reçoit un `IEnumerable<String>` comme second paramètre (par opposition à une seule chaîne dans l'exemple précédent). Cela vous permet d'écrire du contenu à partir d'un tableau de lignes.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

Lentement lire un fichier ligne par ligne via un IEnumerable

Lorsque vous travaillez avec des fichiers volumineux, vous pouvez utiliser la méthode `System.IO.File.ReadLines` pour lire toutes les lignes d'un fichier dans une `IEnumerable<string>` . Ceci est similaire à `System.IO.File.ReadAllLines` , sauf qu'il ne charge pas tout le fichier en mémoire en même temps, ce qui le rend plus efficace lorsque vous travaillez avec des fichiers volumineux.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

Le deuxième paramètre de File.ReadLines est facultatif. Vous pouvez l'utiliser lorsque cela est nécessaire pour spécifier l'encodage.

Il est important de noter qu'appeler `ToArray` , `ToList` ou une autre fonction similaire forcera toutes les lignes à être chargées en une fois, ce qui signifie que les avantages de l'utilisation de `ReadLines` sont annulés. Il est préférable d'énumérer sur `IEnumerable` utilisant une boucle `foreach` ou LINQ si vous utilisez cette méthode.

Créer un fichier

Classe statique de fichier

En utilisant la méthode `Create` de la classe statique `File` , nous pouvons créer des fichiers. Méthode crée le fichier sur le chemin donné, en même temps, il ouvre le fichier et nous donne le `FileStream` du fichier. Assurez-vous de fermer le fichier lorsque vous en avez terminé.

ex1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2:

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// you can write to the fileStream1
}
```

ex3:

```
File.Create("samplePath").Close();
```

Classe FileStream

Il y a beaucoup de surcharges de ce constructeur de classes, ce qui est bien documenté ici . L'exemple ci-dessous concerne celui qui couvre les fonctionnalités les plus utilisées de cette classe.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

Vous pouvez vérifier les énumérations pour FileMode , FileAccess et FileShare à partir de ces liens. Ce qu'ils signifient essentiellement sont les suivants:

FileMode: Réponses "Le fichier doit-il être créé? Ouvert? Créer s'il n'existe pas alors ouvert?" un peu de questions.

FileAccess: Réponses "Dois-je pouvoir lire le fichier, écrire dans le fichier ou les deux?" un peu de questions.

FileShare: Réponses "Les autres utilisateurs devraient-ils pouvoir lire, écrire, etc. sur le fichier alors que je l'utilise simultanément?" un peu de questions.

Copier un fichier

Classe statique de fichier

`File` classe statique de `File` peut être facilement utilisée à cette fin.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remarque: Par cette méthode, le fichier est copié, ce qui signifie qu'il sera lu à partir de la source, puis écrit dans le chemin de destination. Ce processus consomme beaucoup de ressources, sa durée est relativement longue et votre programme peut se bloquer si vous n'utilisez pas de threads.

Déplacer un fichier

Classe statique de fichier

La classe statique de fichier peut facilement être utilisée à cette fin.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remarque 1: Change uniquement l'index du fichier (si le fichier est déplacé dans le même volume). Cette opération ne prend pas de temps relatif à la taille du fichier.

Remarque 2: Impossible de remplacer un fichier existant sur le chemin de destination.

Supprimer le fichier

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

Bien que `Delete` ne `Delete` pas d'exception si le fichier n'existe pas, il lancera une exception, par exemple si le chemin spécifié n'est pas valide ou si l'appelant ne dispose pas des autorisations requises. Vous devez toujours placer les appels à `Delete` dans le bloc try-catch et gérer toutes les exceptions attendues. En cas de conditions de concurrence possibles, encapsulez la logique dans la déclaration de verrouillage .

Fichiers et répertoires

Obtenir tous les fichiers dans le répertoire

```
 var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Retourne un tableau de `FileInfo` , représentant tous les fichiers du répertoire spécifié.

Obtenir des fichiers avec une extension spécifique

```
 var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Retourne un tableau de `FileInfo` , représentant tous les fichiers du répertoire spécifié avec l'extension spécifiée.

Async écrire du texte dans un fichier à l'aide de StreamWriter

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
{
   // Can write either a string or char array
   await file.WriteAsync(text);
}
```

Lire Fichier et flux I / O en ligne: https://riptutorial.com/fr/csharp/topic/4266/fichier-et-flux-i---o

Chapitre 55: FileSystemWatcher

Syntaxe

- public FileSystemWatcher ()
- FileSystemWatcher public (chemin de chaîne)
- public FileSystemWatcher (chemin de chaîne, filtre de chaîne)

Paramètres

chemin	filtre
Répertoire à surveiller, en notation UNC (Universal Naming Convention) ou standard.	Le type de fichiers à regarder Par exemple, "* .txt" surveille les modifications apportées à tous les fichiers texte.

Examples

FileWatcher de base

L'exemple suivant crée un `FileSystemWatcher` pour surveiller le répertoire spécifié au moment de l'exécution. Le composant est configuré pour surveiller les modifications de l'heure **LastWrite** et **LastAccess** , la création, la suppression ou le renommage de fichiers texte dans le répertoire. Si un fichier est modifié, créé ou supprimé, le chemin d'accès au fichier s'imprime sur la console. Lorsqu'un fichier est renommé, l'ancien et le nouveau chemin sont imprimés sur la console.

Utilisez les espaces de noms System.Diagnostics et System.IO pour cet exemple.

```
FileSystemWatcher watcher;

private void watch()
{
  // Create a new FileSystemWatcher and set its properties.
  watcher = new FileSystemWatcher();
  watcher.Path = path;

 /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
  watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                       | NotifyFilters.FileName | NotifyFilters.DirectoryName;

  // Only watch text files.
  watcher.Filter = "*.txt*";

  // Add event handler.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  // Begin watching.
  watcher.EnableRaisingEvents = true;
```

```
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
  //Copies file to another directory or another action.
  Console.WriteLine("File: " +  e.FullPath + " " + e.ChangeType);
}
```

IsFileReady

Une erreur commune commise par beaucoup de gens avec FileSystemWatcher ne tient pas compte du fait que l'événement FileWatcher est déclenché dès que le fichier est créé. Cependant, le traitement du fichier peut prendre un certain temps.

Exemple :

Prenez une taille de fichier de 1 Go par exemple. Le fichier apr demandé est créé par un autre programme (Explorer.exe en le copiant à partir de quelque part) mais cela prendra quelques minutes pour terminer ce processus. L'événement est déclenché au moment de la création et vous devez attendre que le fichier soit prêt à être copié.

Ceci est une méthode pour vérifier si le fichier est prêt.

```
 public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }

        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

Lire FileSystemWatcher en ligne: https://riptutorial.com/fr/csharp/topic/5061/filesystemwatcher

Chapitre 56: Filetage

Remarques

Un **thread** fait partie d'un programme qui peut s'exécuter indépendamment des autres parties. Il peut effectuer des tâches simultanément avec d'autres threads. **Le multithreading** est une fonctionnalité qui permet aux programmes d'exécuter un traitement simultané afin que plusieurs opérations puissent être effectuées à la fois.

Par exemple, vous pouvez utiliser le threading pour mettre à jour un compteur ou un compteur en arrière-plan tout en exécutant simultanément d'autres tâches au premier plan.

Les applications multithread sont plus sensibles aux entrées des utilisateurs et sont également facilement évolutives, car le développeur peut ajouter des threads au fur et à mesure que la charge de travail augmente.

Par défaut, un programme C # a un thread - le thread du programme principal. Cependant, des threads secondaires peuvent être créés et utilisés pour exécuter du code en parallèle avec le thread principal. Ces threads sont appelés threads de travail.

Pour contrôler le fonctionnement d'un thread, le CLR délègue une fonction au système d'exploitation appelé Thread Scheduler. Un planificateur de threads garantit que tous les threads reçoivent un temps d'exécution correct. Il vérifie également que les threads bloqués ou verrouillés ne consomment pas une grande partie du temps processeur.

L'espace de noms .NET Framework `System.Threading` facilite l'utilisation des threads. System.Threading active le multithreading en fournissant un certain nombre de classes et d'interfaces. En plus de fournir des types et des classes pour un thread particulier, il définit également des types pour contenir une collection de threads, une classe de temporisation, etc. Il fournit également son support en permettant un accès synchronisé aux données partagées.

`Thread` est la classe principale de l'espace de noms `System.Threading` . Les autres classes incluent `AutoResetEvent` , `Interlocked` , `Monitor` , `Mutex` et `ThreadPool` .

Certains des délégués présents dans l'espace de noms `System.Threading` incluent `ThreadStart` , `TimerCallback` et `WaitCallback` .

Les énumérations dans l'espace de noms `System.Threading` incluent `ThreadPriority` , `ThreadState` et `EventResetMode` .

Dans .NET Framework 4 et les versions ultérieures, la programmation multithread est simplifiée et simplifiée grâce aux classes `System.Threading.Tasks.Parallel` et `System.Threading.Tasks.Task` , Parallel LINQ (PLINQ), nouvelles classes de collections simultanées dans `System.Collections.Concurrent` noms `System.Collections.Concurrent` et un nouveau modèle de programmation basé sur les tâches.

Examples

Démonstration simple et complète

```
class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Démo de threading simple et complète à l'aide de tâches

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }
```

```
    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Parallélisation des tâches explicites

```
    private static void explicitTaskParallism()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
        Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

        // Start the task.
        taskA.Start();
        taskB.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);
        taskA.Wait();
        taskB.Wait();
        Console.Read();
    }
```

Parallélisme implicite des tâches

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
            () => b.DoSomeOtherWork()
            );

      }
```

Créer et démarrer un deuxième thread

Si vous effectuez plusieurs longs calculs, vous pouvez les exécuter simultanément sur différents threads de votre ordinateur. Pour ce faire, nous créons un nouveau **fil de discussion** et le pointons vers une méthode différente.

```
using System.Threading;
```

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

Commencer un thread avec des paramètres

en utilisant System.Threading;

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

Création d'un thread par processeur

> `Environment.ProcessorCount` Obtient le nombre de processeurs **logiques** sur la machine en cours.

Le CLR planifiera ensuite chaque thread sur un processeur logique, ce qui pourrait théoriquement signifier chaque thread sur un processeur logique différent, tous les threads sur un seul processeur logique ou une autre combinaison.

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }

    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

Éviter de lire et d'écrire des données simultanément

Parfois, vous voulez que vos threads partagent des données simultanément. Lorsque cela se produit, il est important de connaître le code et de verrouiller toutes les pièces susceptibles de tomber en panne. Un exemple simple de comptage de deux threads est présenté ci-dessous.

Voici un code dangereux (incorrect):

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

Vous remarquerez, au lieu de compter 1,2,3,4,5 ... nous comptons 1,1,2,2,3 ...

Pour résoudre ce problème, nous devons **verrouiller** la valeur de count afin que plusieurs threads différents ne puissent pas lire et écrire en même temps. Avec l'ajout d'un verrou et d'une clé, nous pouvons empêcher les threads d'accéder aux données simultanément.

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
```

```
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value
of count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}
```

Parallel.ForEach Loop

Si vous souhaitez accélérer une boucle foreach et quel que soit l'ordre dans lequel la sortie se déroule, vous pouvez la convertir en une boucle parallèle foreach en procédant comme suit:

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

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

Deadlocks (deux threads en attente sur l'autre)

Une impasse est ce qui se produit lorsque deux ou plusieurs threads attendent que chacun se termine ou libère une ressource de telle manière qu'ils attendent pour toujours.

Un scénario typique de deux threads en attente d'exécution l'un de l'autre est lorsqu'un thread d'interface graphique Windows Forms attend un thread de travail et le thread de travail tente d'appeler un objet géré par le thread d'interface graphique. Observez qu'avec ce code, cliquer sur button1 provoquera le blocage du programme.

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

`workerthread.Join()` est un appel qui bloque le thread d'appel jusqu'à la fin de workerthread. `textBox1.Invoke(invoke_delegate)` est un appel qui bloque le thread d'appel jusqu'à ce que le thread GUI ait traité invoke_delegate, mais cet appel provoque des blocages si le thread d'interface graphique attend déjà la fin du thread appelant.

Pour contourner ce problème, il est possible d'utiliser un moyen non bloquant d'appeler la zone de texte à la place:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

Toutefois, cela entraînera des problèmes si vous devez exécuter du code qui dépend de la zone de texte mise à jour en premier. Dans ce cas, exécutez cela dans le cadre de l'appel, mais sachez que cela le fera fonctionner sur le thread d'interface graphique.

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}
```

Vous pouvez également démarrer un tout nouveau thread et laisser celui-ci faire l'attente sur le thread d'interface graphique, afin que workerthread puisse se terminer.

```
private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}
```

Pour minimiser le risque de tomber dans une impasse d'attente mutuelle, évitez toujours les références circulaires entre les threads lorsque cela est possible. Une hiérarchie de threads où les threads de moindre rang ne laissent de messages que pour des threads de rang supérieur et ne les attendent jamais ne se heurteront pas à ce type de problème. Cependant, il serait toujours vulnérable aux blocages basés sur le verrouillage des ressources.

Deadlocks (maintenir la ressource et attendre)

Une impasse est ce qui se produit lorsque deux ou plusieurs threads attendent que chacun se termine ou libère une ressource de telle manière qu'ils attendent pour toujours.

Si thread1 maintient un verrou sur la ressource A et attend la libération de la ressource B alors que thread2 contient la ressource B et attend la libération de la ressource A, elles sont bloquées.

En cliquant sur le bouton1 pour l'exemple de code suivant, votre application entre en état d'interblocage et se bloque

```
private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
```

```
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Pour éviter d'être bloqué de cette façon, on peut utiliser Monitor.TryEnter (lock_object, timeout_in_milliseconds) pour vérifier si un verrou est déjà contenu sur un objet. Si Monitor.TryEnter ne parvient pas à acquérir un verrou sur lock_object avant timeout_in_milliseconds, il retourne false, ce qui donne au thread la possibilité de libérer d'autres ressources et de générer des rendements, donnant ainsi la possibilité aux autres threads de se terminer comme dans cette version légèrement modifiée. :

```
private void button_Click(object sender, EventArgs e)
{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;
```

```
    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        bool mustDoWork = true;
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Notez que cette solution repose sur le fait que thread2 ne veut pas que ses verrous et thread1 soient prêts à céder, de sorte que thread2 a toujours la priorité. Notez également que thread1 doit refaire le travail qu'il a fait après avoir verrouillé la ressource A, quand il cède. Par conséquent, soyez prudent lorsque vous implémentez cette approche avec plusieurs threads, car vous courez le risque d’entrer dans ce que l’on appelle un livelock - un état qui se produirait si deux threads continuaient à faire le premier bit de leur travail , recommençant à plusieurs reprises.

Lire Filetage en ligne: https://riptutorial.com/fr/csharp/topic/51/filetage

Chapitre 57: Filtres d'action

Examples

Filtres d'action personnalisés

Nous écrivons des filtres d'action personnalisés pour diverses raisons. Nous pouvons avoir un filtre d'action personnalisé pour la journalisation ou pour enregistrer des données dans la base de données avant toute exécution d'action. Nous pourrions également en avoir un pour extraire des données de la base de données et les définir comme valeurs globales de l'application.

Pour créer un filtre d'action personnalisé, vous devez effectuer les tâches suivantes:

1. Créer une classe
2. Héritez-le de la classe ActionFilterAttribute

Remplacez au moins l'une des méthodes suivantes:

OnActionExecuting - Cette méthode est appelée avant l'exécution d'une action de contrôleur.

OnActionExecuted - Cette méthode est appelée après l'exécution d'une action de contrôleur.

OnResultExecuting - Cette méthode est appelée avant l'exécution d'un résultat d'action de contrôleur.

OnResultExecuted - Cette méthode est appelée après l'exécution d'un résultat d'action de contrôleur.

Le filtre peut être créé comme indiqué dans la liste ci-dessous:

```
using System;

using System.Diagnostics;

using System.Web.Mvc;



namespace WebApplication1
{

    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }

        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
```

```
                var controllerName = filterContext.RouteData.Values["controller"];
                var actionName = filterContext.RouteData.Values["action"];
                var message = String.Format("{0} controller:{1} action:{2}",
"onactionexecuting", controllerName, actionName);
                Debug.WriteLine(message, "Action Filter Log");
                base.OnActionExecuting(filterContext);
            }
        }
    }
```

Lire Filtres d'action en ligne: https://riptutorial.com/fr/csharp/topic/1505/filtres-d-action

Chapitre 58: Fonction avec plusieurs valeurs de retour

Remarques

Il n'y a pas de réponse inhérente en C # à ce - dit - besoin. Néanmoins, il existe des solutions de rechange pour satisfaire ce besoin.

La raison pour laquelle je qualifie le besoin de "soi-disant" est que nous avons seulement besoin de méthodes avec 2 ou plus de 2 valeurs lorsque nous violons les principes de programmation corrects. Surtout le principe de la responsabilité unique .

Par conséquent, il serait préférable d'être alerté lorsque nous avons besoin de fonctions renvoyant 2 valeurs ou plus et améliorant notre conception.

Examples

"objet anonyme" + solution "mot clé dynamique"

Vous pouvez retourner un objet anonyme à partir de votre fonction

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

Et attribuez le résultat à un objet dynamique et lisez-y les valeurs.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

Solution tuple

Vous pouvez retourner une instance de la classe `Tuple` partir de votre fonction avec deux paramètres de modèle tels que `Tuple<string, MyClass>` :

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

Et lisez les valeurs comme ci-dessous:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

Paramètres Ref et Out

Le mot-clé `ref` est utilisé pour transmettre un argument en tant que référence . `out` fera la même chose que `ref` mais il ne nécessite pas de valeur assignée par l'appelant avant d'appeler la fonction.

Paramètre de référence : -Si vous voulez passer une variable en tant que paramètre ref, vous devez l'initialiser avant de le transmettre en tant que paramètre ref à la méthode.

Paramètre Out: - Si vous voulez passer une variable avec le paramètre out, vous n'avez pas besoin de l'initialiser avant de le passer comme paramètre out à la méthode.

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int
b, out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

Lire Fonction avec plusieurs valeurs de retour en ligne:
https://riptutorial.com/fr/csharp/topic/3908/fonction-avec-plusieurs-valeurs-de-retour

Chapitre 59: Fonctionnalités C # 3.0

Remarques

C # version 3.0 a été publié dans le cadre de .Net version 3.5. De nombreuses fonctionnalités ajoutées à cette version étaient compatibles avec LINQ (Language INtegrated Queries).

Liste des fonctionnalités ajoutées:

- LINQ
- Expressions lambda
- Méthodes d'extension
- Types anonymes
- Variables typées implicitement
- Initialiseurs d'objets et de collections
- Propriétés implémentées automatiquement
- Arbres d'expression

Examples

Variables typées implicitement (var)

Le mot-clé `var` permet à un programmeur de taper implicitement une variable au moment de la compilation. `var` déclarations `var` ont le même type que les variables explicitement déclarées.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

Les types des variables ci-dessus sont respectivement `int` , `double` , `StringBuilder` et un type anonyme.

Il est important de noter qu'une variable `var` n'est pas typée dynamiquement. `SquaredNumber = Builder` n'est pas valide puisque vous essayez de définir un `int` sur une instance de `StringBuilder`

Requêtes linguistiques intégrées (LINQ)

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IEnumerable<int> query = from x in array
```

```
                        where x % 2 == 1
                        orderby x descending
                        select x * x;
// Result: 49, 25, 1
```

Exemple d'article de Wikipédia sur C # 3.0, sous-section LINQ

L'exemple 1 utilise une syntaxe de requête conçue pour ressembler aux requêtes SQL.

```
//Example 2
IEnumerable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

Exemple d'article de Wikipédia sur C # 3.0, sous-section LINQ

L'exemple 2 utilise la syntaxe de la méthode pour obtenir le même résultat que l'exemple 1.

Il est important de noter que, en C #, la syntaxe de requête LINQ est un sucre syntaxique pour la syntaxe de la méthode LINQ. Le compilateur traduit les requêtes en appels de méthode au moment de la compilation. Certaines requêtes doivent être exprimées dans la syntaxe de la méthode. De MSDN - "Par exemple, vous devez utiliser un appel de méthode pour exprimer une requête qui récupère le nombre d'éléments correspondant à une condition spécifiée."

Expression Lambda

Lambda Expresions est une extension de méthodes anonymes qui permettent des paramètres et des valeurs de retour implicitement typés. Leur syntaxe est moins verbeuse que les méthodes anonymes et suit un style de programmation fonctionnel.

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

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

Le code ci-dessus affichera la somme des carrés des nombres 1 à 10 sur la console.

La première expression lambda place les nombres dans la liste. Comme il n'y a que 1 paramètre, la parenthèse peut être omise. Vous pouvez inclure des parenthèses si vous le souhaitez:

```
.Select( (number) => number * number);
```

ou tapez explicitement le paramètre mais les parenthèses sont nécessaires:

```
.Select( (int number) => number * number);
```

Le corps lambda est une expression et a un retour implicite. Vous pouvez également utiliser un corps de déclaration si vous le souhaitez. Ceci est utile pour les lambda plus complexes.

```
.Select( number => { return number * number; } );
```

La méthode select renvoie un nouveau IEnumerable avec les valeurs calculées.

La seconde expression lambda additionne les nombres dans la liste renvoyés par la méthode select. Les parenthèses sont nécessaires car il y a plusieurs paramètres. Les types de paramètres sont explicitement tapés, mais cela n'est pas nécessaire. La méthode ci-dessous est équivalente.

```
.Aggregate( (first, second) => { return first + second; } );
```

Comme celui-ci:

```
.Aggregate( (int first, int second) => first + second );
```

Types anonymes

Les types anonymes offrent un moyen pratique d'encapsuler un ensemble de propriétés en lecture seule dans un objet unique sans avoir à définir explicitement un type en premier. Le nom de 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 pouvez créer des types anonymes en utilisant le `new` mot-clé suivi d'une accolade *(`{`)* . À l'intérieur des accolades, vous pouvez définir des propriétés comme sur le code ci-dessous.

```
var v = new { Amount = 108, Message = "Hello" };
```

Il est également possible de créer un tableau de types anonymes. Voir le code ci-dessous:

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

Ou utilisez-le avec les requêtes LINQ:

```
var productQuery = from prod in products
```

```
                    select new { prod.Color, prod.Price };
```

Lire Fonctionnalités C # 3.0 en ligne: https://riptutorial.com/fr/csharp/topic/3820/fonctionnalites-c-sharp-3-0

Chapitre 60: Fonctionnalités C # 5.0

Syntaxe

- ***Async & Attente***
- **tâche** publique MyTask **Async** () {doSomething (); }

  attendez MyTaskAsync ();
- public **Task** `<string>` MyStringTask **Async** () {return getSomeString (); }

  chaîne MyString = attendre MyStringTaskAsync ();
- ***Informations sur l'appelant***
- public void MyCallerAttributes (chaîne MyMessage,

  [CallerMemberName] string MemberName = "",

  [CallerFilePath] string SourceFilePath = "",

  [CallerLineNumber] int LineNumber = 0)
- Trace.WriteLine ("Mon message:" + MyMessage);

  Trace.WriteLine ("Member:" + MemberName);

  Trace.WriteLine ("Chemin du fichier source:" + SourceFilePath);

  Trace.WriteLine ("Numéro de ligne:" + Numéro de ligne);

Paramètres

Méthode / Modificateur avec paramètre	Détails
`Type<T>`	`T` est le type de retour

Remarques

C # 5.0 est couplé à Visual Studio .NET 2012

Examples

Async & Attente

`async` et `await` sont deux opérateurs destinés à améliorer les performances en libérant des threads et en attendant que les opérations se terminent avant de continuer.

Voici un exemple pour obtenir une chaîne avant de renvoyer sa longueur:

```
//This method is async because:
//1. It has async and Task or Task<T> as modifiers
//2. It ends in "Async"
async Task<int> GetStringLengthAsync(string URL){
    HttpClient client = new HttpClient();
    //Sends a GET request and returns the response body as a string
    Task<string> getString = client.GetStringAsync(URL);
    //Waits for getString to complete before returning its length
    string contents = await getString;
    return contents.Length;
}

private async void doProcess(){
    int length = await GetStringLengthAsync("http://example.com/");
    //Waits for all the above to finish before printing the number
    Console.WriteLine(length);
}
```

Voici un autre exemple de téléchargement d'un fichier et de traitement de ce qui se passe lorsque ses progrès ont changé et lorsque le téléchargement est terminé (il y a deux façons de le faire):

Méthode 1:

```
//This one using async event handlers, but not async coupled with await
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);
    //Download the file asynchronously
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
}

//event called for when download progress has changed
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    //example code
    int i = 0;
    i++;
    doSomething();
}

//event called for when download has finished
private void FileCompleted(object sender, AsyncCompletedEventArgs e){
    Console.WriteLine("Completed!")
}
```

Méthode 2:

```
//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
```

```
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first
example
}
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    int i = 0;
    i++;
    doSomething();
}
private void doProcess(){
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}
```

Informations sur l'appelant

Les CIA sont un moyen simple d'obtenir des attributs de la manière ciblée. Il n'y a vraiment qu'une seule façon de les utiliser et il n'y a que 3 attributs.

Exemple:

```
//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}
```

Exemple de sortie:

```
//Message: Show my attributes.
//Member: doProcess
//Source File Path: c:\Path\To\The\File
//Line Number: 13
```

Lire Fonctionnalités C # 5.0 en ligne: https://riptutorial.com/fr/csharp/topic/4584/fonctionnalites-c-sharp-5-0

Chapitre 61: Fonctionnalités C # 7.0

Introduction

C # 7.0 est la septième version de C #. Cette version contient quelques nouvelles fonctionnalités: prise en charge de la langue pour Tuples, fonctions locales, déclarations `out var` , séparateurs de chiffres, littéraux binaires, correspondance de modèle, expressions de jet, `ref return ref local` et liste de membres avec expression `ref local` et étendue.

Référence officielle: Quoi de neuf en C # 7

Examples

déclaration var out

Un modèle courant dans C # utilise `bool TryParse(object input, out object value)` pour analyser en toute sécurité des objets.

La déclaration `out var` est une fonctionnalité simple pour améliorer la lisibilité. Il permet à une variable d'être déclarée en même temps qu'elle est passée en paramètre out.

Une variable déclarée de cette façon est définie sur le reste du corps au moment où elle est déclarée.

Exemple

En utilisant `TryParse` avant C # 7.0, vous devez déclarer une variable pour recevoir la valeur avant d'appeler la fonction:

7.0

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

Dans C # 7.0, vous pouvez incorporer la déclaration de la variable transmise au paramètre `out` , ce qui élimine la nécessité d'une déclaration de variable distincte:

7.0

```
if (int.TryParse(input, out var value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body
```

Si certains des paramètres qu'une fonction retourne en `out` n'est pas nécessaire , vous pouvez utiliser l'opérateur *de défausse* _ .

```
p.GetCoordinates(out var x, out _); // I only care about x
```

Une déclaration `out var` peut être utilisée avec toute fonction existante qui possède déjà `out` paramètres. La syntaxe de déclaration de la fonction reste la même et aucune exigence supplémentaire n'est requise pour rendre la fonction compatible avec une déclaration `out var` . Cette caractéristique est simplement du sucre syntaxique.

Une autre caractéristique de la déclaration `out var` est qu'elle peut être utilisée avec des types anonymes.

7.0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
                                  {
                                      Source = x,
                                      Mod2 = x % 2
                                  })
                     .GroupBy(x => x.Mod2)
                     .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

Dans ce code, nous créons un `Dictionary` avec la clé `int` et un tableau de valeur de type anonyme. Dans la version précédente de C #, il était impossible d'utiliser la méthode `TryGetValue` ici, car vous deviez déclarer la variable `out` (qui est de type anonyme!). Cependant, avec `out var` il n'est pas nécessaire de spécifier explicitement le type de la variable `out` .

Limites

Notez que les déclarations var sont d'une utilité limitée dans les requêtes LINQ car les expressions sont interprétées comme des corps d'expression lambda, de sorte que la portée des variables introduites est limitée à ces lambda. Par exemple, le code suivant ne fonctionnera pas:

```
var nums =
```

```
    from item in seq
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

Les références

- Proposition de déclaration var originale sur GitHub

Littéraux binaires

Le préfixe **0b** peut être utilisé pour représenter les littéraux binaires.

Les littéraux binaires permettent de construire des nombres à partir de zéros et de uns, ce qui facilite la visualisation des bits définis dans la représentation binaire d'un nombre. Cela peut être utile pour travailler avec des indicateurs binaires.

Les méthodes suivantes permettent de spécifier un `int` avec la valeur `34` (= $2^5 + 2^1$):

```
// Using a binary literal:
//   bits: 76543210
int a1 = 0b00100010;          // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                  // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

Énumérations de drapeaux

Avant, la spécification de valeurs d'indicateur pour un `enum` ne pouvait être effectuée qu'en utilisant l'une des trois méthodes de cet exemple:

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //          decimal         hex       bit shifting
    Monday    =  1,     //     = 0x01    = 1 << 0
    Tuesday   =  2,     //     = 0x02    = 1 << 1
    Wednesday =  4,     //     = 0x04    = 1 << 2
    Thursday  =  8,     //     = 0x08    = 1 << 3
    Friday    = 16,     //     = 0x10    = 1 << 4
    Saturday  = 32,     //     = 0x20    = 1 << 5
    Sunday    = 64,     //     = 0x40    = 1 << 6

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Avec les littéraux binaires, il est plus clair quels bits sont définis et leur utilisation ne nécessite pas

de comprendre les nombres hexadécimaux et l'arithmétique binaire:

```
[Flags]
public enum DaysOfWeek
{
    Monday    = 0b00000001,
    Tuesday   = 0b00000010,
    Wednesday = 0b00000100,
    Thursday  = 0b00001000,
    Friday    = 0b00010000,
    Saturday  = 0b00100000,
    Sunday    = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Séparateurs de chiffres

Le trait de soulignement _ peut être utilisé comme séparateur de chiffres. Pouvoir regrouper des chiffres dans de grands littéraux numériques a un impact significatif sur la lisibilité.

Le trait de soulignement peut apparaître n'importe où dans un littéral numérique, sauf comme indiqué ci-dessous. Des regroupements différents peuvent avoir un sens dans différents scénarios ou avec des bases numériques différentes.

Toute séquence de chiffres peut être séparée par un ou plusieurs traits de soulignement. Le _ est autorisé dans les décimales ainsi que les exposants. Les séparateurs n'ont aucun impact sémantique - ils sont simplement ignorés.

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

Lorsque le _ séparateur de chiffres ne peut pas être utilisé:

- au début de la valeur (`_121`)
- à la fin de la valeur (`121_` ou `121.05_`)
- à côté de la décimale (`10_.0`)
- à côté du caractère de l'exposant (`1.1e_1`)
- à côté du spécificateur de type (`10_f`)
- immédiatement après le `0x` ou `0b` dans les littéraux binaires et hexadécimaux (peut être modifié pour permettre par exemple 0b_1001_1000)

Prise en charge linguistique pour Tuples

Les bases

Un **tuple** est une liste ordonnée et finie d'éléments. Les tuples sont couramment utilisés en programmation comme moyen de travailler collectivement avec une seule entité au lieu de travailler individuellement avec chacun des éléments du tuple et de représenter des lignes individuelles (par exemple des "enregistrements") dans une base de données relationnelle.

En C # 7.0, les méthodes peuvent avoir plusieurs valeurs de retour. Dans les coulisses, le compilateur utilisera la nouvelle structure ValueTuple .

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

Remarque : pour que cela fonctionne dans Visual Studio 2017, vous devez obtenir le package `System.ValueTuple` .

Si un résultat de méthode retournant un tuple est affecté à une seule variable, vous pouvez accéder aux membres par leurs noms définis sur la signature de la méthode:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Déconstruction de Tuple

La déconstruction des tuples sépare un tuple en ses parties.

Par exemple, l'appel de `GetTallies` et l'affectation de la valeur de retour à deux variables distinctes déconstruit le tuple en ces deux variables:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` fonctionne aussi:

```
(var s, var c) = GetTallies();
```

Vous pouvez également utiliser une syntaxe plus courte, avec `var` dehors de `()` :

```
var (s, c) = GetTallies();
```

Vous pouvez également décomposer en variables existantes:

```
int s, c;
(s, c) = GetTallies();
```

L'échange est maintenant beaucoup plus simple (pas de variable temporaire nécessaire):

```
(b, a) = (a, b);
```

Fait intéressant, tout objet peut être déconstruit en définissant une méthode `Deconstruct` dans la classe:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

Dans ce cas, la `(localFirstName, localLastName) = person` appelle `Deconstruct` sur la `person` .

La déconstruction peut même être définie dans une méthode d'extension. Ceci est équivalent à ce qui précède:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string
lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

Une approche alternative pour la classe `Person` est de définir le `Name` lui-même comme un `Tuple` . Considérer ce qui suit:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Ensuite, vous pouvez instancier une personne comme ça (où nous pouvons prendre un tuple comme argument):

```
var person = new Person(("Jane", "Smith"));
```

```
var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last;   // "Smith"
```

Initialisation du tuple

Vous pouvez également créer arbitrairement des tuples dans le code:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

Lors de la création d'un tuple, vous pouvez attribuer des noms d'élément ad hoc aux membres du tuple:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

Type d'inférence

Plusieurs tuples définis avec la même signature (types et nombre correspondants) seront déduits comme types correspondants. Par exemple:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

`stats` peuvent être retournées car la déclaration de la variable `stats` et la signature de retour de la méthode correspondent.

Noms de champ de réflexion et de tuple

Les noms de membres n'existent pas à l'exécution. Reflection considérera les tuples avec le même nombre et les mêmes types de membres, même si les noms des membres ne correspondent pas. La conversion d'un tuple en `object` , puis en un tuple avec les mêmes types de membres, mais avec des noms différents, ne provoquera pas non plus d'exception.

Alors que la classe ValueTuple elle-même ne conserve pas les informations pour les noms de membres, les informations sont disponibles par réflexion dans un TupleElementNamesAttribute. Cet attribut n'est pas appliqué au tuple lui-même mais aux paramètres de méthode, aux valeurs de retour, aux propriétés et aux champs. Cela permet de conserver les noms d'élément de tuple dans les assemblages, c'est-à-dire que si une méthode retourne (nom de chaîne, int nombre), les noms et les noms seront disponibles pour les appelants de la méthode. "nom" et "compte".

Utiliser avec des génériques et async

Les nouvelles fonctionnalités de tuple (utilisant le type `ValueTuple` sous-jacent) prennent entièrement en charge les génériques et peuvent être utilisées comme paramètre de type générique. Cela permet de les utiliser avec le modèle `async` / `await` :

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

Utiliser avec des collections

Il peut être avantageux d'avoir une collection de tuples dans (par exemple) un scénario où vous essayez de trouver un tuple correspondant aux conditions pour éviter le branchement du code.

Exemple:

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string,
string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

Avec les nouveaux tuples peuvent devenir:

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)>
labels = new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

Bien que le nom donné à l'exemple ci-dessus soit assez générique, l'idée d'étiquettes pertinentes permet une meilleure compréhension de ce qui est tenté dans le code par rapport à "item1", "item2" et "item3".

Différences entre ValueTuple et Tuple

La principale raison de l'introduction de `ValueTuple` est la performance.

Nom du type	`ValueTuple`	`Tuple`
Classe ou structure	`struct`	`class`
Mutabilité (changer les valeurs après la création)	mutable	immuable
Nommer les membres et autre support linguistique	Oui	non (à déterminer)

Les références

- Proposition de fonctionnalité de langue d'origine Tuples sur GitHub
- Une solution VS 15 exécutable pour les fonctionnalités C # 7.0
- Forfait NuGet Tuple

Fonctions locales

Les fonctions locales sont définies dans une méthode et ne sont pas disponibles en dehors de celle-ci. Ils ont accès à toutes les variables locales et prennent en charge les itérateurs, la syntaxe `async` / `await` et lambda. De cette façon, les répétitions spécifiques à une fonction peuvent être fonctionnalisées sans surcharger la classe. Comme effet secondaire, cela améliore la performance de la suggestion intellisense.

Exemple

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Les fonctions locales simplifient considérablement le code pour les opérateurs LINQ, où vous devez généralement séparer les vérifications d'argument de la logique réelle pour que les vérifications d'argument soient instantanées, et ne soient retardées qu'après le démarrage de l'itération.

Exemple

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

Les fonctions locales prennent également en charge l' `async` et `await` mots-clés.

Exemple

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-.]+");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");
```

```
    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
    {
        string text;

        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}
```

Une chose importante que vous avez peut-être remarquée est que les fonctions locales peuvent être définies dans la déclaration de `return` , elles ne doivent **pas** nécessairement être définies au-dessus. De plus, les fonctions locales suivent généralement la convention de dénomination "lowerCamelCase" pour se différencier plus facilement des fonctions de portée de classe.

Correspondance de motif

Les extensions de correspondance de modèles pour C # permettent de nombreux avantages de la correspondance de modèles à partir de langages fonctionnels, mais d'une manière qui s'intègre harmonieusement à la convivialité du langage sous-jacent

`switch` expression

La correspondance de motifs étend l' `switch` déclaration pour allumer types:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}
```

```
class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

`is` expression

La correspondance de modèle étend l'opérateur `is` pour vérifier un type et déclare une nouvelle variable en même temps.

Exemple

7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

peut être réécrit comme suit:

7.0

```
if(o is string s)
{
    //Do something with s
};
```

Notez également que la portée de la variable de modèle `s` est étendue en dehors du bloc `if`

atteignant la fin de la portée englobante, par exemple:

```
if(someCondition)
{
   if(o is string s)
   {
      //Do something with s
   }
   else
   {
     // s is unassigned here, but accessible
   }

   // s is unassigned here, but accessible
}
// s is not accessible here
```

ref retour et ref local

Les retours Ref et les sections locales de référence sont utiles pour manipuler et renvoyer des références à des blocs de mémoire au lieu de copier la mémoire sans avoir recours à des pointeurs non sécurisés.

Ref Return

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

Avec cela, vous pouvez passer deux valeurs par référence, l'une d'entre elles étant renvoyée en fonction de certaines conditions:

```
Matrix3D left = …, right = …;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Ref Local

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
…
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Opérations de référence non sécurisées

Dans `System.Runtime.CompilerServices.Unsafe` un ensemble d'opérations dangereuses ont été définies pour vous permettre de manipuler les valeurs `ref` comme s'il s'agissait de pointeurs.

Par exemple, réinterpréter une adresse mémoire (`ref`) sous un autre type:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Faites attention à la finalité lorsque vous faites cela, cependant, par exemple, vérifiez `BitConverter.IsLittleEndian` si nécessaire et gérez en conséquence.

Ou itérer sur un tableau d'une manière dangereuse:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

Ou la `Subtract` similaire:

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

De plus, on peut vérifier si deux valeurs `ref` sont identiques, c'est-à-dire même adresse:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

Liens

Question de Roslyn Github

System.Runtime.CompilerServices.Unsafe sur github

jeter des expressions

C # 7.0 permet de lancer comme expression à certains endroits:

```
class Person
{
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No
name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

Avant C # 7.0, si vous voulez lancer une exception à partir d'un corps d'expression, vous devez:

```
var spoons = "dinner,desert,soup".Split(',');

var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

Ou

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
      {
          throw new Exception("There are no spoons");
      })();
```

En C # 7.0, ce qui précède est maintenant simplifié pour:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

Expression étendue liste des membres corporels

C # 7.0 ajoute des accesseurs, des constructeurs et des finaliseurs à la liste des éléments pouvant avoir des corps d'expression:

```
class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int,
string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _);              // finalizers

    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

Voir également la section de déclaration out var pour l'opérateur de suppression.

ValueTask

`Task<T>` est une **classe** et entraîne la surcharge inutile de son allocation lorsque le résultat est immédiatement disponible.

`ValueTask<T>` est une **structure qui** a été introduite pour empêcher l'allocation d'un objet `Task` au cas où le résultat de l'opération **asynchrone** est déjà disponible au moment de l'attente.

Donc, `ValueTask<T>` offre deux avantages:

1. Augmentation de la performance

Voici un exemple de `Task<T>` :

- Nécessite une allocation de tas
- Prend 120ns avec JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Voici l'exemple `ValueTask<T>` analogique `ValueTask<T>` :

- Pas d'allocation de tas si le résultat est connu de manière synchrone (ce qui n'est pas le cas à cause de `Task.Delay` , mais se trouve souvent dans de nombreux scénarios `async` / d' `await` réels)

- Prend 65ns avec JIT

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2. Flexibilité accrue de la mise en œuvre

Les implémentations d'une interface asynchrone souhaitant être synchrone seraient sinon obligées d'utiliser `Task.Run` ou `Task.FromResult` (ce qui entraînerait une pénalité de performance discutée ci-dessus). Il y a donc une certaine pression contre les implémentations synchrones.

Mais avec `ValueTask<T>` , les implémentations sont plus libres de choisir entre être synchrones ou asynchrones sans affecter les appelants.

Par exemple, voici une interface avec une méthode asynchrone:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

... et voici comment cette méthode pourrait s'appeler:

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

Avec `ValueTask` , le code ci-dessus fonctionnera avec **des implémentations synchrones ou asynchrones** :

Implémentation synchrone:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

Implémentation asynchrone

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
```

```
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

Remarques

Bien que la structure `ValueTask` ait été prévue pour être ajoutée à C # 7.0 , elle a été conservée comme une autre bibliothèque pour le moment. Le package ValueTask <T> `System.Threading.Tasks.Extensions` peut être téléchargé à partir de Nuget Gallery

Lire Fonctionnalités C # 7.0 en ligne: https://riptutorial.com/fr/csharp/topic/1936/fonctionnalites-c-sharp-7-0

Chapitre 62: Fonctions C # 4.0

Examples

Paramètres facultatifs et arguments nommés

Nous pouvons omettre l'argument dans l'appel si cet argument est un argument facultatif Chaque argument optionnel a sa propre valeur par défaut Il prendra une valeur par défaut si nous ne fournissons pas la valeur Une valeur par défaut d'un argument facultatif doit être

1. Expression constante
2. Doit être un type de valeur tel que enum ou struct.
3. Doit être une expression du formulaire par défaut (valueType)

Il doit être défini à la fin de la liste des paramètres

Paramètres de méthode avec les valeurs par défaut:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

Comme indiqué par MSDN, un argument nommé,

Vous permet de passer l'argument à la fonction en associant le nom du paramètre Pas besoin de mémoriser la position des paramètres dont nous n'avons pas toujours conscience. Pas besoin de regarder l'ordre des paramètres dans la liste de paramètres de la fonction appelée. Nous pouvons spécifier le paramètre pour chaque argument par son nom.

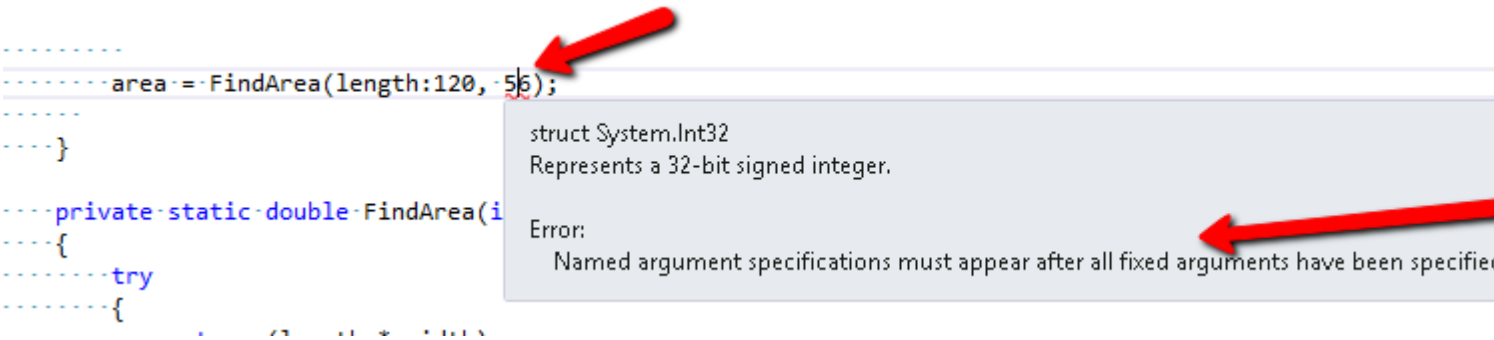
Arguments nommés:

```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

Limitation de l'utilisation d'un argument nommé

La spécification d'argument nommé doit apparaître après que tous les arguments fixes ont été spécifiés.

Si vous utilisez un argument nommé avant un argument fixe, vous obtiendrez une erreur de compilation comme suit.



La spécification d'argument nommé doit apparaître après que tous les arguments fixes ont été spécifiés

Variance

Les paramètres et les délégués génériques peuvent avoir leurs paramètres de type marqués comme *covariant* ou *contravariant en* utilisant respectivement les mots-clés `out` et `in` . Ces déclarations sont ensuite respectées pour les conversions de types, à la fois implicites et explicites, ainsi que pour la compilation et l'exécution.

Par exemple, l'interface existante `IEnumerable<T>` a été redéfinie comme étant covariante:

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

L'interface existante IComparer a été redéfinie comme étant contravariante:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Mot clé ref optionnel lors de l'utilisation de COM

Le mot-clé ref pour les appelants de méthodes est désormais facultatif lors de l'appel de méthodes fournies par les interfaces COM. Étant donné une méthode COM avec la signature

```
void Increment(ref int x);
```

l'invocation peut maintenant être écrite soit

```
Increment(0); // no need for "ref" or a place holder variable any more
```

Recherche de membre dynamique

Un nouveau type de pseudo- `dynamic` est introduit dans le système de type C #. Il est traité comme `System.Object` , mais en outre, tout accès de membre (appel de méthode, champ, propriété ou

accès à l'indexeur ou appel de délégué) ou l'application d'un opérateur sur une valeur de ce type est autorisé sans vérification de type. et sa résolution est reportée à l'exécution. Ceci est connu comme le typage de canard ou la liaison tardive. Par exemple:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");        // a string has a Length property,
GetLength(new int[] { 1, 2, 3 }); // and so does an array,
GetLength(42);                    // but not an integer - an exception will be thrown
                                  // in GetLength method at run-time
```

Dans ce cas, le type dynamique est utilisé pour éviter une réflexion plus verbeuse. Il utilise toujours Reflection sous le capot, mais il est généralement plus rapide grâce à la mise en cache.

Cette fonctionnalité est principalement destinée à l'interopérabilité avec les langages dynamiques.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

Le type dynamique a des applications même dans un code typé principalement de manière statique, par exemple, il permet une double répartition possible sans implémenter le motif de visiteur.

Lire Fonctions C # 4.0 en ligne: https://riptutorial.com/fr/csharp/topic/3093/fonctions-c-sharp-4-0

Chapitre 63: Fonctions C # 6.0

Introduction

Cette sixième itération du langage C # est fournie par le compilateur Roslyn. Ce compilateur est sorti avec la version 4.6 du .NET Framework, mais il peut générer du code de manière rétrocompatible pour permettre de cibler des versions antérieures du framework. Le code C # version 6 peut être compilé de manière totalement compatible avec .NET 4.0. Il peut également être utilisé pour des frameworks antérieurs, mais certaines fonctionnalités nécessitant un support de framework supplémentaire peuvent ne pas fonctionner correctement.

Remarques

La sixième version de C # a été publiée en juillet 2015 aux côtés de Visual Studio 2015 et .NET 4.6.

Outre l'ajout de nouvelles fonctionnalités linguistiques, il inclut une réécriture complète du compilateur. Auparavant, `csc.exe` était une application Win32 native écrite en C ++, avec C # 6, il s'agit désormais d'une application gérée .NET écrite en C #. Cette réécriture était connue sous le nom de projet "Roslyn" et le code est maintenant open source et disponible sur GitHub .

Examples

Nom de l'opérateurde

L'opérateur `nameof` renvoie le nom d'un élément de code sous la forme d'une `string` . Ceci est utile lors de la levée des exceptions liées aux arguments de méthode et lors de l'implémentation d' `INotifyPropertyChanged` .

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

L'opérateur `nameof` est évalué au moment de la compilation et change l'expression en un littéral de chaîne. Ceci est également utile pour les chaînes nommées d'après leur membre qui les expose. Considérer ce qui suit:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Comme les expressions `nameof` sont des constantes à la compilation, elles peuvent être utilisées dans des attributs, des étiquettes de `case` , des instructions de `switch` , etc.

Il est pratique d'utiliser `nameof` avec `Enum` s. Au lieu de:

```
Console.WriteLine(Enum.One.ToString());
```

il est possible d'utiliser:

```
Console.WriteLine(nameof(Enum.One))
```

La sortie sera `One` dans les deux cas.

L'opérateur `nameof` peut accéder aux membres non statiques en utilisant une syntaxe de type statique. Au lieu de faire:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

Peut être remplacé par:

```
string lengthName = nameof(string.Length);
```

La sortie sera `Length` dans les deux exemples. Cependant, ce dernier empêche la création d'instances inutiles.

Bien que l'opérateur `nameof` fonctionne avec la plupart des constructions de langage, il existe certaines limitations. Par exemple, vous ne pouvez pas utiliser l'opérateur `nameof` sur les types génériques ouverts ou les valeurs de retour de méthode:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

De plus, si vous l'appliquez à un type générique, le paramètre de type générique sera ignoré:

```
Console.WriteLine(nameof(List<int>));  // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

Pour plus d'exemples, consultez cette rubrique dédiée à `nameof` .

Solution de contournement pour les versions

précédentes (plus de détails)

Bien que l'opérateur `nameof` n'existe pas dans C # pour les versions antérieures à 6.0, il est possible d'avoir des fonctionnalités similaires en utilisant `MemberExpression` comme suit:

6,0

Expression:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Usage:

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

Notez que cette approche entraîne la création d'une arborescence d'expression à chaque appel. Les performances sont donc nettement inférieures à `nameof` opérateur `nameof` qui est évalué au moment de la compilation et dont le temps d'exécution est nul.

Membres de la fonction avec expression

Les membres de fonctions à corps d'expression permettent l'utilisation d'expressions lambda en tant que corps membres. Pour les membres simples, le code peut être plus propre et plus lisible.

Les fonctions avec expression peuvent être utilisées pour les propriétés, les indexeurs, les méthodes et les opérateurs.

Propriétés

```
public decimal TotalPrice => BasePrice + Taxes;
```

Est équivalent à:

```
public decimal TotalPrice
{
    get
```

```
    {
        return BasePrice + Taxes;
    }
}
```

Lorsqu'une fonction contenant une expression est utilisée avec une propriété, la propriété est implémentée en tant que propriété de lecture seule.

Voir la démo

Indexeurs

```
public object this[string key] => dictionary[key];
```

Est équivalent à:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

Les méthodes

```
static int Multiply(int a, int b) => a * b;
```

Est équivalent à:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Qui peut également être utilisé avec des méthodes `void` :

```
public void Dispose() => resource?.Dispose();
```

Un remplacement de `ToString` pourrait être ajouté à la classe `Pair<T>` :

```
public override string ToString() => $"{First}, {Second}";
```

En outre, cette approche simpliste fonctionne avec le mot-clé `override` :

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

Les opérateurs

Cela peut également être utilisé par les opérateurs:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

Limites

Les membres de la fonction avec expression ont certaines limitations. Ils ne peuvent contenir aucune instruction de bloc ni aucune autre instruction contenant des blocs: `if` , `switch` , `for` , `foreach` , `while` , `do` , `try` , etc.

Certains `if` les déclarations peuvent être remplacées par des opérateurs ternaires. Certaines instructions `for` et `foreach` peuvent être converties en requêtes LINQ, par exemple:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

Dans tous les autres cas, l'ancienne syntaxe des membres de la fonction peut être utilisée.

Les membres de la fonction avec expression peuvent contenir `async` / `await` , mais ils sont souvent redondants:

```
async Task<int> Foo() => await Bar();
```

Peut être remplacé par:

```
Task<int> Foo() => Bar();
```

Filtres d'exception

Les filtres d'exception permettent aux développeurs d'ajouter une condition (sous la forme d'une expression `boolean`) à un bloc catch , ce qui permet à `catch` d'exécuter uniquement si la condition est évaluée à `true` .

Les filtres d'exception permettent la propagation des informations de débogage dans l'exception d'origine, car en utilisant une instruction `if` dans un bloc `catch` et en relançant l'exception, la propagation des informations de débogage dans l'exception d'origine est interrompue. Avec les filtres d'exception, l'exception continue à se propager vers le haut dans la pile d'appels, *sauf si* la condition est remplie. Par conséquent, les filtres d'exception facilitent considérablement le débogage. Au lieu de s'arrêter sur l'instruction `throw` , le débogueur s'arrête sur l'instruction générant l'exception, avec l'état actuel et toutes les variables locales préservées. Les décharges accidentelles sont affectées de la même manière.

> Les filtres d'exception sont pris en charge par le **CLR** depuis le début et ils sont accessibles depuis plus de dix ans depuis VB.NET et F # en exposant une partie du modèle de gestion des exceptions du CLR. Ce n'est qu'après la sortie de C # 6.0 que la fonctionnalité était également disponible pour les développeurs C #.

Utilisation de filtres d'exception

Les filtres d'exception sont utilisés en ajoutant une clause `when` à l'expression `catch` . Il est possible d'utiliser n'importe quelle expression renvoyant un `bool` dans une clause `when` (sauf wait). La variable d'exception déclarée `ex` est accessible depuis la clause `when` :

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

Plusieurs blocs d' `catch` avec `when` clauses peuvent être combinées. La première `when` la clause de retour `true` entraînera l'exception à prendre. Son bloc `catch` sera entré, tandis que les autres clauses `catch` seront ignorées (leurs clauses `when` ne seront pas évaluées). Par exemple:

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                          //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                                       //someCondition evaluates to false
```

```
{ ... }
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

Article risqué quand

> **Mise en garde**
>
> Il peut être risqué d'utiliser des filtres d'exception: lorsqu'une `Exception` est générée à partir de la clause `when` , la clause `Exception` from the `when` est ignorée et traitée comme `false` . Cette approche permet aux développeurs d'écrire `when` la clause sans prendre en charge des cas invalides.

L'exemple suivant illustre un tel scénario:

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

Voir la démo

Notez que les filtres d'exception évitent les problèmes de numéro de ligne déroutants associés à l'utilisation de la méthode `throw` lorsque le code défaillant fait partie de la même fonction. Par exemple, dans ce cas, le numéro de ligne est indiqué par 6 au lieu de 3:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

Le numéro de ligne d'exception est signalé par 6 car l'erreur a été interceptée et renvoyée avec l'instruction `throw` sur la ligne 6.

La même chose ne se produit pas avec les filtres d'exception:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }
```

Dans cet exemple, `a` est 0, alors la clause `catch` est ignorée mais 3 est signalée comme numéro de ligne. C'est parce qu'ils **ne déroulent pas la pile** . Plus précisément, l'exception *n'est pas pris* en ligne 5 car `a` fait effectivement égal `0` et donc il n'y a pas possibilité de l'exception à relancées sur la ligne 6 , car la ligne 6 n'exécute pas.

Enregistrement comme effet secondaire

Les appels de méthode dans la condition peuvent entraîner des effets secondaires, donc les filtres d'exception peuvent être utilisés pour exécuter du code sur des exceptions sans les intercepter. Un exemple courant qui en profite est une méthode `Log` qui renvoie toujours `false` . Cela permet de tracer les informations du journal lors du débogage sans avoir à relancer l'exception.

> **Sachez que même** si cela semble être un moyen confortable de journalisation, cela peut être risqué, surtout si des assemblages de journalisation tiers sont utilisés. Celles-ci peuvent générer des exceptions lors de la connexion à des situations non évidentes qui peuvent ne pas être détectées facilement (voir la section **Risque `when(...)`** ci-dessus).

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // This catch block will never be reached
}

// ...

static bool Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
    return false;
}
```

Voir la démo

L'approche courante dans les versions précédentes de C # consistait à enregistrer et à renvoyer

l'exception.

6,0

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
     Log(ex, "An error occurred");
     throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

Voir la démo

Le bloc `finally`

Le bloc `finally` s'exécute chaque fois que l'exception soit lancée ou non. Une subtilité avec des expressions dans `when` les filtres d'exception sont exécutés plus haut dans la pile *avant d'* entrer dans les blocs `finally` internes. Cela peut entraîner des résultats et des comportements inattendus lorsque le code tente de modifier l'état global (comme l'utilisateur ou la culture du thread en cours) et le restaurer dans un bloc `finally` .

Exemple: bloc `finally`

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
```

```
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Sortie produite:

Début
Evaluates à: True
Intérieur enfin
Capture
Extérieur

Voir la démo

Dans l'exemple ci-dessus, si la méthode `SomeOperation` ne souhaite pas "modifier" l'état global des modifications apportées aux clauses `when` l'appelant, elle devrait également contenir un bloc `catch` pour modifier l'état. Par exemple:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
       Flag = false;
       throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Il est également courant de voir les classes d'aide `IDisposable` tirer parti de la sémantique de l' utilisation de blocs pour atteindre le même objectif, car `IDisposable.Dispose` sera toujours appelé avant qu'une exception appelée dans un bloc `using` commence à se former dans la pile.

introduction

Les propriétés peuvent être initialisées avec l'opérateur `=` après la fermeture `}` . La classe de `Coordinate` ci-dessous montre les options disponibles pour initialiser une propriété:

6,0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

Accesseurs avec une visibilité différente

Vous pouvez initialiser les propriétés automatiques ayant une visibilité différente sur leurs accesseurs. Voici un exemple avec un setter protégé:

```
    public string Name { get; protected set; } = "Cheeze";
```

L'accesseur peut également être `internal` , `internal protected` ou `private` .

Propriétés en lecture seule

Outre la flexibilité de la visibilité, vous pouvez également initialiser les propriétés automatiques en lecture seule. Voici un exemple:

```
    public List<string> Ingredients { get; } =
        new List<string> { "dough", "sauce", "cheese" };
```

Cet exemple montre également comment initialiser une propriété avec un type complexe. En outre, les propriétés automatiques ne peuvent pas être en écriture seule, ce qui empêche également l'initialisation en écriture seule.

Vieux style (pre C # 6.0)

Avant C # 6, cela nécessitait un code beaucoup plus détaillé. Nous utilisions une variable supplémentaire appelée propriété de sauvegarde pour que la propriété donne une valeur par défaut ou pour initialiser la propriété publique comme ci-dessous,

6,0

```
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

Remarque: *Avant C # 6.0, vous pouviez toujours initialiser* ***les propriétés implémentées en*** *lecture et en écriture (propriétés avec un getter et un setter) à partir du constructeur, mais vous ne pouviez pas initialiser la propriété avec sa déclaration*

Voir la démo

Usage

Les initialiseurs doivent évaluer les expressions statiques, tout comme les initialiseurs de champs. Si vous devez référencer des membres non statiques, vous pouvez initialiser des propriétés dans des constructeurs comme auparavant ou utiliser des propriétés avec des expressions. Les expressions non statiques, comme celle ci-dessous (commentée), génèrent une erreur de compilation:

```
// public decimal X { get; set; } = InitMe();  // generates compiler error

decimal InitMe() { return 4m; }
```

Mais les méthodes statiques **peuvent** être utilisées pour initialiser les propriétés automatiques:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

Cette méthode peut également être appliquée aux propriétés ayant un niveau d'accesseur

différent:

```
public short Type { get; private set; } = 15;
```

L'initialiseur de propriété automatique permet d'affecter des propriétés directement dans leur déclaration. Pour les propriétés en lecture seule, il prend en charge toutes les exigences requises pour que la propriété soit immuable. Prenons l'exemple de la classe `FingerPrint` dans l'exemple suivant:

```
public class FingerPrint
{
  public DateTime TimeStamp { get; } = DateTime.UtcNow;

  public string User { get; } =
    System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

  public string Process { get; } =
    System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

Voir la démo

Notes de mise en garde

Veillez à ne pas confondre les initialiseurs de propriétés automatiques et de champs avec les méthodes de corps d'expression similaires qui utilisent `=>` par opposition à `=` et les champs qui n'incluent pas `{ get; }` .

Par exemple, chacune des déclarations suivantes sont différentes.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Missing `{ get; }` dans la déclaration de propriété résulte dans un champ public. Les utilisateurs en lecture seule en lecture seule `Users1` et en lecture-écriture `Users2` sont initialisés qu'une seule fois, mais un champ public permet de modifier l'instance de collecte en dehors de la classe, ce qui est généralement indésirable. Changer une propriété automatique en lecture seule avec un corps d'expression en propriété en lecture seule avec initialiseur nécessite non seulement de supprimer `>` de `=>` , mais d'ajouter `{ get; }` .

Le symbole différent (`=>` au lieu de `=`) dans `Users3` résultat que chaque accès à la propriété

retourne une nouvelle instance du `HashSet<UserDto>` qui, alors que C # valide (du point de vue du compilateur) est probablement le comportement souhaité utilisé pour un membre de la collection.

Le code ci-dessus est équivalent à:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

Initialiseurs d'index

Les initialiseurs d'index permettent de créer et d'initialiser des objets avec des index en même temps.

Cela rend l'initialisation des dictionnaires très facile:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Tout objet ayant un getter ou un setter indexé peut être utilisé avec cette syntaxe:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };
```

```
        Console.ReadKey();
    }
}
```

Sortie:

Indice: toto, valeur: 34
Indice: bar, valeur: 42

Voir la démo

Si la classe a plusieurs indexeurs, il est possible de les affecter tous dans un seul groupe d'instructions:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}
```

Sortie:

Indice: toto, valeur: 34
Indice: bar, valeur: 42
Index: 10, valeur: Dix
Index: 42, valeur: Sens de la vie

Il convient de noter que l'accesseur de `set` indexeurs peut se comporter différemment par rapport à une méthode `Add` (utilisée dans les initialiseurs de collection).

Par exemple:

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one
```

contre:

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

Interpolation de chaîne

L'interpolation de chaînes permet au développeur de combiner des `variables` et du texte pour former une chaîne.

Exemple de base

Deux variables `int` sont créées: `foo` et `bar` .

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

Sortie :

> Le foo est 34 et le bar 42.

Voir la démo

Les accolades dans les chaînes peuvent toujours être utilisées, comme ceci:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is {{foo}}, and the bar is {{bar}}.");
```

Cela produit la sortie suivante:

> Le foo est {foo}, et la barre est {bar}.

Utilisation de l'interpolation avec des littéraux de chaîne textuels

L'utilisation de `@` avant la chaîne entraînera l'interprétation de la chaîne textuellement. Par exemple, les caractères Unicode ou les sauts de ligne resteront exactement tels qu'ils ont été saisis. Cependant, cela n'affectera pas les expressions d'une chaîne interpolée, comme illustré dans l'exemple suivant:

```
Console.WriteLine($@"In case it wasn't clear:
\u00B9
The foo
is {foo},
and the bar
is {bar}.");
```

Sortie:

> Au cas où ce ne serait pas clair:
> \ u00B9
> Le foo
> est 34,
> et le bar
> est 42.

Voir la démo

Expressions

Avec l'interpolation de chaîne, les *expressions* entre accolades `{}` peuvent également être évaluées. Le résultat sera inséré à l'emplacement correspondant dans la chaîne. Par exemple, pour calculer le maximum de `foo` et de `bar` et l'insérer, utilisez `Math.Max` dans les accolades:

```
Console.WriteLine($"And the greater one is: { Math.Max(foo, bar) }");
```

Sortie:

> Et le plus grand est: 42

Remarque: Tout espace de début ou de fin (espace, tabulation et CRLF / newline compris) entre l'accolade et l'expression est complètement ignoré et n'est pas inclus dans la sortie.

Voir la démo

Comme autre exemple, les variables peuvent être mises en forme en tant que devise:

```
Console.WriteLine($"Foo formatted as a currency to 4 decimal places: {foo:c4}");
```

Sortie:

> Foo au format 4 décimales: $ 34.0000

Voir la démo

Ou ils peuvent être formatés en dates:

```
Console.WriteLine($"Today is: {DateTime.Today:dddd, MMMM dd - yyyy}");
```

Sortie:

> Nous sommes aujourd'hui: lundi 20 juillet 2015

Voir la démo

Les instructions avec un opérateur conditionnel (ternaire) peuvent également être évaluées dans l'interpolation. Cependant, ceux-ci doivent être mis entre parenthèses, car les deux points sont utilisés pour indiquer le formatage comme indiqué ci-dessus:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

Sortie:

> Le bar est plus grand que le foo!

Voir la démo

Les expressions conditionnelles et les spécificateurs de format peuvent être mélangés:

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00'-bit'} process");
```

Sortie:

> Environnement: processus 32 bits

Séquences d'échappement

Les barres obliques inverses (`\`) et les guillemets (`"`) fonctionnent exactement de la même manière dans les chaînes interpolées que dans les chaînes non interpolées, à la fois pour les chaînes littérales textuelles et non textuelles:

```
Console.WriteLine($"Foo is: {foo}. In a non-verbatim string, we need to escape \" and \\ with
backslashes.");
Console.WriteLine($@"Foo is: {foo}. In a verbatim string, we need to escape "" with an extra
```

```
quote, but we don't need to escape \");
```

Sortie:

> Foo a 34 ans. Dans une chaîne non verbatim, nous devons nous échapper "et \ avec des barres obliques inverses.
> Foo a 34 ans. Dans une chaîne verbatim, nous devons nous échapper "avec une citation supplémentaire, mais nous n'avons pas besoin de nous échapper \

Pour inclure une accolade `{` ou `}` dans une chaîne interpolée, utilisez deux accolades `{{` ou `}}` :

```
$"{{foo}} is: {foo}"
```

Sortie:

> {foo} est: 34

Voir la démo

Type de chaîne formatée

Le type d'une expression d'interpolation de chaîne `$"..."` n'est pas toujours une simple chaîne. Le compilateur décide quel type attribuer en fonction du contexte:

```
string s = $"hello, {name}";
System.FormattableString s = $"Hello, {name}";
System.IFormattable s = $"Hello, {name}";
```

C'est également l'ordre de préférence de type lorsque le compilateur doit choisir la méthode surchargée à appeler.

Un nouveau type , `System.FormattableString` , représente une chaîne de format composite, avec les arguments à mettre en forme. Utilisez ceci pour écrire des applications qui gèrent spécifiquement les arguments d'interpolation:

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

Appelez la méthode ci-dessus avec:

```
AddLogItem($"The foo is {foo}, and the bar is {bar}.");
```

Par exemple, on pourrait choisir de ne pas encourir le coût des performances de la mise en forme de la chaîne si le niveau de journalisation était déjà utilisé pour filtrer l'élément de journal.

Conversions implicites

Il existe des conversions de types implicites à partir d'une chaîne interpolée:

```
var s = $"Foo: {foo}";
System.IFormattable s = $"Foo: {foo}";
```

Vous pouvez également produire une variable `IFormattable` qui vous permet de convertir la chaîne avec un contexte invariant:

```
var s = $"Bar: {bar}";
System.FormattableString s = $"Bar: {bar}";
```

Méthodes de culture actuelles et invariantes

Si l'analyse de code est activée, les chaînes interpolées produiront toutes un avertissement CA1305 (spécifiez `IFormatProvider`). Une méthode statique peut être utilisée pour appliquer la culture actuelle.

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }
    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

Ensuite, pour produire une chaîne correcte pour la culture actuelle, utilisez simplement l'expression:

```
Culture.Current($"interpolated {typeof(string).Name} string.")
Culture.Invariant($"interpolated {typeof(string).Name} string.")
```

Remarque : `Current` et `Invariant` ne peuvent pas être créés en tant que méthodes d'extension car, par défaut, le compilateur attribue le type `String` à l' *expression de chaîne interpolée* , ce qui empêche la compilation du code suivant:

```
$"interpolated {typeof(string).Name} string.".Current();
```

`FormattableString` classe `FormattableString` contient déjà la méthode `Invariant()` , de sorte que le moyen le plus simple de passer à la culture invariante est d' `using static` :

```
using static System.FormattableString;


string invariant = Invariant($"Now = {DateTime.Now}");
string current = $"Now = {DateTime.Now}";
```

Dans les coulisses

Les chaînes interpolées ne sont qu'un sucre syntaxique pour `String.Format()` . Le compilateur (Roslyn) le transformera en `String.Format` en coulisse:

```
var text = $"Hello {name + lastName}";
```

Ce qui précède sera converti en quelque chose comme ceci:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

Interpolation de chaînes et Linq

Il est possible d'utiliser des chaînes interpolées dans les instructions Linq pour augmenter la lisibilité.

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Peut être réécrit comme:

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select $"{x["foo"]}{x["bar"]}").ToList();
```

Cordes interpolées réutilisables

Avec `string.Format` , vous pouvez créer des chaînes de format réutilisables:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";
```

```
// ...

Logger.Log(string.Format(ErrorFormat, ex));
```

Les chaînes interpolées, cependant, ne seront pas compilées avec les espaces réservés faisant référence à des variables inexistantes. Les éléments suivants ne seront pas compilés:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

Au lieu de cela, créez un `Func<>` qui consomme des variables et renvoie une `String` :

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";

// ...

Logger.Log(FormatError(ex));
```

Interpolation et localisation de chaînes

Si vous localisez votre application, vous pouvez vous demander s'il est possible d'utiliser l'interpolation de chaînes avec la localisation. En effet, il serait agréable d'avoir la possibilité de stocker dans des fichiers ressources `String` s comme:

```
"My name is {name} {middlename} {surname}"
```

au lieu de beaucoup moins lisible:

```
"My name is {0} {1} {2}"
```

`String` processus d'interpolation de `String` se produit *au moment de la compilation* , contrairement à la chaîne de formatage avec `string.Format` qui se produit *au moment de l'exécution* . Les expressions dans une chaîne interpolée doivent référencer des noms dans le contexte actuel et doivent être stockées dans des fichiers de ressources. Cela signifie que si vous voulez utiliser la localisation, vous devez le faire comme:

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
```

```
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

Si les chaînes de ressources des langages utilisés ci-dessus sont correctement stockées dans les fichiers de ressources individuels, vous devez obtenir la sortie suivante:

> Bonjour, mon nom est John
> Bonjour, nommez-vous John
> Bonjour je m'appelle John

Notez que cela implique que le nom suit la chaîne localisée dans toutes les langues. Si ce n'est pas le cas, vous devez ajouter des espaces réservés aux chaînes de ressources et modifier la fonction ci-dessus ou vous devez interroger les informations de culture dans la fonction et fournir une instruction de casse contenant les différents cas. Pour plus de détails sur les fichiers de ressources, consultez Comment utiliser la localisation en C # .

Il est recommandé d'utiliser un langage de secours par défaut que la plupart des gens comprendront, au cas où une traduction ne serait pas disponible. Je suggère d'utiliser l'anglais comme langue de secours par défaut.

Interpolation récursive

Bien que ce ne soit pas très utile, il est permis d'utiliser une `string` interpolée récursivement à l'intérieur des accolades d'un autre:

```
Console.WriteLine($"String has {$"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");
```

Sortie:

> La chaîne a 27 caractères:
>
> Ma classe s'appelle MyClass.

Attendez dans la prise et enfin

Il est possible d'utiliser une expression en `await` pour appliquer un opérateur en attente à Tasks ou Task (Of TResult) dans les blocs `catch` et `finally` dans C # 6.

Il n'était pas possible d'utiliser l'expression `await` dans les blocs `catch` et `finally` dans les versions antérieures en raison des limitations du compilateur. C # 6 rend l'attente des tâches asynchrones beaucoup plus facile en permettant l'expression d' `await` .

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

Il était nécessaire en C # 5 d'utiliser un `bool` ou de déclarer une `Exception` dehors du try try pour effectuer des opérations asynchrones. Cette méthode est illustrée dans l'exemple suivant:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

Propagation nulle

Le `?.` L'opérateur et l'opérateur `?[...]` sont appelés l' opérateur null-conditionnel . Il est également parfois appelé par d'autres noms tels que l' opérateur de navigation sécurisé .

Ceci est utile, car si le `.` L'opérateur (accesseur membre) est appliqué à une expression dont la valeur est `null` , le programme lancera une `NullReferenceException` . Si le développeur utilise plutôt le `?.` (null-conditionnel) opérateur, l'expression évaluera à null au lieu de lancer une exception.

Notez que si le `?.` l'opérateur est utilisé et l'expression est non nulle, `?.` et `.` sont équivalents.

Les bases

```
var teacherName = classroom.GetTeacher().Name;
// throws NullReferenceException if GetTeacher() returns null
```

Voir la démo

Si la `classroom` n'a pas d'enseignant, `GetTeacher()` peut renvoyer `null` . Lorsqu'il est `null` et que la propriété `Name` est `NullReferenceException` , une `NullReferenceException` sera lancée.

Si nous modifions cette déclaration pour utiliser le `?.` la syntaxe, le résultat de l'expression entière sera `null` :

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

Voir la démo

Par la suite, si la `classroom` pouvait également être `null` , nous pourrions également écrire cette déclaration comme suit:

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

Voir la démo

Voici un exemple de mise en court-circuit: Lorsqu'une opération d'accès conditionnel utilisant l'opérateur null-conditionnel a la valeur null, l'expression entière est évaluée immédiatement à null, sans traitement du reste de la chaîne.

Lorsque le membre terminal d'une expression contenant l'opérateur null-conditionnel est d'un type valeur, l'expression est évaluée à un `Nullable<T>` de ce type et ne peut donc pas être utilisée comme remplacement direct de l'expression sans `?.` .

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
```

```
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null

bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

Utiliser avec l'opérateur Null-Coalescing (??)

Vous pouvez combiner l'opérateur null-conditionnel avec l'opérateur Null-coalescing (`??`) pour renvoyer une valeur par défaut si l'expression est résolue à `null` . En utilisant notre exemple ci-dessus:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

Utiliser avec des indexeurs

L'opérateur null-conditionnel peut être utilisé avec les indexeurs :

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

Dans l'exemple ci-dessus:

- Le premier `?.` s'assure que la `classroom` n'est pas `null` .
- La seconde `?` s'assure que toute la collection `Students` n'est pas `null` .
- Le troisième `?.` après l'indexeur s'assure que l'indexeur `[0]` n'a pas renvoyé d'objet `null` . Il convient de noter que cette opération peut **toujours** lancer une `IndexOutOfRangeException` .

Utiliser avec fonctions vides

L'opérateur Null-Conditionnel peut également être utilisé avec les fonctions `void` . Cependant, dans ce cas, l'instruction ne sera pas évaluée à `null` . Cela empêchera simplement une `NullReferenceException` .

```
List<string> list = null;
list?.Add("hi");          // Does not evaluate to null
```

Utiliser avec l'invocation d'événement

En supposant la définition d'événement suivante:

```
private event EventArgs OnCompleted;
```

Lors de l'appel d'un événement, traditionnellement, il est recommandé de vérifier si l'événement est `null` si aucun abonné n'est présent:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Comme l'opérateur null-conditionnel a été introduit, l'invocation peut être réduite à une seule ligne:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

Limites

L'opérateur Null-conditionnel produit rvalue, pas lvalue, c'est-à-dire qu'il ne peut pas être utilisé pour l'affectation de propriété, l'abonnement à un événement, etc. Par exemple, le code suivant ne fonctionnera pas:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or -=
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

Gotchas

Notez que:

```
int? nameLength = person?.Name.Length;    // safe if 'person' is null
```

n'est **pas** la même chose que:

```
int? nameLength = (person?.Name).Length;  // avoid this
```

parce que le premier correspond à:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

et ce dernier correspond à:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

Malgré l'opérateur ternaire `?:` Est utilisé ici pour expliquer la différence entre deux cas, ces opérateurs ne sont pas équivalents. Cela peut être facilement démontré avec l'exemple suivant:

```
void Main()
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

Quelles sorties:

Propagation nulle
J'ai été lu
0
Ternaire
J'ai été lu
J'ai été lu
0

Voir la démo

Pour éviter les invocations multiples, l'équivalent serait:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

Et cette différence explique en partie pourquoi l'opérateur de propagation nul n'est pas encore pris en charge dans les arbres d'expression.

En utilisant le type statique

La directive `using static [Namespace.Type]` permet d'importer des membres statiques de types et de valeurs d'énumération. Les méthodes d'extension sont importées en tant que méthodes d'extension (à partir d'un seul type) et non dans la portée de niveau supérieur.

6,0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;

class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Live Demo Fiddle

6,0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

Amélioration de la résolution de la surcharge

L'extrait de code suivant montre un exemple de passage d'un groupe de méthodes (par opposition à un groupe lambda) lorsqu'un délégué est attendu. La résolution de la surcharge va maintenant résoudre ce problème au lieu de générer une erreur de surcharge ambiguë en raison de la capacité de **C # 6** à vérifier le type de retour de la méthode qui a été transmise.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
       Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
       Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
```

```
        Console.WriteLine(0);
        return 0;
    }
}
```

Résultats:

6,0

Sortie

> surcharge avec Func <int> appelée

Voir la démo

5.0

Erreur

> erreur CS0121: l'appel est ambigu entre les méthodes ou propriétés suivantes: 'Program.Overloaded (System.Action)' et 'Program.Overloaded (System.Func)'

C # 6 peut également gérer le cas suivant de correspondance exacte pour les expressions lambda, ce qui aurait entraîné une erreur dans **C # 5** .

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

Changements mineurs et corrections de bugs

Les parenthèses sont désormais interdites autour des paramètres nommés. Les éléments suivants sont compilés en C # 5, mais pas en C # 6

5.0

```
Console.WriteLine((value: 23));
```

Les opérandes de `is` et `as` ne sont plus autorisés à être des groupes de méthodes. Les éléments suivants sont compilés en C # 5, mais pas en C # 6

5.0

```
var result = "".Any is byte;
```

> Le compilateur natif a autorisé cela (bien qu'il ait montré un avertissement), et n'a même pas vérifié la compatibilité des méthodes d'extension, autorisant des choses délirantes comme `1.Any is string` ou `IDisposable.Dispose is object` .

Voir cette référence pour les mises à jour sur les modifications.

Utilisation d'une méthode d'extension pour l'initialisation de la collection

La syntaxe d'initialisation de la collection peut être utilisée lors de l'instanciation de toute classe qui implémente `IEnumerable` et possède une méthode nommée `Add` qui prend un seul paramètre.

Dans les versions précédentes, cette méthode `Add` devait être une méthode d' **instance** sur la classe en cours d'initialisation. En C # 6, cela peut aussi être une méthode d'extension.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}
```

Cela va sortir:

Elément ajouté avec la méthode d'instance ajouter: 1
Elément ajouté avec la méthode d'instance ajouter: 2
Elément ajouté avec la méthode d'instance ajouter: 3
Article ajouté avec l'extension add method: 4
Élément ajouté avec l'extension add method: 5
Article ajouté avec l'extension add method: 6

Désactiver les améliorations des avertissements

Dans C # 5.0 et versions antérieures, le développeur ne pouvait supprimer que les avertissements par numéro. Avec l'introduction des analyseurs Roslyn, C # doit pouvoir désactiver les avertissements émis par des bibliothèques spécifiques. Avec C # 6.0, la directive pragma peut supprimer les avertissements par nom.

Avant:

```
#pragma warning disable 0501
```

C # 6.0:

```
#pragma warning disable CS0501
```

Lire Fonctions C # 6.0 en ligne: https://riptutorial.com/fr/csharp/topic/24/fonctions-c-sharp-6-0

Chapitre 64: Fonctions de hachage

Remarques

MD5 et SHA1 ne sont pas sûrs et doivent être évités. Les exemples existent à des fins éducatives et parce que les logiciels existants peuvent encore utiliser ces algorithmes.

Examples

MD5

Les fonctions de hachage mappent les chaînes binaires de longueur arbitraire sur de petites chaînes binaires de longueur fixe.

L'algorithme `MD5` est une fonction de hachage largement utilisée qui produit une valeur de hachage de 128 bits (16 octets, 32 caractères hexadécimaux).

La méthode `ComputeHash` de la classe `System.Security.Cryptography.MD5` renvoie le hachage sous la forme d'un tableau de 16 octets.

Exemple:

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

Sortie: Le hash MD5 de Hello World! est: ED076287532E86365E841E92BFC50D8C

Les problèmes de sécurité:

Comme la plupart des fonctions de hachage, MD5 n'est ni un cryptage ni un encodage. Il peut être inversé par une attaque par force brute et souffre d'importantes vulnérabilités contre les collisions et les attaques par pré-image.

SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-",String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
 }
```

Sortie:

Le hash SHA1 de Hello Word! est: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

SHA256

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
```

```
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Sortie:

Le hash SHA256 de Hello World! est:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

SHA384

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Sortie:

Le hash SHA384 de Hello World! est:
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD

SHA512

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Sortie: Le hash SHA512 de Hello World! est:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

PBKDF2 pour hachage de mot de passe

PBKDF2 ("Fonction de dérivation de clé par mot de passe 2") est l'une des fonctions de hachage recommandées pour le hachage de mot de passe. Cela fait partie de la RFC-2898 .

`Rfc2898DeriveBytes` -Class de .NET est basé sur HMACSHA1.

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2 nécessite un sel et le nombre d'itérations.

Itérations:

Un nombre élevé d'itérations ralentira l'algorithme, ce qui rend le piratage des mots de passe beaucoup plus difficile. Un nombre élevé d'itérations est donc recommandé. PBKDF2 est un ordre de grandeur plus lent que MD5 par exemple.

Sel:

Un sel empêchera la recherche de valeurs de hachage dans les tables arc- en- ciel. Il doit être stocké avec le mot de passe hash. Un sel par mot de passe (pas un sel global) est recommandé.

Solution complète de hachage de mot de passe avec Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
  {
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could
be persisted as bytes or as a string using the convenience methods in the next class to
convert from byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary
based attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
      }
    }
    /// <summary>
```

```
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
  public sealed class PasswordHashContainer
  {
    /// <summary>
```

```
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
```

```
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/
```

S'il vous plaît voir cette excellente ressource Crackstation - hachage de mot de passe salé - le faire bien pour plus d'informations. Une partie de cette solution (la fonction de hachage) était basée sur le code de ce site.

Lire Fonctions de hachage en ligne: https://riptutorial.com/fr/csharp/topic/2774/fonctions-de-hachage

Chapitre 65: Fonderie

Remarques

Le casting est différent de la *conversion* . Il est possible de convertir la valeur de chaîne `"-1"` en une valeur entière (`-1`), mais cela doit être fait par des méthodes de bibliothèque telles que `Convert.ToInt32()` ou `Int32.Parse()` . Cela ne peut pas être fait en utilisant directement la syntaxe de casting.

Examples

Lancer un objet sur un type de base

Compte tenu des définitions suivantes:

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Lancer un objet dans un exemple de type de base:

```
    MyClass obj = new MyClass();

    IMyInterface1 myClass1 = (IMyInterface1)obj;
    IMyInterface2 myClass2 = (IMyInterface2)obj;

    Console.WriteLine("I am : {0}", myClass1.GetName());
    Console.WriteLine("I am : {0}", myClass2.GetName());

    // Outputs :
    // I am : IMyInterface1
    // I am : IMyInterface2
```

Casting explicite

Si vous savez qu'une valeur est d'un type spécifique, vous pouvez la convertir explicitement dans ce type afin de l'utiliser dans un contexte où ce type est nécessaire.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

Si nous essayions de transmettre directement la `value` à `Math.Abs()` , nous obtiendrions une exception à la `Math.Abs()` car `Math.Abs()` n'a pas de surcharge qui prend un `object` comme paramètre.

Si la `value` n'a pas pu être `InvalidCastException` en `int` , la deuxième ligne de cet exemple `InvalidCastException` une `InvalidCastException`

Casting explicite sûr (opérateur `as`)

Si vous ne savez pas si une valeur est du type que vous pensez, vous pouvez la lancer en toute sécurité en utilisant l'opérateur `as` . Si la valeur n'est pas de ce type, la valeur résultante sera `null` .

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Notez que `null` valeurs `null` n'ont pas de type, de sorte `as` mot-clé `as` produira en toute sécurité `null` lors de la conversion de toute valeur `null` .

Casting implicite

Une valeur sera automatiquement convertie dans le type approprié si le compilateur sait qu'il peut toujours être converti en ce type.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

Dans cet exemple, nous n'avons pas eu besoin d'utiliser la syntaxe de transtypage explicite car le compilateur sait que tous les `int` s peuvent être convertis en `object` s. En fait, nous pourrions éviter de créer des variables et transmettre directement `-1` comme argument de `Console.WriteLine()` qui attend un `object` .

```
Console.WriteLine(-1);
```

Vérification de la compatibilité sans coulée

Si vous avez besoin de savoir si le type d'une valeur étend ou implémente un type donné, mais que vous ne voulez pas le lancer comme tel, vous pouvez utiliser l'opérateur `is` .

```
if(value is int)
{
   Console.WriteLine(value + "is an int");
}
```

Conversions numériques explicites

Les opérateurs de transtypage explicite peuvent être utilisés pour effectuer des conversions de types numériques, même s'ils ne s'étendent pas ou s'implémentent mutuellement.

```
double value = -1.1;
int number = (int) value;
```

Notez que dans les cas où le type de destination est moins précis que le type d'origine, la précision sera perdue. Par exemple, `-1.1` tant que valeur double dans l'exemple ci-dessus devient `-1` tant que valeur entière.

De même, les conversions numériques reposent sur des types à la compilation, de sorte qu'elles ne fonctionneront pas si les types numériques ont été "encadrés" dans des objets.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

Opérateurs de conversion

En C #, les types peuvent définir des *opérateurs de conversion* personnalisés, qui permettent de convertir les valeurs depuis et vers d'autres types en utilisant des *conversions* explicites ou implicites. Par exemple, considérez une classe destinée à représenter une expression JavaScript:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

Si nous voulions créer une expression JsExpression représentant une comparaison de deux valeurs JavaScript, nous pourrions faire quelque chose comme ceci:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Mais nous pouvons ajouter *des opérateurs de conversion explicites* à `JsExpression` , pour permettre une conversion simple lors de l'utilisation de transtypage explicite.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Ou, nous pourrions changer ces opérateurs en *implicite* pour rendre la syntaxe beaucoup plus simple.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

Opérations de coulée LINQ

Supposons que vous ayez des types comme ceux-ci:

```
interface IThing {  }
class Thing : IThing {  }
```

LINQ vous permet de créer une projection qui modifie le type générique de compilation d'un `IEnumerable<>` via les méthodes d'extension `Enumerable.Cast<>()` et `Enumerable.OfType<>()` .

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

Lorsque `things2` est évalué, la méthode `Cast<>()` essaiera de transposer toutes les valeurs des `things` dans `Thing` s. S'il rencontre une valeur qui ne peut pas être `InvalidCastException` , une

`InvalidCastException` sera lancée.

Lorsque `things3` est évalué, la `OfType<>()` fera de même, sauf que si elle rencontre une valeur qui ne peut pas être convertie, elle omettra simplement cette valeur plutôt que de lancer une exception.

En raison du type générique de ces méthodes, ils ne peuvent pas appeler des opérateurs de conversion ou effectuer des conversions numériques.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

Vous pouvez simplement effectuer un transtypage dans un `.Select()` comme solution de contournement:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Lire Fonderie en ligne: https://riptutorial.com/fr/csharp/topic/2690/fonderie

Chapitre 66: Garbage Collector dans .Net

Examples

Compactage de tas de gros objets

Par défaut, le tas d'objets volumineux n'est pas compacté, contrairement au segment d'objets classique, ce qui peut entraîner une fragmentation de la mémoire et entraîner une `OutOfMemoryException` s.

À partir de .NET 4.5.1, il existe une option permettant de compacter explicitement le tas de grands objets (avec un ramasse-miettes):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Tout comme toute demande explicite de récupération de place (appelée requête parce que le CLR n'est pas obligé de le faire), utiliser avec précaution et par défaut l'éviter si possible, car elle peut dé-calibrer les statistiques du `GC` , en réduisant ses performances.

Références faibles

Dans .NET, le GC alloue des objets lorsqu'il ne leur reste aucune référence. Par conséquent, même si un objet peut toujours être atteint à partir du code (il y a une référence forte), le GC n'allouera pas cet objet. Cela peut devenir un problème s'il y a beaucoup de gros objets.

Une référence faible est une référence qui permet au GC de collecter l'objet tout en permettant d'accéder à l'objet. Une référence faible n'est valide que pendant la durée indéterminée jusqu'à ce que l'objet soit collecté en l'absence de références fortes. Lorsque vous utilisez une référence faible, l'application peut toujours obtenir une référence forte à l'objet, ce qui l'empêche d'être collecté. Les références faibles peuvent donc être utiles pour conserver des objets volumineux dont l'initialisation est coûteuse, mais qui doivent être disponibles pour la récupération de mémoire s'ils ne sont pas activement utilisés.

Utilisation simple:

```
WeakReference reference = new WeakReference(new object(), false);

GC.Collect();

object target = reference.Target;
if (target != null)
  DoSomething(target);
```

Des références faibles pourraient donc être utilisées pour conserver, par exemple, un cache d'objets. Cependant, il est important de garder à l'esprit qu'il existe toujours le risque que le garbage collector atteigne l'objet avant qu'une référence forte ne soit rétablie.

Les références faibles sont également utiles pour éviter les fuites de mémoire. Un cas d'utilisation typique concerne les événements.

Supposons que nous ayons un gestionnaire pour un événement sur une source:

```
Source.Event += new EventHandler(Handler)
```

Ce code enregistre un gestionnaire d'événement et crée une référence forte de la source d'événement à l'objet d'écoute. Si l'objet source a une durée de vie plus longue que l'écouteur et que l'écouteur n'a plus besoin de l'événement lorsqu'il n'y a pas d'autres références, l'utilisation d'événements .NET normaux provoque une fuite de mémoire: l'objet source contient des objets d'écouteur en mémoire devrait être ramassé des ordures.

Dans ce cas, il peut être utile d'utiliser le modèle d'événement faible .

Quelque chose comme:

```
public static class WeakEventManager
    {
    public static void SetHandler<S, TArgs>(
    Action<EventHandler<TArgs>> add,
    Action<EventHandler<TArgs>> remove,
    S subscriber,
    Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
        {
            var subscrWeakRef = new WeakReference(subscriber);
            EventHandler<TArgs> handler = null;

            handler = (s, e) =>
            {
                var subscrStrongRef = subscrWeakRef.Target as S;
                if (subscrStrongRef != null)
                {
                    action(subscrStrongRef, e);
                }
                else
                {
                    remove(handler);
                    handler = null;
                }
            };

            add(handler);
        }
    }
```

et utilisé comme ceci:

```
 EventSource s = new EventSource();
 Subscriber subscriber = new Subscriber();
 WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

Dans ce cas, bien sûr, nous avons des restrictions - l'événement doit être un

```
public event EventHandler<SomeEventArgs> Event;
```

Comme MSDN le suggère:

- Utilisez des références longues et faibles uniquement lorsque cela est nécessaire, car l'état de l'objet est imprévisible après la finalisation.
- Évitez d'utiliser des références faibles à de petits objets, car le pointeur lui-même peut être plus grand ou plus grand.
- Évitez d'utiliser des références faibles comme solution automatique aux problèmes de gestion de la mémoire. Au lieu de cela, développez une stratégie de mise en cache efficace pour gérer les objets de votre application.

Lire Garbage Collector dans .Net en ligne: https://riptutorial.com/fr/csharp/topic/1287/garbage-collector-dans--net

Chapitre 67: Générateur de requêtes Lambda générique

Remarques

La classe s'appelle `ExpressionBuilder` . Il a trois propriétés:

```
 private static readonly MethodInfo ContainsMethod = typeof(string).GetMethod("Contains",
new[] { typeof(string) });
 private static readonly MethodInfo StartsWithMethod = typeof(string).GetMethod("StartsWith",
new[] { typeof(string) });
 private static readonly MethodInfo EndsWithMethod = typeof(string).GetMethod("EndsWith",
new[] { typeof(string) });
```

Une méthode publique `GetExpression` qui renvoie l'expression lambda et trois méthodes privées:

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

Toutes les méthodes sont expliquées en détail dans les exemples.

Examples

Classe QueryFilter

Cette classe contient des valeurs de filtres de prédicats.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Enum pour contenir les valeurs des opérations:

```
 public enum Operator
```

```
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

Méthode GetExpression

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param
part
    // To create a ParameterExpression need the type of the entity that the query is against
an a name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good parctice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put I in the query two at a time
            while (filters.Count > 0)
            {
                // Retreive the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration
finds the next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
```

```
                if (filters.Count == 1)
                {
                    exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
                    filters.RemoveAt(0);
                }
            }
        }
    }
    else
        // It is result from direct call.
        exp = GetExpression<T>(param, filters[0]);

           // converts the Expression into Lambda and retuns the query
    return Expression.Lambda<Func<T, bool>>(exp, param);
}
```

GetExpression Surcharge privée

Pour un filtre:

Voici où la requête est créée, il reçoit un paramètre d'expression et un filtre.

```
private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for
example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
```

```
            return Expression.Call(member, StartsWithMethod, constant);

        case Operator.EndsWith:
            return Expression.Call(member, EndsWithMethod, constant);
    }

    return null;
}
```

Pour deux filtres:

Il retourne l'instance BinaryExpresion au lieu de l'expression simple.

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter
filter1, QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

Méthode ConstantExpression

`ConstantExpression` doit être du même type que `MemberExpression` . La valeur dans cet exemple est une chaîne, qui est convertie avant de créer l'instance de `ConstantExpression` .

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if(type == typeof(string))
    {
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
        }
        constant = Expression.Constant(flag);
```

```
    }
    else if (type == typeof(decimal))
    {
        decimal number;
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

Usage

Filtres de collection = new List (); QueryFilter filter = new QueryFilter ("Name", "Burger", Operator.StartsWith); filtres.Add (filtre);

```
 Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

Dans ce cas, il s'agit d'une requête sur l'entité Food, qui souhaite rechercher tous les aliments commençant par "Burger" dans le nom.

Sortie:

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Lire Générateur de requêtes Lambda générique en ligne:
https://riptutorial.com/fr/csharp/topic/6721/generateur-de-requetes-lambda-generique

Chapitre 68: Génération de code T4

Syntaxe

- **Syntaxe T4**
- `<#@...#>` // Déclaration des propriétés, y compris les modèles, les assemblys et les espaces de noms, et la langue utilisée par le modèle
- `Plain Text` // Déclaration de texte pouvant être parcouru en boucle pour les fichiers générés
- `<#=...#>` // Déclaration de scripts
- `<#+...#>` // Déclaration de scriptlets
- `<#...#>` // Déclaration de blocs de texte

Examples

Génération de code d'exécution

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

Lire Génération de code T4 en ligne: https://riptutorial.com/fr/csharp/topic/4824/generation-de-code-t4

Chapitre 69: Génération de nombres aléatoires en C

Syntaxe

- Au hasard()
- Aléatoire (en graine)
- int Next ()
- int Next (int maxValue)
- int Next (int minValue, int maxValue)

Paramètres

Paramètres	Détails
La graine	Une valeur pour générer des nombres aléatoires. Si elle n'est pas définie, la valeur par défaut est déterminée par l'heure système actuelle.
minValue	Les nombres générés ne seront pas inférieurs à cette valeur. S'il n'est pas défini, la valeur par défaut est 0.
Valeur max	Les nombres générés seront plus petits que cette valeur. S'il n'est pas défini, la valeur par défaut est `Int32.MaxValue` .
valeur de retour	Renvoie un nombre avec une valeur aléatoire.

Remarques

La graine aléatoire générée par le système n'est pas la même dans chaque cycle.

Les graines produites dans le même temps peuvent être les mêmes.

Examples

Générer un int aléatoire

Cet exemple génère des valeurs aléatoires comprises entre 0 et 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Générer un double aléatoire

Générer un nombre aléatoire entre 0 et 1.0. (n'incluant pas 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Générer un int aléatoire dans une plage donnée

Générer un nombre aléatoire entre `minValue` et `maxValue - 1` .

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Générer la même séquence de nombres aléatoires encore et encore

Lors de la création d'instances `Random` avec la même graine, les mêmes numéros seront générés.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
   Console.WriteLine("Random instance " + i);
   Random rnd = new Random(seed);
   for (int j = 0; j < 5; j++)
   {
      Console.Write(rnd.Next());
      Console.Write(" ");
   }

   Console.WriteLine();
}
```

Sortie:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

Créer plusieurs classes aléatoires avec différentes graines simultanément

Deux classes aléatoires créées en même temps auront la même valeur de départ.

En utilisant `System.Guid.NewGuid().GetHashCode()` peut obtenir une graine différente même dans le même temps.

```
Random rnd1 = new Random();
```

```
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

Une autre façon d'obtenir différentes valeurs consiste à utiliser une autre instance `Random` pour récupérer les valeurs de départ.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

Cela permet également de contrôler le résultat de toutes les instances `Random` en définissant uniquement la valeur de départ pour les `rndSeeds` . Toutes les autres instances seront dérivées de manière déterministe de cette valeur de graine unique.

Générer un caractère aléatoire

Générer une lettre aléatoire entre `a` et `z` en utilisant la `Next()` surcharge pour une plage de nombres, puis la conversion résultant `int` à un `char`

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

Générer un nombre qui est un pourcentage d'une valeur maximale

Un besoin commun pour les nombres aléatoires de générer un nombre qui est `X%` d'une valeur maximale. Cela peut être fait en traitant le résultat de `NextDouble()` en pourcentage:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

Lire Génération de nombres aléatoires en C # en ligne:
https://riptutorial.com/fr/csharp/topic/1975/generation-de-nombres-aleatoires-en-c-sharp

Chapitre 70: Génériques

Syntaxe

- `public void SomeMethod <T> () { }`
- `public void SomeMethod<T, V>() { }`
- `public T SomeMethod<T>(IEnumerable<T> sequence) { ... }`
- `public void SomeMethod<T>() where T : new() { }`
- `public void SomeMethod<T, V>() where T : new() where V : struct { }`
- `public void SomeMethod<T>() where T: IDisposable { }`
- `public void SomeMethod<T>() where T: Foo { }`
- `public class MyClass<T> { public T Data {get; set; } }`

Paramètres

Paramètres)	La description
LA TÉLÉ	Tapez les espaces réservés pour les déclarations génériques

Remarques

Les génériques en C # sont pris en charge jusqu'au moment de l'exécution: les types génériques construits avec C # verront leur sémantique générique préservée même après la compilation en CIL .

Cela signifie que, en C #, il est possible de réfléchir aux types génériques et de les voir tels qu'ils ont été déclarés ou de vérifier si un objet est une instance de type générique, par exemple. Ceci est en contraste avec l' effacement de type , où les informations de type génériques sont supprimées lors de la compilation. Cela contraste également avec l'approche par modèle des génériques, où plusieurs types génériques concrets deviennent de multiples types non génériques à l'exécution, et toutes les métadonnées requises pour instancier davantage les définitions de type génériques d'origine sont perdues.

Soyez prudent, cependant, lorsque vous réfléchissez sur des types génériques: les noms des types génériques seront modifiés lors de la compilation, en remplaçant les noms entre parenthèses et les paramètres de type par un backtick suivi du nombre de paramètres de type génériques. Ainsi, un `Dictionary<TKey, Tvalue>` sera traduit en `Dictionary`2` .

Examples

Paramètres de type (classes)

Déclaration:

```
class MyGenericClass<T1, T2, T3, ...>
{
    // Do something with the type parameters.
}
```

Initialisation:

```
var x = new MyGenericClass<int, char, bool>();
```

Utilisation (comme type de paramètre):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Paramètres de type (méthodes)

Déclaration:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Invocation:

Il n'est pas nécessaire de fournir des arguments de type à une méthode générique, car le compilateur peut implicitement déduire le type.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

Cependant, s'il ya une ambiguïté, les méthodes génériques doivent être appelées avec des arguments de type

```
MyGenericMethod<int, int, string>(x,y,z);
```

Paramètres de type (interfaces)

Déclaration:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Utilisation (en héritage):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }
```

```
class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Utilisation (comme type de paramètre):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

Inférence de type implicite (méthodes)

Lors de la transmission d'arguments formels à une méthode générique, les arguments de type générique pertinents peuvent généralement être déduits implicitement. Si tous les types génériques peuvent être déduits, leur spécification dans la syntaxe est facultative.

Considérons la méthode générique suivante. Il a un paramètre formel et un paramètre de type générique. Il existe une relation très évidente entre eux - le type passé en argument au paramètre de type générique doit être le même que le type de compilation de l'argument passé au paramètre formel.

```
void M<T>(T obj)
{
}
```

Ces deux appels sont équivalents:

```
M<object>(new object());
M(new object());
```

Ces deux appels sont aussi équivalents:

```
M<string>("");
M("");
```

Et ainsi sont ces trois appels:

```
M<object>("");
M((object) "");
M("" as object);
```

Notez que si au moins un argument de type ne peut être déduit, tous doivent être spécifiés.

Considérons la méthode générique suivante. Le premier argument de type générique est le même que le type de l'argument formel. Mais il n'y a pas une telle relation pour le second argument de type générique. Par conséquent, le compilateur n'a aucun moyen d'inférer le deuxième argument de type générique dans aucun appel à cette méthode.

```
void X<T1, T2>(T1 obj)
{
```

```
}
```

Cela ne fonctionne plus:

```
X("");
```

Cela ne fonctionne pas non plus, car le compilateur ne sait pas si nous spécifions le premier ou le deuxième paramètre générique (les deux seraient valables comme `object`):

```
X<object>("");
```

Nous sommes tenus de les taper tous les deux, comme ceci:

```
X<string, object>("");
```

Contraintes de type (classes et interfaces)

Les contraintes de type peuvent forcer un paramètre de type à implémenter une certaine interface ou classe.

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
```

```
new NonGeneric().DoSomething(new AnotherType());
```

Syntaxe pour plusieurs contraintes:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

Les contraintes de type fonctionnent de la même manière que l'héritage, car il est possible de spécifier plusieurs interfaces en tant que contraintes sur le type générique, mais une seule classe:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

Une autre règle est que la classe doit être ajoutée en tant que première contrainte, puis les interfaces:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

Toutes les contraintes déclarées doivent être satisfaites simultanément pour qu'une instanciation générique particulière fonctionne. Il n'y a aucun moyen de spécifier deux ou plusieurs autres ensembles de contraintes.

Contraintes de type (class et struct)

Il est possible de spécifier si l'argument de type doit ou non être un type de référence ou un type de valeur en utilisant la `class` ou la `struct` contraintes correspondante. Si ces contraintes sont utilisées, elles *doivent* être définies *avant que toutes les* autres contraintes (par exemple, un type parent ou `new()`) puissent être répertoriées.

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

Contraintes de type (new-mot-clé)

En utilisant la contrainte `new()` , il est possible d'imposer des paramètres de type pour définir un constructeur vide (par défaut).

```
class Foo
{
    public Foo () { }
}

class Bar
{
    public Bar (string s) { ... }
}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty
constructor.
```

Le deuxième appel à `Create()` donnera une erreur de compilation avec le message suivant:

> 'Bar' doit être un type non abstrait avec un constructeur public sans paramètre afin de l'utiliser comme paramètre 'T' dans le type générique ou la méthode 'Factory'

Il n'y a pas de contrainte pour un constructeur avec des paramètres, seuls les constructeurs sans paramètre sont pris en charge.

Type d'inférence (classes)

Les développeurs peuvent être surpris par le fait que l'inférence de type *ne fonctionne pas* pour les constructeurs:

```
class Tuple<T1,T2>
{
   public Tuple(T1 value1, T2 value2)
   {
   }
}

var x = new Tuple(2, "two");              // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

La première manière de créer une instance sans spécifier explicitement les paramètres de type entraînera une erreur de compilation qui indiquerait:

> L'utilisation du type générique 'Tuple <T1, T2>' nécessite 2 arguments de type

Une solution commune consiste à ajouter une méthode d'assistance dans une classe statique:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
         return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two");  // This WILL work...
```

Réflexion sur les paramètres de type

L'opérateur `typeof` fonctionne sur les paramètres de type.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Paramètres de type explicites

Il existe différents cas où vous devez spécifier explicitement les paramètres de type pour une méthode générique. Dans les deux cas ci-dessous, le compilateur ne peut pas déduire tous les paramètres de type des paramètres de méthode spécifiés.

Un cas est quand il n'y a pas de paramètres:

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
```

```
}

SomeMethod(); // doesn't compile
SomeMethod<int, bool>(); // compiles
```

Le deuxième cas est celui où un (ou plusieurs) des paramètres de type ne fait pas partie des paramètres de la méthode:

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

Utiliser une méthode générique avec une interface en tant que type de contrainte.

Voici un exemple d'utilisation du type générique TFood dans la méthode Eat sur la classe Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
```

```
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

Vous pouvez appeler la méthode Eat comme ceci:

```
var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore
```

Dans ce cas, si vous essayez d'appeler:

```
sheep.Eat(lion);
```

Cela ne sera pas possible car l'objet lion n'implémente pas l'interface IFood. Tenter de faire l'appel ci-dessus générera une erreur de compilation: "Le type 'Carnivore' ne peut pas être utilisé comme paramètre de type 'TFood' dans le type générique ou la méthode 'Animal.Eat (TFood)'. Carnivore 'à' IFood '. "

Covariance

Quand un `IEnumerable<T>` un sous-type d'un autre `IEnumerable<T1>` ? Lorsque `T` est un sous-type de `T1` . `IEnumerable` est *covariant* dans son paramètre `T` , ce qui signifie que la relation de sous-type d' `IEnumerable` va dans *le même sens* que celle de `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>
```

Une instance d'un type générique covariant avec un paramètre de type donné est implicitement convertible dans le même type générique avec un paramètre de type moins dérivé.

Cette relation est valable car `IEnumerable` *produit des* `T` s mais ne les consomme pas. Un objet qui produit un `Dog` peut être utilisé comme s'il produisait un `Animal` .

Les paramètres de type de covariant sont déclarés à l'aide du mot clé `out` , car le paramètre ne

doit être utilisé qu'en tant que *sortie* .

```
interface IEnumerable<out T> { /* ... */ }
```

Un paramètre de type déclaré comme covariant peut ne pas apparaître comme une entrée.

```
interface Bad<out T>
{
    void SetT(T t);  // type error
}
```

Voici un exemple complet:

```
using NUnit.Framework;

namespace ToyStore
{
   enum Taste { Bitter, Sweet };

   interface IWidget
   {
      int Weight { get; }
   }

   interface IFactory<out TWidget>
       where TWidget : IWidget
   {
      TWidget Create();
   }

   class Toy : IWidget
   {
      public int Weight { get; set; }
      public Taste Taste { get; set; }
   }

   class ToyFactory : IFactory<Toy>
   {
      public const int StandardWeight = 100;
      public const Taste StandardTaste = Taste.Sweet;

      public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste };
}
   }

   [TestFixture]
   public class GivenAToyFactory
   {
      [Test]
      public static void WhenUsingToyFactoryToMakeWidgets()
      {
         var toyFactory = new ToyFactory();

         //// Without out keyword, note the verbose explicit cast:
         // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

         // covariance: concrete being assigned to abstract (shiny and new)
         IFactory<IWidget> widgetFactory = toyFactory;
```

```
         IWidget anotherToy = widgetFactory.Create();
         Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
         Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); //
concrete contract
      }
   }
}
```

Contravariance

Quand un `IComparer<T>` un sous-type d'un autre `IComparer<T1>` ? Lorsque `T1` est un sous-type de `T` `IComparer` est *contravariant* dans son paramètre `T` , ce qui signifie que la relation de sous-type d' `IComparer` va dans la *direction opposée à* celle de `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

Une instance d'un type générique contravariant avec un paramètre de type donné est implicitement convertible dans le même type générique avec un paramètre de type plus dérivé.

Cette relation est valable car `IComparer` *consomme* `T` s mais ne les produit pas. Un objet qui peut comparer deux `Animal` peut être utilisé pour comparer deux `Dog` .

Les paramètres de type contravariant sont déclarés à l'aide du mot-clé `in` , car le paramètre ne doit être utilisé qu'en *entrée* .

```
interface IComparer<in T> { /* ... */ }
```

Un paramètre de type déclaré comme contravariant peut ne pas apparaître comme une sortie.

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

Invariance

`IList<T>` n'est jamais un sous-type d'un `IList<T1>` différent `IList<T1>` . `IList` est *invariant* dans son paramètre type.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

Il n'y a pas de relation de sous-type pour les listes car vous pouvez mettre des valeurs dans une liste *et* en extraire des valeurs.

Si `IList` était covariant, vous pourriez ajouter des éléments du *sous-type incorrect* à une liste donnée.

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

Si `IList` était contravariant, vous pourriez extraire des valeurs du sous-type incorrect d'une liste donnée.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() };  // if this were allowed...
Dog dog = dogs[1];  // ... then this would be allowed, which is bad!
```

Les paramètres de type invariant sont déclarés en omettant les mots-clés `in` et `out` .

```
interface IList<T> { /* ... */ }
```

Interfaces de variantes

Les interfaces peuvent avoir des paramètres de type variant.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

mais les classes et les structures ne peuvent pas

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
}
```

ni faire des déclarations de méthodes génériques

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
    {
        // ...
    }
}
```

L'exemple ci-dessous montre plusieurs déclarations de variance sur la même interface

```
interface IFoo<in T1, out T2, T3>
//  T1 : Contravariant type
//  T2 : Covariant type
//  T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

Variantes de délégués

Les délégués peuvent avoir des paramètres de type variant.

```
delegate void Action<in T>(T t);    // T is an input
delegate T Func<out T>();           // T is an output
delegate T2 Func<in T1, out T2>();  // T1 is an input, T2 is an output
```

Cela découle du principe de substitution de Liskov , qui stipule (entre autres) qu'une méthode D peut être considérée comme plus dérivée qu'une méthode B si:

- D a un type de retour dérivé égal ou supérieur à B
- D a des types de paramètres correspondants égaux ou plus généraux que B

Par conséquent, les affectations suivantes sont toutes de type sécurisé:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

Types de variantes en tant que paramètres et valeurs de retour

Si un type de covariant apparaît en sortie, le type contenant est covariant. Produire un producteur de `T` est comme produire des `T`

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

Si un type contravariant apparaît en tant que sortie, le type contenant est contravariant. Produire un consommateur de `T` s, c'est comme consommer des `T` s.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

Si un type de covariant apparaît en entrée, le type contenant est contravariant. Consommer un producteur de `T` est comme consommer des `T`

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

Si un type contravariant apparaît en entrée, le type contenant est covariant. Consommer un consommateur de `T` est comme produire des `T`

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Vérification de l'égalité des valeurs génériques.

Si la logique de la classe ou de la méthode générique nécessite de vérifier l'égalité des valeurs de type générique, utilisez la propriété `EqualityComparer<TType>.Default` :

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
}
```

Cette approche est mieux que d'appeler `Object.Equals()` méthode, parce que les contrôles de mise en œuvre de comparateur par défaut, si `TBar` Type implémente `IEquatale<TBar>` Interface et si oui, appelle `IEquatable<TBar>.Equals(TBar other)` méthode. Cela permet d'éviter la boxe / unboxing des types de valeur.

Casting de type générique

```
    /// <summary>
    /// Converts a data type to another data type.
    /// </summary>
    public static class Cast
    {
        /// <summary>
        /// Converts input to Type of default value or given as typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input, T defaultValue)
```

```
        {
            var result = defaultValue;
            try
            {
                if (input == null || input == DBNull.Value) return result;
                if (typeof (T).IsEnum)
                {
                    result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
                }
                else
                {
                    result = (T) Convert.ChangeType(input, typeof (T));
                }
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }

        /// <summary>
        /// Converts input to Type of typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input)
        {
            return To(input, default(T));
        }



    }
```

Coutumes:

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);


// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

Lecteur de configuration avec conversion de type générique

```
    /// <summary>
    /// Read configuration values from app.config and convert to specified types
    /// </summary>
    public static class ConfigurationReader
    {
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>AppSettings value against key is returned in Type of default value or
given as typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (ConfigurationManager.AppSettings[strKey] != null)
                    result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <returns>AppSettings value against key is returned in Type given as typeparam
T</returns>
        public static T GetConfigKeyValue<T>(string strKey)
        {
            return GetConfigKeyValue(strKey, default(T));
        }

    }
```

Coutumes:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Lire Génériques en ligne: https://riptutorial.com/fr/csharp/topic/27/generiques

Chapitre 71: Gestion de FormatException lors de la conversion d'une chaîne en d'autres types

Examples

Conversion de chaîne en entier

Il existe différentes méthodes pour convertir explicitement une `string` en un `integer` , par exemple:

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

Mais toutes ces méthodes vont lancer une `FormatException` , si la chaîne d'entrée contient des caractères non numériques. Pour cela, nous devons écrire une gestion des exceptions supplémentaire (`try..catch`) pour les traiter dans de tels cas.

Explication avec des exemples:

Alors, laissez notre contribution être:

```
string inputString = "10.2";
```

Exemple 1: `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

***Remarque: Il en** va de même pour les autres méthodes mentionnées, à savoir - `Convert.ToInt16();` et `Convert.ToInt64();`*

Exemple 2: `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

Comment pouvons-nous contourner cela?

Comme indiqué précédemment, pour gérer les exceptions, nous avons généralement besoin d'un `try..catch` comme indiqué ci-dessous:

```
try
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}
```

Mais, en utilisant le `try..catch` partout ne sera pas une bonne pratique, et il peut y avoir des scénarios où l' on a voulu donner `0` si l'entrée est erronée, *(si nous suivons la méthode ci - dessus , nous devons attribuer `0` à `convertedInt` de la bloc de capture).* Pour gérer de tels scénarios, nous pouvons utiliser une méthode spéciale appelée `.TryParse()` .

La méthode `.TryParse()` ayant une gestion interne des exceptions, qui vous donnera le résultat du paramètre `out` et renvoie une valeur booléenne indiquant le statut de la conversion *(`true` si la conversion a réussi; `false` si elle a échoué).* Sur la base de la valeur de retour, nous pouvons déterminer le statut de conversion. Voyons un exemple:

Utilisation 1: Stocke la valeur de retour dans une variable booléenne

```
 int convertedInt; // Be the required integer
 bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

Nous pouvons vérifier la variable `isSuccessConversion` après l'exécution pour vérifier l'état de la conversion. S'il est faux, la valeur de `convertedInt` sera `0` *(pas besoin de vérifier la valeur de retour si vous voulez `0` pour un échec de conversion).*

Usage 2: Vérifiez la valeur de retour avec `if`

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

Utilisation 3: Sans vérifier la valeur de retour, vous pouvez utiliser ce qui suit si vous ne vous souciez pas de la valeur de retour *(converti ou non, `0` sera correct)*

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

Lire Gestion de FormatException lors de la conversion d'une chaîne en d'autres types en ligne: https://riptutorial.com/fr/csharp/topic/2886/gestion-de-formatexception-lors-de-la-conversion-d-une-chaine-en-d-autres-types

Chapitre 72: Gestion des exceptions

Examples

Gestion des exceptions de base

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Notez que la gestion de toutes les exceptions avec le même code n'est souvent pas la meilleure approche.
Ceci est couramment utilisé lorsque toutes les routines de gestion des exceptions internes échouent, en dernier recours.

Gestion de types d'exceptions spécifiques

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

Veillez à ce que les exceptions soient évaluées dans l'ordre et l'héritage est appliqué. Vous devez donc commencer par les plus spécifiques et terminer par leur ancêtre. A tout moment, un seul bloc catch sera exécuté.

Utiliser l'objet exception

Vous êtes autorisé à créer et à lancer des exceptions dans votre propre code. L'instanciation d'une exception se fait de la même manière que tout autre objet C #.

```
Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

Vous pouvez ensuite utiliser le mot-clé `throw` pour déclencher l'exception:

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

Remarque: Si vous lancez une nouvelle exception dans un bloc catch, vérifiez que l'exception d'origine est transmise en tant qu "'exception interne", par exemple

```
void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.Write(ex.ToString());
    }
}
```

Dans ce cas, il est supposé que l'exception ne peut pas être gérée mais certaines informations utiles sont ajoutées au message (et l'exception d'origine peut toujours être accédée via `ex.InnerException` par un bloc d'exception externe).

Il montrera quelque chose comme:

> System.DivideByZeroException: impossible de diviser par b car il est zéro --->
> System.DivideByZeroException: tentative de division par zéro.
> à UserQuery.g__DoSomething0_0 () dans C: [...] \ LINQPadQuery.cs: ligne 36
> --- Fin de trace de pile d'exception interne ---
> à UserQuery.g__DoSomething0_0 () dans C: [...] \ LINQPadQuery.cs: ligne 42
> à UserQuery.Main () dans C: [...] \ LINQPadQuery.cs: ligne 55

Si vous essayez cet exemple dans LinqPad, vous remarquerez que les numéros de ligne ne sont pas très significatifs (ils ne vous aident pas toujours). Cependant, le fait de transmettre un texte d'erreur utile, comme suggéré ci-dessus, réduit considérablement le temps nécessaire pour localiser l'erreur, ce qui est clairement dans cet exemple la ligne

> c = a / b;

dans la fonction `DoSomething()` .

Essayez-le avec .NET Fiddle

Bloquer enfin

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

Le bloc `try / catch / finally` peut être très utile lors de la lecture de fichiers.
Par exemple:

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}
```

Un bloc try doit être suivi soit d'un bloc `catch` soit d'un bloc `finally` . Cependant, comme il n'y a pas de bloc catch, l'exécution provoquera une terminaison. Avant la fin, les instructions contenues dans le bloc finally seront exécutées.

Dans la lecture de fichiers, nous aurions pu utiliser un bloc `using` `FileStream` (ce `OpenRead` renvoie `OpenRead`) implémente `IDisposable` .

Même s'il existe une instruction de `return` dans le bloc `try` , le bloc `finally` sera généralement exécuté; il y a quelques cas où cela ne va pas:

- Lorsqu'un StackOverflow se produit .
- `Environment.FailFast`
- Le processus d'application est interrompu, généralement par une source externe.

Implémentation d'IErrorHandler pour les services WCF

L'implémentation d'IErrorHandler pour les services WCF est un excellent moyen de centraliser la gestion des erreurs et la journalisation. L'implémentation présentée ici doit intercepter toute exception non gérée à la suite d'un appel à l'un de vos services WCF. Cet exemple montre également comment renvoyer un objet personnalisé et comment renvoyer JSON plutôt que le code XML par défaut.

Implémenter IErrorHandler:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
```

```
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new
List<Type> { typeof(BaseDataResponseContract) }));

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process
them differently
            }

            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name,
webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

Dans cet exemple, nous attachons le gestionnaire au comportement du service. Vous pouvez également associer ceci à IEndpointBehavior, IContractBehavior ou IOperationBehavior de la même manière.

Attacher aux comportements de service:

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
BindingParameterCollection bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
```

```
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace
```

Configs dans Web.config:

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

Voici quelques liens utiles sur ce sujet:

https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/qui-four-behavior-interfaces-exist-for-interact-with-a-service-or-client-description-what-methods-do-they- mettre en œuvre et

Autres exemples:

IErrorHandler renvoie un corps de message incorrect lorsque le code d'état HTTP est 401 Non autorisé

IErrorHandler ne semble pas gérer mes erreurs dans WCF .. des idées?

Comment faire pour que le gestionnaire d'erreurs WCF personnalisé renvoie une réponse JSON avec un code HTTP non OK?

Comment définissez-vous l'en-tête Content-Type pour une requête HttpClient?

Création d'exceptions personnalisées

Vous êtes autorisé à implémenter des exceptions personnalisées pouvant être lancées comme toute autre exception. Cela a du sens lorsque vous voulez distinguer vos exceptions des autres erreurs lors de l'exécution.

Dans cet exemple, nous allons créer une exception personnalisée pour gérer clairement les problèmes éventuels de l'application tout en analysant une entrée complexe.

Création d'une classe d'exception personnalisée

Pour créer une exception personnalisée, créez une sous-classe d' `Exception` :

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

Les exceptions personnalisées deviennent très utiles lorsque vous souhaitez fournir des informations supplémentaires au receveur:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
      base($"Parser error in {fileName}:{lineNumber}")
    {
      FileName = fileName;
      LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Maintenant, quand vous `catch(ParserException x)` vous aurez une sémantique supplémentaire pour affiner la gestion des exceptions.

Les classes personnalisées peuvent implémenter les fonctionnalités suivantes pour prendre en charge des scénarios supplémentaires.

lancer à nouveau

Pendant le processus d'analyse, l'exception d'origine est toujours intéressante. Dans cet exemple, il s'agit d'une `FormatException` car le code tente d'analyser un morceau de chaîne, qui devrait être un nombre. Dans ce cas, l'exception personnalisée devrait prendre en charge l'inclusion de « **InnerException** »:

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

sérialisation

Dans certains cas, vos exceptions peuvent avoir à franchir les limites de AppDomain. C'est le cas si votre analyseur s'exécute dans son propre AppDomain pour prendre en charge le rechargement à chaud de nouvelles configurations d'analyseur. Dans Visual Studio, vous pouvez utiliser le modèle `Exception` pour générer du code comme celui-ci.

```
[Serializable]
public class ParserException : Exception
{
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
    // generic automatically generated messages.
    public ParserException(string message)
        : base(message)
    {}

    // Constructor for serialization support. If your exception contains custom
    // properties, read their values here.
    protected ParserException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {}
}
```

Utiliser l'Exception Parser

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

Vous pouvez également utiliser des exceptions personnalisées pour intercepter et encapsuler des exceptions. De cette façon, de nombreuses erreurs différentes peuvent être converties en un seul type d'erreur plus utile pour l'application:

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
      $"Failed to read {token} as number.",
      FileName,
      LineNumber,
      ex);
}
```

Lorsque vous gérez des exceptions en levant vos propres exceptions personnalisées, vous devez généralement inclure une référence à l'exception d'origine dans la `InnerException` , comme indiqué ci-dessus.

Problèmes de sécurité

Si exposer la raison de l'exception peut compromettre la sécurité en permettant aux utilisateurs de voir le fonctionnement interne de votre application, il peut être une mauvaise idée d'envelopper l'exception interne. Cela peut s'appliquer si vous créez une bibliothèque de classes qui sera utilisée par d'autres.

Voici comment vous pouvez générer une exception personnalisée sans encapsuler l'exception interne:

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
```

```
}
```

Conclusion

Lorsque vous déclenchez une exception personnalisée (avec une nouvelle enveloppe ou une nouvelle exception non emballée), vous devez générer une exception significative pour l'appelant. Par exemple, un utilisateur d'une bibliothèque de classes peut ne pas savoir comment cette bibliothèque exécute son travail interne. Les exceptions levées par les dépendances de la bibliothèque de classes ne sont pas significatives. Au contraire, l'utilisateur souhaite une exception qui concerne la manière dont la bibliothèque de classes utilise ces dépendances de manière erronée.

```
try
{
  // ...
}
catch (IOException ex)
{
  // ...
  throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

Exception Anti-patrons

Exceptions à la déglutition

Il faut toujours relancer l’exception de la manière suivante:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Relancer une exception comme ci-dessous masquera l'exception d'origine et perdra la trace de la pile d'origine. On ne devrait jamais faire ça! La trace de la pile avant la capture et la relance sera perdue.

```
try
{
    ...
}
```

```
catch (Exception ex)
{
    ...
    throw ex;
}
```

Gestion des exceptions de baseball

Il ne faut pas utiliser les exceptions comme substitut aux constructions de contrôle de flux normales comme les instructions if-then et les boucles while. Cet anti-pattern est parfois appelé gestion des exceptions de baseball .

Voici un exemple d'anti-pattern:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Voici une meilleure façon de le faire:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.Write("Here are your account details: " + found.Details.ToString());
```

attraper (Exception)

Il n'y a presque pas (certaines ne disent rien!) De raisons pour intercepter le type d'exception générique dans votre code. Vous ne devriez intercepter que les types d'exception que vous prévoyez, car vous masquez autrement les bogues dans votre code.

```
try
{
     var f = File.Open(myfile);
     // do something
}
catch (Exception x)
{
     // Assume file not found
     Console.Write("Could not open file");
     // but maybe the error was a NullReferenceException because of a bug in the file handling
code?
}
```

Mieux vaut faire:

```
try
{
     var f = File.Open(myfile);
     // do something which should normally not throw exceptions
}
catch (IOException)
{
     Console.Write("File not found");
}
// Unfortunatelly, this one does not derive from the above, so declare separatelly
catch (UnauthorizedAccessException)
{
     Console.Write("Insufficient rights");
}
```

Si une autre exception se produit, nous autorisons délibérément l'application à tomber en panne, elle intervient donc directement dans le débogueur et nous pouvons résoudre le problème. Nous ne devons pas expédier un programme où toute autre exception que celle-ci se produit de toute façon, donc ce n'est pas un problème d'avoir un crash.

Ce qui suit est un mauvais exemple, car il utilise des exceptions pour contourner une erreur de programmation. Ce n'est pas pour ça qu'ils sont conçus.

```
public void DoSomething(String s)
{
     if (s == null)
         throw new ArgumentNullException(nameof(s));
     // Implementation goes here
}

try
{
     DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

Exceptions globales / exceptions multiples d'une méthode

Qui dit que vous ne pouvez pas lancer plusieurs exceptions dans une méthode. Si vous n'avez pas l'habitude de jouer avec AggregateExceptions, vous pouvez être tenté de créer votre propre structure de données pour représenter de nombreuses erreurs. Il y a bien sûr une autre structure de données qui n'est pas une exception serait plus idéale comme les résultats d'une validation. Même si vous jouez avec AggregateExceptions, vous êtes peut-être du côté de la réception et vous les manipulez toujours sans vous rendre compte qu'elles peuvent vous être utiles.

Il est tout à fait plausible de faire exécuter une méthode et même si cela sera un échec dans son ensemble, vous voudrez mettre en évidence plusieurs choses qui ont mal tourné dans les exceptions lancées. A titre d'exemple, ce comportement peut être vu avec le fonctionnement des méthodes Parallel, une tâche divisée en plusieurs threads et un nombre quelconque d'entre elles pouvant générer des exceptions et cela doit être signalé. Voici un exemple stupide de la façon dont vous pourriez en bénéficier:

```
    public void Run()
    {
        try
        {
            this.SillyMethod(1, 2);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine(ex.Message);
            foreach (Exception innerException in ex.InnerExceptions)
            {
                Console.WriteLine(innerException.Message);
            }
        }
    }

    private void SillyMethod(int input1, int input2)
    {
        var exceptions = new List<Exception>();

        if (input1 == 1)
        {
            exceptions.Add(new ArgumentException("I do not like ones"));
        }
        if (input2 == 2)
        {
            exceptions.Add(new ArgumentException("I do not like twos"));
        }
        if (exceptions.Any())
        {
            throw new AggregateException("Funny stuff happended during execution",
exceptions);
        }
    }
```

Nidification des exceptions et essayez les blocs de capture.

On peut imbriquer une exception / `try` bloc `catch` dans l'autre.

De cette façon, vous pouvez gérer de petits blocs de code capables de fonctionner sans perturber l’ensemble de votre mécanisme.

```
try
{
//some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
 //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}
```

Remarque: évitez les exceptions de déglutition lors du lancement sur le bloc catch parent

Les meilleures pratiques

Cheatsheet

FAIRE	Ne pas
Contrôle du flux avec des instructions de contrôle	Contrôle des flux avec des exceptions
Suivre les exceptions ignorées (absorbées) en se connectant	Ignorer l'exception
Répétez l'exception en utilisant `throw`	Re-jeter une exception - `throw new ArgumentNullException()` ou `throw ex`
Lancer des exceptions système prédéfinies	Jeter des exceptions personnalisées similaires aux exceptions système prédéfinies
Jeter une exception personnalisée / prédéfinie si elle est cruciale pour la logique de l'application	Lancer des exceptions personnalisées / prédéfinies pour indiquer un avertissement dans le flux
Catch exceptions que vous souhaitez gérer	Attraper toutes les exceptions

NE PAS gérer la logique métier avec des exceptions.

Le contrôle de flux ne doit PAS être effectué par des exceptions. Utilisez plutôt des instructions conditionnelles. Si un contrôle peut être effectué avec l'instruction `if-else` clairement, n'utilisez pas les exceptions car cela réduit la lisibilité et les performances.

Considérez l'extrait suivant de Mr. Bad Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

Lorsque l'exécution atteint `Console.WriteLine(myObject.ToString());` l'application lancera une exception NullReferenceException. M. Bad Practices s'est rendu compte que `myObject` était nul et a modifié son extrait pour intercepter et gérer `NullReferenceException` :

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

Comme l'extrait de `myObject` précédent ne couvre que la logique d'exception, que dois-je faire si `myObject` n'est pas nul à ce stade? Où devrais-je couvrir cette partie de la logique? Juste après `Console.WriteLine(myObject.ToString());` ? Qu'en est-il après le `try...catch` block?

Qu'en est-il de Mr. Best Practices? Comment va-t-il gérer ça?

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

M. Best Practices a atteint la même logique avec moins de code et une logique claire et compréhensible.

NE PAS rejeter les exceptions

Relancer les exceptions est coûteux. Cela a un impact négatif sur les performances. Pour le code qui échoue régulièrement, vous pouvez utiliser des modèles de conception pour minimiser les problèmes de performances. Cette rubrique décrit deux modèles de conception utiles lorsque des exceptions peuvent avoir un impact significatif sur les performances.

NE PAS absorber les exceptions sans enregistrement

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

Ne jamais avaler les exceptions. Ignorer les exceptions sauvera ce moment mais créera un chaos pour la maintenabilité plus tard. Lors de la journalisation des exceptions, vous devez toujours consigner l'instance d'exception afin que la trace complète de la pile soit consignée et non uniquement le message d'exception.

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

Ne pas intercepter les exceptions que vous ne pouvez pas gérer

De nombreuses ressources, telles que celle-ci , vous invitent fortement à réfléchir aux raisons pour lesquelles vous attrapez une exception à l'endroit où vous la capturez. Vous ne devriez attraper une exception que si vous pouvez la gérer à cet endroit. Si vous pouvez faire quelque chose pour atténuer le problème, par exemple essayer un autre algorithme, vous connecter à une base de données de sauvegarde, essayer un autre nom de fichier, attendre 30 secondes et réessayer ou avertir un administrateur, vous pouvez détecter l'erreur. S'il n'y a rien que vous puissiez faire de manière plausible et raisonnable, il suffit de "laisser aller" et de laisser l'exception être traitée à un niveau supérieur. Si l'exception est suffisamment catastrophique et qu'il n'y a pas d'autre option raisonnable que le blocage complet du programme en raison de la gravité du problème, laissez-le tomber en panne.

```
try
{
```

```
    //Try to save the data to the main database.
}
catch(SqlException ex)
{
    //Try to save the data to the alternative database.
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the
exception bubble up to a level where it can be handled.
```

Exception non gérée et thread

AppDomain.UnhandledException Cet événement fournit une notification des exceptions non interceptées. Il permet à l'application de consigner des informations sur l'exception avant que le gestionnaire par défaut du système ne signale l'exception à l'utilisateur et qu'il mette fin à l'application. des actions peuvent être entreprises - telles que la sauvegarde des données du programme pour une récupération ultérieure. Une mise en garde est recommandée, car les données du programme peuvent être corrompues lorsque des exceptions ne sont pas gérées.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

Application.ThreadException Cet événement permet à votre application Windows Forms de gérer les exceptions non gérées qui se produisent dans les threads Windows Forms. Attachez vos gestionnaires d'événements à l'événement ThreadException pour gérer ces exceptions, ce qui laissera votre application dans un état inconnu. Dans la mesure du possible, les exceptions doivent être gérées par un bloc de gestion des exceptions structuré.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
        Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
    }
```

Et enfin la gestion des exceptions

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (Exception)e.ExceptionObject;
        // your code
    }
```

```
static void ThreadException(object sender, ThreadExceptionEventArgs e)
    {
        Exception ex = e.Exception;
        // your code
    }
```

Lancer une exception

Votre code peut, et devrait souvent, lancer une exception lorsque quelque chose d'inhabituel s'est produit.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/csharp/topic/40/gestion-des-exceptions

Chapitre 73: Gestionnaire d'authentification C

Examples

Gestionnaire d'authentification

```
 public class AuthenticationHandler : DelegatingHandler
    {
        /// <summary>
        /// Holds request's header name which will contains token.
        /// </summary>
        private const string securityToken = "__RequestAuthToken";

        /// <summary>
        /// Default overridden method which performs authentication.
        /// </summary>
        /// <param name="request">Http request message.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/>
class asynchronously.</returns>
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
        {
            if (request.Headers.Contains(securityToken))
            {
                bool authorized = Authorize(request);
                if (!authorized)
                {
                    return ApiHttpUtility.FromResult(request, false,
HttpStatusCode.Unauthorized, MessageTypes.Error, Resource.UnAuthenticatedUser);
                }
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
MessageTypes.Error, Resource.UnAuthenticatedUser);
            }

            return base.SendAsync(request, cancellationToken);
        }

        /// <summary>
        /// Authorize user by validating token.
        /// </summary>
        /// <param name="requestMessage">Authorization context.</param>
        /// <returns>Returns a value indicating whether current request is authenticated or
not.</returns>
        private bool Authorize(HttpRequestMessage requestMessage)
        {
            try
            {
                HttpRequest request = HttpContext.Current.Request;
                string token = request.Headers[securityToken];
                return SecurityUtility.IsTokenValid(token, request.UserAgent,
```

```
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
            }
            catch (Exception)
            {
                return false;
            }
        }
    }
```

Lire Gestionnaire d'authentification C # en ligne:
https://riptutorial.com/fr/csharp/topic/5430/gestionnaire-d-authentification-c-sharp

Chapitre 74: Guid

Introduction

GUID (ou UUID) est un acronyme pour «Globally Unique Identifier» (ou «Universally Unique Identifier»). C'est un nombre entier de 128 bits utilisé pour identifier les ressources.

Remarques

`Guid` sont des *identificateurs globaux uniques* , également appelés *identificateurs universels uniques (UUID*).

Ce sont des valeurs pseudo-aléatoires de 128 bits. Il y a tellement de `Guid` valides (environ 10 ^ 18 `Guid` pour chaque cellule de chaque peuple sur Terre) que s'ils sont générés par un bon algorithme pseudo-aléatoire, ils peuvent être considérés comme uniques dans tout l'univers par tous les moyens pratiques.

`Guid` sont le plus souvent utilisés comme clés primaires dans les bases de données. Leur avantage est que vous n'avez pas besoin d'appeler la base de données pour obtenir un nouvel identifiant qui est (presque) garanti unique.

Examples

Obtenir la représentation sous forme de chaîne d'un Guid

Une représentation sous forme de chaîne d'un Guid peut être obtenue à l'aide de la méthode `ToString` intégrée

```
string myGuidString = myGuid.ToString();
```

Selon vos besoins, vous pouvez également formater le Guid en ajoutant un argument de type de format à l'appel `ToString` .

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None          "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens       "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces        "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses   "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex           "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
```

```
Console.WriteLine(guid.ToString("X"));
```

Créer un guide

Ce sont les moyens les plus courants de créer une instance de Guid:

- Créer un guid vide (`00000000-0000-0000-0000-000000000000`):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Créer un nouveau (pseudo-aléatoire) Guid:

```
Guid g = Guid.NewGuid();
```

- Créer des guids avec une valeur spécifique:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Déclaration d'un GUID nullable

Comme d'autres types de valeur, GUID a également un type nullable qui peut prendre une valeur nulle.

Déclaration:

```
Guid? myGuidVar = null;
```

Ceci est particulièrement utile lors de la récupération de données à partir de la base de données lorsqu'il est possible que la valeur d'une table soit NULL.

Lire Guid en ligne: https://riptutorial.com/fr/csharp/topic/1153/guid

Chapitre 75: Héritage

Syntaxe

- class DerivedClass: BaseClass
- class DerivedClass: BaseClass, IExampleInterface
- class DerivedClass: BaseClass, IExampleInterface, IAnotherInterface

Remarques

Les classes peuvent hériter directement d'une seule classe, mais (ou à la fois) peuvent implémenter une ou plusieurs interfaces.

Les structures peuvent implémenter des interfaces mais ne peuvent pas explicitement hériter d'aucun type. Ils héritent implicitement de `System.ValueType` , qui à son tour hérite directement de `System.Object` .

Les classes statiques ne peuvent pas implémenter des interfaces.

Examples

Hériter d'une classe de base

Pour éviter la duplication de code, définissez des méthodes et des attributs communs dans une classe générale en tant que base:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Maintenant que vous avez une classe qui représente `Animal` en général, vous pouvez définir une classe décrivant les particularités de certains animaux:

```
public class Cat : Animal
```

```
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}
```

La classe Cat accède non seulement aux méthodes décrites explicitement dans sa définition, mais aussi à toutes les méthodes définies dans la classe de base `Animal` . Tout animal (chat ou non) pouvait manger, regarder ou rouler. Un animal ne serait pas en mesure de gratter, à moins que ce ne soit également un chat. Vous pouvez ensuite définir d'autres classes décrivant d'autres animaux. (Comme Gopher avec une méthode pour détruire les jardins de fleurs et la paresse sans aucune méthode supplémentaire.)

Héritage d'une classe et implémentation d'une interface

```
public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Hériter d'une classe et implémenter plusieurs interfaces

```
public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
```

```
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Tester et naviguer dans l'héritage

```
interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name);  //BaseClass
Console.WriteLine(baseType.BaseType.Name);  //Object
Console.WriteLine(typeof(object).BaseType);  //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance));  //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance));  //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface
```

```
Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

Extension d'une classe de base abstraite

Contrairement aux interfaces, qui peuvent être décrites comme des contrats d'implémentation, les classes abstraites servent de contrats d'extension.

Une classe abstraite ne peut pas être instanciée, elle doit être étendue et la classe résultante (ou la classe dérivée) peut être instanciée.

Les classes abstraites sont utilisées pour fournir des implémentations génériques

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

L'exemple ci-dessus montre comment toute classe qui étend sa voiture recevra automatiquement la méthode HonkHorn avec l'implémentation. Cela signifie que tout développeur qui crée une nouvelle voiture n'aura pas à se soucier de la façon dont il va klaxonner.

Constructeurs dans une sous-classe

Lorsque vous créez une sous-classe d'une classe de base, vous pouvez construire la classe de base en utilisant `: base` après les paramètres du constructeur de la sous-classe.

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;
```

```
    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

Héritage. Séquence d'appels des constructeurs

Considérons que nous avons un `Animal` classe qui a un `Dog` classe enfant

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

Par défaut, chaque classe hérite implicitement de la classe `Object` .

Ceci est identique au code ci-dessus.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

Lors de la création d'une instance de classe `Dog` , le **constructeur par défaut des classes de base (sans paramètres) sera appelé s'il n'y a pas d'appel explicite à un autre constructeur de la classe parent** . Dans notre cas, nous appellerons d'abord `Object's` constructeur `Object's` , puis `Object's` constructeur de `Animal's` et à la fin de `Dog's` .

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

La sortie sera

> Dans le constructeur d'Animal

Dans le constructeur de Dog

Voir la démo

Appelez explicitement le constructeur du parent.

Dans les exemples ci-dessus, notre constructeur de classe `Dog` appelle le constructeur **par défaut** de la classe `Animal` . Si vous le souhaitez, vous pouvez spécifier quel constructeur doit être appelé: il est possible d'appeler n'importe quel constructeur défini dans la classe parente.

Considérez que nous avons ces deux classes.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

Qu'est-ce qui se passe ici?

Nous avons 2 constructeurs dans chaque classe.

Que signifie la `base` ?

`base` est une référence à la classe parente. Dans notre cas, lorsque nous créons une instance de classe `Dog` comme celle-ci

```
Dog dog = new Dog();
```

Le runtime appelle d'abord le `Dog()` , qui est le constructeur sans paramètre. Mais son corps ne

fonctionne pas immédiatement. Après les parenthèses du constructeur, nous avons un tel appel: `base()` , ce qui signifie que lorsque nous appelons le constructeur par défaut de `Dog` , il appellera à son tour le constructeur **par défaut** du parent. Une fois le constructeur du parent exécuté, il retournera puis exécutera le corps du constructeur `Dog()` .

Donc, la sortie sera comme ceci:

> Constructeur par défaut de l'animal
> Constructeur par défaut du chien

Voir la démo

Maintenant, si on appelle le constructeur `Dog's` avec un paramètre?

```
Dog dog = new Dog("Rex");
```

Vous savez que les membres de la classe parent qui ne sont pas privés sont hérités par la classe enfant, ce qui signifie que `Dog` aura également le champ `name` .
Dans ce cas, nous avons passé un argument à notre constructeur. Il transmet à son tour l'argument au constructeur de la classe parente **avec un paramètre** qui initialise le champ de `name` .

La sortie sera

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

Résumé:

Chaque création d'objet commence à partir de la classe de base. Dans l'héritage, les classes de la hiérarchie sont chaînées. Comme toutes les classes dérivent d' `Object` , le constructeur de la classe `Object` est le premier constructeur à être appelé lorsqu'un objet est créé. Ensuite, le constructeur suivant dans la chaîne est appelé et seulement après que tous soient appelés, l'objet est créé

mot clé de base

1. Le mot-clé base est utilisé pour accéder aux membres de la classe de base à partir d'une classe dérivée:
2. Appelez une méthode sur la classe de base qui a été remplacée par une autre méthode. Indiquez quel constructeur de classe de base doit être appelé lors de la création d'instances de la classe dérivée.

Méthodes d'héritage

Il existe plusieurs manières d’hériter des méthodes

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }

    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

Héritage Anti-patrons

Héritage incorrect

Disons qu'il y a 2 classes classe `Foo` et `Bar` . `Foo` a deux fonctionnalités `Do1` et `Do2` . `Bar` doit utiliser `Do1` de `Foo` , mais elle ne nécessite pas `Do2` ou nécessite une fonctionnalité équivalente à `Do2` mais qui fait quelque chose de complètement différent.

Mauvaise manière : rendre `Do2()` sur `Foo` virtual, puis le remplacer dans `Bar` ou simplement `throw Exception` dans `Bar` pour `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

Bonne façon

Sortez `Do1()` de `Foo` et mettez-le dans la nouvelle classe `Baz` puis héritez de `Foo` et `Bar` de `Baz` et implémentez `Do2()` séparément

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Maintenant, pourquoi le premier exemple est mauvais et le second bon: quand le développeur nr2 doit faire un changement dans `Foo` , il est probable qu'il va casser l'implémentation de `Bar` parce que `Bar` est désormais indissociable de `Foo` . Par exemple, `Foo` and `Bar` commonalty a été déplacé dans `Baz` et ils ne s'affectent pas (comme dans le cas contraire).

Classe de base avec spécification de type récursif

Définition unique d'une classe de base générique avec spécificateur de type récursif. Chaque nœud a un parent et plusieurs enfants.

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }
```

```
    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }
    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}
```

Ce qui précède peut être réutilisé chaque fois qu'une hiérarchie d'arbres d'objets doit être définie. L'objet nœud dans l'arbre doit hériter de la classe de base avec

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

chaque classe de noeud sait où elle se trouve dans la hiérarchie, quel est l'objet parent et quels sont les objets enfants. Plusieurs types intégrés utilisent une structure arborescente, comme `Control` ou `XmlElement` et l' `Tree<T>` ci-dessus `Tree<T>` peut être utilisé comme classe de base de *n'importe quel* type dans votre code.

Par exemple, pour créer une hiérarchie d'éléments dont le poids total est calculé à partir du poids de *tous* les enfants, procédez comme suit:

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); }
}
}
```

à utiliser comme

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//    \
//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

Un autre exemple serait la définition des trames de coordonnées relatives. Dans ce cas, la position réelle du cadre de coordonnées dépend des positions de *tous* les cadres de coordonnées parents.

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null,
PointF.Empty) { };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}
```

à utiliser comme

```
// Define the following coordinate system hierarchy
//
// o--> [A1] --+--> [B1] -----> [C1]
//             |
//             +--> [B2] --+--> [C2]
//                         |
//                         +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);
```

```
double dist1 = C1.TotalDistance;
```

Lire Héritage en ligne: https://riptutorial.com/fr/csharp/topic/29/heritage

Chapitre 76: IClonable

Syntaxe

- object ICloneable.Clone () {return Clone (); } // Implémentation privée de la méthode d'interface qui utilise notre fonction publique personnalisée Clone ().
- public Foo Clone () {retourne le nouveau Foo (this); } // La méthode de clone publique doit utiliser la logique du constructeur de copie.

Remarques

Le `CLR` requiert un `object Clone()` définition de méthode `object Clone()` qui n'est pas sûr pour le type. Il est courant de remplacer ce comportement et de définir une méthode sécurisée de type qui renvoie une copie de la classe contenant.

Il appartient à l'auteur de décider si le clonage ne signifie qu'une copie superficielle ou une copie profonde. Pour les structures immuables contenant des références, il est recommandé de faire une copie en profondeur. Pour les classes qui sont elles-mêmes des références, il est probablement bon d'implémenter une copie superficielle.

REMARQUE: En `C#` une méthode d'interface peut être implémentée en privé avec la syntaxe indiquée ci-dessus.

Examples

Implémenter ICloneable dans une classe

Implémenter `ICloneable` dans une classe avec une torsion. Exposez un type sécurisé `Clone()` et implémentez l' `object Clone()` privé.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
```

```
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Plus tard pour être utilisé comme suit:

```
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}
```

Notez que changer l'âge de `bob` ne change pas l'âge de `bob_clone` . En effet, la conception utilise le clonage au lieu d’attribuer des variables (de référence).

Implémenter ICloneable dans une structure

L'implémentation d'ICloneable pour une structure n'est généralement pas nécessaire car les structs font une copie membre avec l'opérateur d'affectation `=` . Mais la conception peut nécessiter l'implémentation d'une autre interface qui hérite d' `ICloneable` .

Une autre raison serait que la structure contienne un type de référence (ou un tableau) nécessitant une copie également.

```
// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
```

```
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Plus tard pour être utilisé comme suit:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

Lire IClonable en ligne: https://riptutorial.com/fr/csharp/topic/7917/iclonable

Chapitre 77: IComparable

Examples

Trier les versions

Classe:

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[0-9]+(\.[0-9]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

Tester:

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
```

```
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");
```

Sortie:

NUL
1
1.0.1
1.1.5
2.0
3.0.10

Démo:

Démo en direct sur Ideone

Lire IComparable en ligne: https://riptutorial.com/fr/csharp/topic/4222/icomparable

Chapitre 78: Identité ASP.NET

Introduction

Tutoriels concernant asp.net Identity tels que la gestion des utilisateurs, la gestion des rôles, la création de jetons et bien plus.

Examples

Comment implémenter le jeton de réinitialisation de mot de passe dans l'identité asp.net à l'aide du gestionnaire d'utilisateurs.

1. Créez un nouveau dossier appelé MyClasses et créez et ajoutez la classe suivante

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Configurez votre classe d'identité

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com",
"(any subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Ajoutez vos informations d'identification au site Web.config. Je n'ai pas utilisé gmail dans cette partie car l'utilisation de gmail est bloquée sur mon lieu de travail et fonctionne toujours parfaitement.

```
<add key="GmailUserName" value="youremail@yourdomain.com"/>
<add key="GmailPassword" value="yourPassword"/>
<add key="GmailHost" value="yourServer"/>
<add key="GmailPort" value="yourPort"/>
<add key="GmailSsl" value="chooseTrueOrFalse"/>
<!--Smptp Server (confirmations emails)-->
```

4. Apportez les modifications nécessaires à votre contrôleur de compte. Ajoutez le code en surbrillance suivant.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();


        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext
        {
            _context = new ApplicationDbContext();

        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signIn
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));

        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
```

```
//
// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {

        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicki
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Compiler puis exécuter. À votre santé!

Lire Identité ASP.NET en ligne: https://riptutorial.com/fr/csharp/topic/9577/identite-asp-net

Chapitre 79: IEnumerable

Introduction

`IEnumerable` est l'interface de base pour toutes les collections non génériques comme ArrayList pouvant être énumérées. `IEnumerator<T>` est l'interface de base pour tous les énumérateurs génériques tels que List <>.

`IEnumerable` est une interface qui implémente la méthode `GetEnumerator` . La méthode `GetEnumerator` renvoie un `IEnumerator` qui fournit des options permettant de parcourir la collection comme foreach.

Remarques

IEnumerable est l'interface de base pour toutes les collections non génériques pouvant être énumérées

Examples

IEnumerable

Dans sa forme la plus élémentaire, un objet qui implémente IEnumerable représente une série d'objets. Les objets en question peuvent être itérés à l'aide du mot clé c # `foreach` .

Dans l'exemple ci-dessous, l'objet `sequenceOfNumbers` implémente IEnumerable. Il représente une série d'entiers. La boucle `foreach` parcourt chacun à son tour.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

IEnumerable avec un énumérateur personnalisé

L'implémentation de l'interface IEnumerable permet d'énumérer les classes de la même manière que les collections BCL. Cela nécessite l'extension de la classe Enumerator qui suit l'état de l'énumération.

Outre l'itération sur une collection standard, voici des exemples:

- Utilisation de plages de nombres basées sur une fonction plutôt que sur une collection d'objets
- Implémenter différents algorithmes d'itération sur des collections, comme DFS ou BFS sur une collection de graphes

```
public static void Main(string[] args) {

    foreach (var coffee in new CoffeeCollection()) {
        Console.WriteLine(coffee);
    }
}

public class CoffeeCollection : IEnumerable {
    private CoffeeEnumerator enumerator;

    public CoffeeCollection() {
        enumerator = new CoffeeEnumerator();
    }

    public IEnumerator GetEnumerator() {
        return enumerator;
    }

    public class CoffeeEnumerator : IEnumerator {
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };
        int currentIndex = -1;

        public object Current {
            get {
                return beverages[currentIndex];
            }
        }

        public bool MoveNext() {
            currentIndex++;

            if (currentIndex < beverages.Length) {
                return true;
            }

            return false;
        }

        public void Reset() {
            currentIndex = 0;
        }
    }
}
```

Lire IEnumerable en ligne: https://riptutorial.com/fr/csharp/topic/2220/ienumerable

Chapitre 80: ILGenerator

Examples

Crée un DynamicAssembly contenant une méthode d'assistance UnixTimestamp

Cet exemple montre comment utiliser ILGenerator en générant du code utilisant des membres existants et nouveaux, ainsi que la gestion des exceptions de base. Le code suivant émet un DynamicAssembly contenant un équivalent à ce code c #:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds =
checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's to late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubstract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubstract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an,
AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");
```

```
var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubstract); //Call the substract method on the input
DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on
overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's to late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it
```

```
dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

Créer une substitution de méthode

Cet exemple montre comment remplacer la méthode `ToString` dans la classe générée

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public |
TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

Lire ILGenerator en ligne: https://riptutorial.com/fr/csharp/topic/667/ilgenerator

Chapitre 81: Immutabilité

Examples

Classe System.String

En C # (et .NET), une chaîne est représentée par la classe System.String. Le mot-clé `string` est un alias pour cette classe.

La classe System.String est immuable, c'est-à-dire qu'une fois créée, son état ne peut pas être modifié.

Donc, toutes les opérations que vous effectuez sur une chaîne comme Substring, Remove, Replace, concatenation using `+` operator etc. vont créer une nouvelle chaîne et la renvoyer.

Voir le programme suivant pour la démonstration -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

Cela imprimera des `string` et des `mystring` respectivement.

Cordes et immuabilité

Les types immuables sont des types qui, lorsqu'ils sont modifiés, créent une nouvelle version de l'objet en mémoire, plutôt que de modifier l'objet existant en mémoire. L'exemple le plus simple est le type de `string` intégré.

Prenant le code suivant, qui ajoute "world" sur le mot "Hello"

```
string myString = "hello";
myString += " world";
```

Ce qui se passe en mémoire dans ce cas, c'est qu'un nouvel objet est créé lorsque vous ajoutez à la `string` dans la deuxième ligne. Si vous faites cela dans le cadre d'une grande boucle, cela peut entraîner des problèmes de performances dans votre application.

L'équivalent mutable pour une `string` est un `StringBuilder`

Prendre le code suivant

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

Lorsque vous exécutez cela, vous modifiez l'objet `StringBuilder` lui-même en mémoire.

Lire Immutabilité en ligne: https://riptutorial.com/fr/csharp/topic/1863/immutabilite

Chapitre 82: Importer les contacts Google

Remarques

Les données de contacts de l'utilisateur seront reçues au format JSON, nous les extraire et finalement nous parcourons ces données pour obtenir les contacts Google.

Examples

Exigences

Pour importer des contacts Google (Gmail) dans une application ASP.NET MVC, commencez par télécharger "Configuration de l'API Google". Vous obtenez les références suivantes:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Ajoutez-les à l'application concernée.

Code source dans le contrôleur

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = "";  // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
```

```
redirectUrl + "&amp&response_type=code&amp&client_id=" + clientId +
"&amp&scope=https://www.google.com/m8/feeds/&amp;approval_prompt=force&amp;access_type=offline");

            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {
                  // You will get all contacts here
                    return View("Index",contacts);
                }
                else
                {
                    return RedirectToAction("Index","Home");
                }
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        public List<GmailContacts> GetAccessToken()
        {
            string code = Request.QueryString["code"];
            string google_client_id = ""; //your google client Id
            string google_client_sceret = ""; // your google secret key
            string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";


            HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
            webRequest.Method = "POST";
            string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
            byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
            webRequest.ContentType = "application/x-www-form-urlencoded";
            webRequest.ContentLength = byteArray.Length;
            Stream postStream = webRequest.GetRequestStream();
            // Add the post data to the web request
            postStream.Write(byteArray, 0, byteArray.Length);
            postStream.Close();
            WebResponse response = webRequest.GetResponse();
            postStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(postStream);
            string responseFromServer = reader.ReadToEnd();
            GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
            /*End*/
            return GetContacts(serStatus);
        }

        public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
        {
            string google_client_id = "";  //client id
```

```
            string google_client_sceret = ""; //secret key
            /*Get Google Contacts From Access Token and Refresh Token*/
            // string refreshToken = serStatus.refresh_token;
            string accessToken = serStatus.access_token;
            string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
            OAuth2Parameters oAuthparameters = new OAuth2Parameters()
            {
                ClientId = google_client_id,
                ClientSecret = google_client_sceret,
                RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
                Scope = scopes,
                AccessToken = accessToken,
                //  RefreshToken = refreshToken
            };

            RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
            ContactsRequest cr = new ContactsRequest(settings);
            ContactsQuery query = new
ContactsQuery(ContactsQuery.CreateContactsUri("default"));
            query.NumberToRetrieve = 5000;
            Feed<Contact> ContactList = cr.GetContacts();

            List<GmailContacts> olist = new List<GmailContacts>();
            foreach (Contact contact in ContactList.Entries)
            {
                foreach (EMail email in contact.Emails)
                {
                    GmailContacts gc = new GmailContacts();
                    gc.EmailID = email.Address;
                    var a = contact.Name.FullName;
                    olist.Add(gc);
                }
            }
            return olist;
        }

        public class GmailContacts
        {
            public string EmailID
            {
                get { return _EmailID; }
                set { _EmailID = value; }
            }
            private string _EmailID;
        }

        public class GooglePlusAccessToken
        {

            public GooglePlusAccessToken()
            { }


            public string access_token
            {
                get { return _access_token; }
                set { _access_token = value; }
            }
            private string _access_token;

            public string token_type
```

```
            {
                get { return _token_type; }
                set { _token_type = value; }
            }
            private string _token_type;

            public string expires_in
            {
                get { return _expires_in; }
                set { _expires_in = value; }
            }
            private string _expires_in;

        }
    }
}
```

Code source dans la vue.

La seule méthode d'action à ajouter est d'ajouter un lien d'action ci-dessous.

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Lire Importer les contacts Google en ligne: https://riptutorial.com/fr/csharp/topic/6744/importer-les-contacts-google

Chapitre 83: Indexeur

Syntaxe

- public ReturnType this [IndexType index] {get {...} set {...}}

Remarques

Indexer permet à une syntaxe de type tableau d'accéder à une propriété d'un objet avec un index.

- Peut être utilisé sur une classe, une structure ou une interface.
- Peut être surchargé.
- Peut utiliser plusieurs paramètres.
- Peut être utilisé pour accéder et définir des valeurs.
- Peut utiliser n'importe quel type pour son index.

Examples

Un indexeur simple

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Usage:

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

Voir la démo

Indexeur avec 2 arguments et interface

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}

class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

Surcharger l'indexeur pour créer un SparseArray

En surchargeant l'indexeur, vous pouvez créer une classe qui ressemble à un tableau, mais qui ne l'est pas. Il aura les méthodes get et set de O (1), pourra accéder à un élément à l'index 100, tout en ayant la taille des éléments qu'il contient. La classe SparseArray

```
class SparseArray
    {
        Dictionary<int, string> array = new Dictionary<int, string>();

        public string this[int i]
        {
            get
            {
                if(!array.ContainsKey(i))
                {
                    return null;
                }
                return array[i];
            }
            set
            {
                if(!array.ContainsKey(i))
                    array.Add(i, value);
            }
        }
    }
```

Lire Indexeur en ligne: https://riptutorial.com/fr/csharp/topic/1660/indexeur

Chapitre 84: Initialisation des propriétés

Remarques

Lorsque vous décidez comment créer une propriété, commencez par une propriété implémentée automatiquement pour plus de simplicité et de concision.

Passez à une propriété avec un champ de sauvegarde uniquement lorsque les circonstances l'exigent. Si vous avez besoin d'autres manipulations au-delà d'un simple jeu et que vous obtenez, vous devrez peut-être introduire un champ de sauvegarde.

Examples

C # 6.0: initialiser une propriété implémentée automatiquement

Créez une propriété avec getter et / ou setter et initialisez tout en une ligne:

```
public string Foobar { get; set; } = "xyz";
```

Initialisation d'une propriété avec un champ de sauvegarde

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

Initialisation de la propriété dans le constructeur

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

Initialisation de la propriété pendant l'instanciation de l'objet

Les propriétés peuvent être définies lorsqu'un objet est instancié.

```
var redCar = new Car
{
    Wheels = 2,
```

```
    Year = 2016,
    Color = Color.Red
};
```

Lire Initialisation des propriétés en ligne: https://riptutorial.com/fr/csharp/topic/82/initialisation-des-proprietes

Chapitre 85: Initialiseurs d'objets

Syntaxe

- SomeClass sc = new SomeClass {Propriété1 = valeur1, Propriété2 = valeur2, ...};
- SomeClass sc = new SomeClass (param1, param2, ...) {Propriété1 = valeur1, Propriété2 = valeur2, ...}

Remarques

Les parenthèses du constructeur ne peuvent être omises que si un type par défaut (sans paramètre) est disponible pour le type instancié.

Examples

Usage simple

Les initialiseurs d'objet sont pratiques lorsque vous devez créer un objet et définir quelques propriétés immédiatement, mais les constructeurs disponibles ne sont pas suffisants. Dis que tu as un cours

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

Pour initialiser une nouvelle instance de la classe avec un initialiseur:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

Ceci est équivalent à

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Utilisation avec des types anonymes

Les initialiseurs d'objet sont le seul moyen d'initialiser les types anonymes, qui sont des types générés par le compilateur.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

Pour cette raison, les initialiseurs d'objet sont largement utilisés dans les requêtes select LINQ, car ils fournissent un moyen pratique de spécifier les parties d'un objet interrogé qui vous intéressent.

```
var albumTitles = from a in albums
                  select new
                  {
                     Title = a.Title,
                     Artist = a.Band
                  };
```

Utilisation avec des constructeurs autres que ceux par défaut

Vous pouvez combiner des initialiseurs d'objet avec des constructeurs pour initialiser des types si nécessaire. Prenons par exemple une classe définie comme telle:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

Cela va d'abord instancier un `Book` avec le constructeur `Book(int)` , puis définir chaque propriété dans l'initialiseur. C'est équivalent à:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Lire Initialiseurs d'objets en ligne: https://riptutorial.com/fr/csharp/topic/738/initialiseurs-d-objets

Chapitre 86: Initialiseurs de collection

Remarques

La seule condition requise pour qu'un objet soit initialisé à l'aide de ce sucre syntaxique est que le type implémente `System.Collections.IEnumerable` et la méthode `Add` . Bien que nous l'appelions un initialiseur de collection, l'objet ne doit *pas* nécessairement être une collection.

Examples

Initialiseurs de collection

Initialiser un type de collection avec des valeurs:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Les initialiseurs de collection sont des symboles syntaxiques pour les appels `Add()` . Le code ci-dessus est équivalent à:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Notez que l'initialisation se fait de manière atomique en utilisant une variable temporaire, pour éviter les conditions de course.

Pour les types qui offrent plusieurs paramètres dans leur méthode `Add()` , placez les arguments séparés par des virgules entre accolades:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

Ceci est équivalent à:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

Initialiseurs d'index C # 6

À partir de C # 6, les collections avec indexeurs peuvent être initialisées en spécifiant l'index à attribuer entre crochets, suivi d'un signe égal, suivi de la valeur à affecter.

Initialisation du dictionnaire

Un exemple de cette syntaxe utilisant un dictionnaire:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

Ceci est équivalent à:

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

La syntaxe d'initialisation de la collection à faire avant C # 6 était la suivante:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Ce qui correspondrait à:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

Il y a donc une différence de fonctionnalité significative, car la nouvelle syntaxe utilise l' *indexeur* de l'objet initialisé pour attribuer des valeurs au lieu d'utiliser sa méthode `Add()` . Cela signifie que la nouvelle syntaxe ne nécessite qu'un indexeur accessible au public et fonctionne pour tout objet qui en possède un.

```
public class IndexableClass
{
    public int this[int index]
    {
        set
        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }
}
```

```
var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}
```

Cela produirait:

```
10 was assigned to index 0
20 was assigned to index 1
```

Initialiseurs de collection dans les classes personnalisées

Pour créer une classe prenant en charge les initialiseurs de collection, il doit implémenter l'interface `IEnumerable` et disposer d'au moins une méthode `Add` . Depuis C # 6, toute collection implémentant `IEnumerable` peut être étendue avec des méthodes `Add` personnalisées à l'aide de méthodes d'extension.

```
class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
```

```
        @this.Add(value.ToString());
}
```

Initialiseurs de collections avec tableaux de paramètres

Vous pouvez mélanger des paramètres normaux et des tableaux de paramètres:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

Cette syntaxe est maintenant possible:

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool"  , 35663, 35732, 12312, 75685},
    {"Bruce"    , 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

Utilisation de l'initialiseur de collection à l'intérieur de l'initialiseur d'objet

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` sont une propriété de type collection. Lorsque l'objet `Tag` est créé à l'aide de la syntaxe d'initialisation d'objet, les `Synonyms` peuvent également être initialisés avec la syntaxe d'initialisation de collection:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

La propriété de collection peut être en lecture seule et prendre en charge la syntaxe d'initialisation de collection. Considérez cet exemple modifié (la propriété `Synonyms` a maintenant un setter privé):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }

    public IList<string> Synonyms { get; private set; }
```

```
}
```

Un nouvel objet `Tag` peut être créé comme ceci:

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

Cela fonctionne parce que les initialiseurs de collection ne sont que du sucre syntatique sur les appels à `Add()` . Il n'y a pas de nouvelle liste créée ici, le compilateur génère simplement des appels à `Add()` sur l'objet existant.

Lire Initialiseurs de collection en ligne: https://riptutorial.com/fr/csharp/topic/21/initialiseurs-de-collection

Chapitre 87: Injection de dépendance

Remarques

La définition de Wikipedia de l'injection de dépendance est la suivante:

En génie logiciel, l'injection de dépendance est un modèle de conception de logiciel qui implémente l'inversion du contrôle pour résoudre les dépendances. Une dépendance est un objet pouvant être utilisé (un service). Une injection est le passage d'une dépendance à un objet dépendant (un client) qui l'utilise.

** **Ce site propose une réponse à la question Comment expliquer l'injection de dépendance à un enfant de 5 ans. La réponse la mieux notée, fournie par John Munsch, fournit une analogie étonnamment précise ciblant l'inquisiteur (imaginaire) âgé de cinq ans: lorsque vous sortez du réfrigérateur pour vous-même, vous pouvez causer des problèmes. Vous pourriez laisser la porte ouverte, vous pourriez obtenir quelque chose que maman ou papa ne veut pas que vous ayez. Vous pourriez même chercher quelque chose que nous n'avons même pas ou qui a expiré. Ce que vous devriez faire, c'est déclarer un besoin: «J'ai besoin de boire quelque chose avec le déjeuner» et nous veillerons à ce que vous ayez quelque chose à manger lorsque vous vous asseyez. En termes de développement de logiciels orientés objet, cela signifie que les classes collaboratrices (les enfants de cinq ans) doivent compter sur l'infrastructure (les parents) pour fournir**

** Ce code utilise MEF pour charger dynamiquement la DLL et résoudre les dépendances. La dépendance ILogger est résolue par MEF et injectée dans la classe d'utilisateurs. La classe d'utilisateurs ne reçoit jamais l'implémentation concrète d'ILogger et elle n'a aucune idée de la nature ou de l'utilisation de l'enregistreur. **

Examples

Injection de dépendance à l'aide de MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
```

```
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);
```

```
        //Fill the imports of this object
        try
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
}
```

Injection de dépendances C # et ASP.NET avec Unity

D'abord pourquoi devrions-nous utiliser l'injection de depedency dans notre code? Nous voulons dissocier les autres composants des autres classes de notre programme. Par exemple, nous avons la classe AnimalController qui a du code comme ceci:

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

Nous regardons ce code et nous pensons que tout va bien, mais maintenant notre AnimalController dépend de l'objet _SantaAndHisReindeer. Automatiquement mon contrôleur est mauvais pour le test et la réutilisation de mon code sera très difficile.

Très bonne explication de la raison pour laquelle nous devons utiliser l'injection de dépendance et les interfaces ici .

Si nous voulons qu'Unity gère le DI, la voie à suivre est très simple :) Avec NuGet (gestionnaire de paquets), nous pouvons facilement importer l'unité dans notre code.

> dans Visual Studio Tools -> Gestionnaire de packages NuGet -> Gérer les packages pour la solution -> dans l'unité de saisie des entrées de recherche -> choisissez notre projet-> cliquez sur installer

Maintenant, deux fichiers avec de bons commentaires seront créés.

> dans le dossier App-Data UnityConfig.cs et UnityMvcActivator.cs
>
> UnityConfig - Dans la méthode RegisterTypes, on peut voir le type qui sera injecté dans nos constructeurs.

```
namespace Vegan.WebUi.App_Start
{

public class UnityConfig
{
```

```
    #region Unity Container
    private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
        return container;
    });

    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
    {
        return container.Value;
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API
controllers (unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not
previously registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IProductRepository, ProductRepository>();

        container.RegisterType<ISanta, SantaAndHisReindeer>();

     }
 }
}
```

UnityMvcActivator -> également avec de bons commentaires qui disent que cette classe intègre Unity avec ASP.NET MVC

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
```

```
        var container = UnityConfig.GetConfiguredContainer();


FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().Firs

        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRec

    }

    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

Maintenant, nous pouvons découpler notre contrôleur de la classe SantAndHisReindeer :)

```
 public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

Il y a une dernière chose à faire avant de lancer notre application.

Dans Global.asax.cs, nous devons ajouter une nouvelle ligne: UnityWebActivator.Start () qui démarrera, configurera Unity et enregistrera nos types.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
```

```
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}
```

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/csharp/topic/5766/injection-de-dependance

Chapitre 88: Interface IDisposable

Remarques

- C'est aux clients de la classe implémentant `IDisposable` de s'assurer qu'ils appellent la méthode `Dispose` lorsqu'ils ont fini d'utiliser l'objet. Il n'y a rien dans le CLR qui recherche directement les objets pour qu'une méthode `Dispose` appelle.
- Il n'est pas nécessaire d'implémenter un finaliseur si votre objet ne contient que des ressources gérées. Assurez-vous d'appeler `Dispose` sur tous les objets que votre classe utilise lorsque vous implémentez votre propre méthode `Dispose` .
- Il est recommandé de sécuriser la classe contre les appels multiples à `Dispose` , même si elle ne devrait idéalement être appelée qu'une seule fois. Cela peut être réalisé en ajoutant une variable `private bool` à votre classe et en définissant la valeur sur `true` lorsque la méthode `Dispose` est exécutée.

Examples

Dans une classe contenant uniquement des ressources gérées

Les ressources gérées sont des ressources connues et contrôlées par le ramasse-miettes de l'environnement d'exécution. Il existe de nombreuses classes disponibles dans la BCL, par exemple, une classe `SqlConnection` qui est une classe wrapper pour une ressource non gérée. Ces classes implémentent déjà l'interface `IDisposable` - c'est à votre code de les nettoyer lorsque vous avez terminé.

Il n'est pas nécessaire d'implémenter un finaliseur si votre classe ne contient que des ressources gérées.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

Dans une classe avec des ressources gérées et non gérées

Il est important de laisser la finalisation ignorer les ressources gérées. Le finaliseur s'exécute sur un autre thread - il est possible que les objets gérés n'existent plus au moment où le finaliseur s'exécute. L'implémentation d'une méthode protégée `Dispose(bool)` est une pratique courante pour s'assurer que les ressources gérées n'ont pas leur méthode `Dispose` appelée depuis un finaliseur.

```
public class ManagedAndUnmanagedObject : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

IDisposable, Dispose

.NET Framework définit une interface pour les types nécessitant une méthode de suppression:

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()` est principalement utilisé pour nettoyer les ressources, comme les références non gérées. Cependant, il peut également être utile de forcer l'élimination des autres ressources même si elles sont gérées. Au lieu d'attendre que le GC finisse par nettoyer votre connexion à la base de données, vous pouvez vous assurer que cela est fait dans votre propre implémentation `Dispose()` .

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
   {
       this.CurrentDatabaseConnection.Dispose();
```

```
        this.CurrentDatabaseConnection = null;
    }
}
```

Lorsque vous devez accéder directement à des ressources non managées, telles que des pointeurs non gérés ou des ressources win32, créez une classe héritant de `SafeHandle` et utilisez les conventions / outils de cette classe pour le faire.

Dans une classe héritée avec des ressources gérées

Il est assez courant de créer une classe qui implémente `IDisposable` , puis de dériver des classes contenant également des ressources gérées. Il est recommandé de marquer la méthode `Dispose` avec le mot clé `virtual` afin que les clients puissent nettoyer toutes les ressources dont ils sont propriétaires.

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

en utilisant le mot clé

Lorsqu'un objet implémente l' `IDisposable` interface, il peut être créé dans l' `using` de la syntaxe:

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
```

```
    {
        Console.WriteLine("dispose called");
    }
}
```

Voir la démo

`using` sucre syntatique pour un bloc `try/finally` ; l'utilisation ci-dessus se traduirait à peu près par:

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

Lire Interface IDisposable en ligne: https://riptutorial.com/fr/csharp/topic/1795/interface-idisposable

Chapitre 89: Interface INotifyPropertyChanged

Remarques

L'interface `INotifyPropertyChanged` est nécessaire chaque fois que vous avez besoin de faire en sorte que votre classe signale les modifications apportées à ses propriétés. L'interface définit un événement unique `PropertyChanged` .

Avec XAML, la liaison de l'événement `PropertyChanged` est automatiquement configurée. Vous n'avez donc qu'à implémenter l'interface INotifyPropertyChanged sur votre modèle de vue ou sur les classes de contexte de données pour travailler avec XAML Binding.

Examples

Implémenter INotifyPropertyChanged en C # 6

L'implémentation de `INotifyPropertyChange` peut être `INotifyPropertyChange` erreurs, car l'interface nécessite de spécifier le nom de la propriété en tant que chaîne. Afin de rendre l'implémentation plus robuste, un attribut `CallerMemberName` peut être utilisé.

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Si vous avez plusieurs classes implémentant `INotifyPropertyChanged` , vous pouvez trouver utile de

refactoriser l'implémentation de l'interface et la méthode d'assistance dans la classe de base commune:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

INotifyPropertyChanged Avec la méthode d'ensemble générique

La classe `NotifyPropertyChangedBase` ci-dessous définit une méthode Set générique pouvant être appelée à partir de n'importe quel type dérivé.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName =
null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

Pour utiliser cette méthode Set générique, il vous suffit de créer une classe dérivée de NotifyPropertyChangedBase.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
```

```
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

Comme indiqué ci-dessus, vous pouvez appeler `Set(ref _fieldName, value);` dans le setter d'une propriété et il déclenchera automatiquement un événement PropertyChanged si nécessaire.

Vous pouvez ensuite vous inscrire à l'événement PropertyChanged à partir d'une autre classe qui doit gérer les modifications de propriété.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }

    private readonly SomeViewModel _vm;

}
```

Lire Interface INotifyPropertyChanged en ligne:
https://riptutorial.com/fr/csharp/topic/2990/interface-inotifypropertychanged

Chapitre 90: Interface IQueryable

Examples

Traduire une requête LINQ en requête SQL

Les interfaces `IQueryable` et `IQueryable<T>` permettent aux développeurs de traduire une requête LINQ (une requête «intégrée au langage») en une source de données spécifique, par exemple une base de données relationnelle. Prenez cette requête LINQ écrite en C #:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

Si la variable `books` est d'un type qui implémente `IQueryable<Book>` la requête ci-dessus est transmise au fournisseur (défini sur la propriété `IQueryable.Provider`) sous la forme d'un arbre d'expression, une structure de données qui reflète la structure du code. .

Le fournisseur peut inspecter l'arborescence d'expression à l'exécution pour déterminer:

- qu'il existe un prédicat pour la propriété `Author` de la classe `Book` ;
- que la méthode de comparaison utilisée est 'égale' (`==`);
- que la valeur à égaler est `"Stephen King"` .

Avec ces informations, le fournisseur peut traduire la requête C # en requête SQL à l'exécution et transmettre cette requête à une base de données relationnelle pour extraire uniquement les livres correspondant au prédicat:

```
select *
from Books
where Author = 'Stephen King'
```

Le fournisseur est appelé lorsque la variable de `query` est itérée sur (`IQueryable` implémente `IEnumerable`).

(Le fournisseur utilisé dans cet exemple nécessiterait des métadonnées supplémentaires pour savoir quelle table interroger et savoir comment faire correspondre les propriétés de la classe C # aux colonnes de la table, mais ces métadonnées ne relèvent pas de l'interface `IQueryable` .)

Lire Interface IQueryable en ligne: https://riptutorial.com/fr/csharp/topic/3094/interface-iqueryable

Chapitre 91: Interfaces

Examples

Implémenter une interface

Une interface est utilisée pour imposer la présence d'une méthode dans toute classe qui l'implémente. L'interface est définie avec le mot clé `interface` et une classe peut l'implémenter en ajoutant `: InterfaceName` après le nom de la classe. Une classe peut implémenter plusieurs interfaces en séparant chaque interface par une virgule.
`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Comme ils implémentent `INoiseMaker` , `cat` et `dog` doivent tous deux inclure la `string MakeNoise()` et ne pourront pas être compilés sans cette méthode.

Implémentation de plusieurs interfaces

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
```

```
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Implémentation d'interface explicite

Une implémentation d'interface explicite est nécessaire lorsque vous implémentez plusieurs interfaces qui définissent une méthode commune, mais différentes implémentations sont requises selon l'interface utilisée pour appeler la méthode (notez que vous n'avez pas besoin d'implémentations explicites si plusieurs interfaces partagent la même méthode et une implémentation commune est possible).

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }

    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}


GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

L'implémentation ne peut pas être appelée de n'importe où sauf en utilisant l'interface:

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
    {
```

```
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

De ce fait, il peut être avantageux de placer le code d'implémentation complexe d'une interface explicitement implémentée dans une méthode séparée et privée.

Une implémentation d'interface explicite ne peut bien entendu être utilisée que pour les méthodes existantes pour cette interface:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

De même, l'utilisation d'une implémentation d'interface explicite sans déclarer cette interface sur la classe provoque également une erreur.

Allusion:

L'implémentation d'interfaces peut également être utilisée pour éviter le code mort. Lorsqu'une méthode n'est plus nécessaire et est supprimée de l'interface, le compilateur se plaindra de chaque implémentation existante.

Remarque:

Les programmeurs s'attendent à ce que le contrat soit le même, quel que soit le contexte du type, et une implémentation explicite ne devrait pas exposer un comportement différent à l'appel. Donc, contrairement à l'exemple ci-dessus, `IGolfPlayer.Drive` et `Drive` devraient faire la même chose lorsque cela est possible.

Pourquoi nous utilisons des interfaces

Une interface est une définition d'un contrat entre l'utilisateur de l'interface et la classe qui l'implémente. Une façon de penser à une interface est de déclarer qu'un objet peut exécuter certaines fonctions.

Disons que nous définissons une interface `IShape` pour représenter différents types de formes, nous attendons une forme pour avoir une zone, nous allons donc définir une méthode pour forcer les implémentations d'interface à renvoyer leur zone:

```
public interface IShape
{
    double ComputeArea();
}
```

Disons que nous avons les deux formes suivantes: un `Rectangle` et un `Circle`

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

Chacun d'entre eux a sa propre définition de son aire, mais les deux sont des formes. Il est donc logique de les voir comme `IShape` dans notre programme:

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea());
    }
}
```

```
// Output:
// Area : 50.00
// Area : 78.54
```

Bases de l'interface

Une fonction d'interface appelée "contrat" de fonctionnalité. Cela signifie qu'il déclare des propriétés et des méthodes mais ne les implémente pas.

Donc, contrairement aux classes Interfaces:

- Ne peut être instancié
- Ne peut avoir aucune fonctionnalité
- Ne peut contenir que des méthodes * *(les propriétés et les événements sont des méthodes internes)*
- L'héritage d'une interface s'appelle "Implementing"
- Vous pouvez hériter d'une classe, mais vous pouvez "implémenter" plusieurs interfaces

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Choses à noter:

- Le préfixe "I" est une convention de dénomination utilisée pour les interfaces.
- Le corps de la fonction est remplacé par un point-virgule ";".
- Les propriétés sont également autorisées car en interne elles sont aussi des méthodes

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

.

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

Ceci est particulièrement utile lorsque vous travaillez avec des frameworks d'interface utilisateur tels que WinForms ou WPF car il est obligatoire d'hériter d'une classe de base pour créer un contrôle utilisateur et vous perdez la possibilité de créer des abstractions sur différents types de contrôles. Un exemple? À venir

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

Le problème proposé est que les deux contiennent un concept de "texte" mais les noms de propriété diffèrent. Et vous ne pouvez pas créer une *classe de base abstraite* car ils ont un héritage obligatoire pour 2 classes différentes. Une interface peut atténuer cela

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

Maintenant, MyButton et MyTextBlock sont interchangeables.

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");


    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;
```

```
    /* the solution is to check the type first.
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors


}
```

"Cacher" les membres avec une implémentation explicite

Ne le détestez-vous pas lorsque les interfaces vous polluent la classe avec trop de membres dont vous ne vous souciez même pas? Eh bien, j'ai une solution! Implémentations explicites

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

Normalement, vous implémenteriez la classe comme ceci.

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

Chaque membre est public

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Réponse: je ne sais pas Donc, il ne devrait pas non plus être déclaré public, mais le simple fait de déclarer les membres comme privés entraînera une erreur du compilateur.

La solution consiste à utiliser une implémentation explicite:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {
```

```
    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

Alors maintenant, vous avez implémenté les membres comme il se doit et ils n'exposent aucun membre en tant que public.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

Si vous voulez sérieusement accéder au membre même s'il est explicitement implémenté, tout ce que vous avez à faire est de placer l'objet sur l'interface et de continuer.

```
((IMessageService)obj).OnMessageRecieve();
```

IComparable comme exemple d'implémentation d'une interface

Les interfaces peuvent sembler abstraites jusqu'à ce que vous les apparaissiez en pratique. `IComparable` et `IComparable<T>` sont d'excellents exemples de la raison pour laquelle les interfaces peuvent nous être utiles.

Disons que dans un programme pour une boutique en ligne, nous avons une variété d'articles que vous pouvez acheter. Chaque article a un nom, un numéro d'identification et un prix.

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity

}
```

Nous avons nos `Item` stockés dans une `List<Item>` , et dans notre programme quelque part, nous voulons trier notre liste par numéro d'identification du plus petit au plus grand. Au lieu d'écrire notre propre algorithme de tri, nous pouvons utiliser la méthode `Sort()` que `List<T>` déjà. Cependant, comme notre classe `Item` est en ce moment, la `List<T>` pas de comprendre l'ordre de tri de la liste. Voici où l'interface `IComparable` entre en jeu.

Pour implémenter correctement la méthode `CompareTo` , `CompareTo` doit renvoyer un nombre positif si le paramètre est "inférieur à" l'actuel, zéro s'il est égal et un nombre négatif si le paramètre est

"supérieur à".

```
Item apple = new Item();
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Voici l'exemple `Item` de mise en œuvre de l'interface »:

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity

}
```

Au niveau de la surface, la méthode `CompareTo` de notre article renvoie simplement la différence entre leurs numéros d'identification, mais que fait-il dans la pratique?

Maintenant, quand nous appelons `Sort()` sur une `List<Item>` objet, la `List` appellera automatiquement le `Item` de `CompareTo` méthode quand il a besoin de déterminer quel ordre de mettre des objets. De plus, en plus de la `List<T>` , tout autre objet ceux qui ont besoin de la possibilité de comparer deux objets fonctionneront avec l' `Item` car nous avons défini la possibilité de comparer deux `Item` différents les uns avec les autres.

Lire Interfaces en ligne: https://riptutorial.com/fr/csharp/topic/2208/interfaces

Chapitre 92: Interopérabilité

Remarques

Travailler avec l'API Win32 en utilisant C #

Windows expose de nombreuses fonctionnalités sous la forme d'une API Win32. En utilisant ces API, vous pouvez effectuer des opérations directes dans Windows, ce qui améliore les performances de votre application.Source Cliquez ici

Windows expose une large gamme d'API. Pour obtenir des informations sur les différentes API, vous pouvez consulter des sites tels que pinvoke .

Examples

Fonction d'importation à partir d'une DLL C ++ non gérée

Voici un exemple d'importation d'une fonction définie dans une DLL C ++ non gérée. Dans le code source C ++ pour "myDLL.dll", la fonction `add` est définie:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Ensuite, il peut être inclus dans un programme C # comme suit:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

Reportez-vous à la section Conventions d'appel et nom C ++ pour savoir pourquoi `extern "C"` et `__stdcall` sont nécessaires.

Trouver la bibliothèque dynamique

Lorsque la méthode extern est invoquée pour la première fois, le programme C # recherche et charge la DLL appropriée. Pour plus d'informations sur l'emplacement de recherche de la DLL et sur la manière dont vous pouvez influencer les emplacements de recherche, consultez cette question sur le stackoverflow .

Code simple pour exposer la classe pour com

```
using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }

        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

C ++ nom mangling

Les compilateurs C ++ codent des informations supplémentaires dans les noms des fonctions exportées, telles que les types d'argument, pour créer des surcharges avec différents arguments possibles. Ce processus s'appelle le mangling name . Cela pose des problèmes avec l'importation de fonctions dans C # (et l'interopérabilité avec d'autres langages en général), le nom de la fonction `int add(int a, int b)` n'étant plus `add` , cela peut être `?add@@YAHHH@Z` , `_add@8` ou autre chose, en fonction du compilateur et de la convention d'appel.

Il y a plusieurs façons de résoudre le problème de la manipulation de noms:

- Exporter des fonctions en utilisant `extern "C"` pour basculer vers la liaison externe C qui utilise la gestion du nom C:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  Le nom de la fonction sera toujours `_add@8` (`_add@8`), mais le nom de `StdCall` + `extern "C"` est reconnu par le compilateur C #.

- Spécification des noms de fonction exportés dans le fichier de définition de module `myDLL.def` :

```
EXPORTS
  add
```

```
int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  Le nom de la fonction sera pur `add` dans ce cas.

- Importer un nom mutilé. Vous aurez besoin d'un visualiseur de DLL pour voir le nom tronqué, vous pourrez alors le spécifier explicitement:

```
__declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

Conventions d'appel

Il existe plusieurs conventions d'appel de fonctions, spécifiant qui (appelant ou appelé) affiche les arguments de la pile, comment les arguments sont transmis et dans quel ordre. C ++ utilise la convention d'appel `Cdecl` par défaut, mais C # attend `StdCall` , qui est généralement utilisé par l'API Windows. Vous devez changer l'un ou l'autre:

- Changer la convention d' `StdCall` en `StdCall` en C ++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- Ou, changez la convention d' `Cdecl` en `Cdecl` en C #:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Si vous souhaitez utiliser une fonction avec la convention d'appel `Cdecl` et un nom tronqué, votre code ressemblera à ceci:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** (**__thiscall**) est principalement utilisé dans les fonctions membres d'une classe.
- Lorsqu'une fonction utilise **thiscall** (**__thiscall**), un pointeur vers la classe est transmis comme premier paramètre.

Chargement et déchargement dynamiques de DLL non gérées

Lorsque vous utilisez l'attribut `DllImport` vous devez connaître le nom de la DLL et de la méthode au *moment de la compilation* . Si vous souhaitez être plus flexible et décider au *moment* de l' *exécution* quelles DLL et méthodes charger, vous pouvez utiliser les méthodes de l'API Windows `LoadLibrary()` , `GetProcAddress()` et `FreeLibrary()` . Cela peut être utile si la bibliothèque à utiliser dépend des conditions d'exécution.

Le pointeur renvoyé par `GetProcAddress()` peut être converti en un délégué à l'aide de `Marshal.GetDelegateForFunctionPointer()` .

L'exemple de code suivant illustre cela avec le `myDLL.dll` des exemples précédents:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
```

```
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module);  // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
typeof(AddDelegate));

        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

Traitement des erreurs Win32

Lorsque vous utilisez des méthodes d'interopérabilité, vous pouvez utiliser l'API **GetLastError** pour obtenir des informations supplémentaires sur vos appels d'API.

Attribut DllImport Attribut SetLastError

SetLastError = true

Indique que l'appelé appellera SetLastError (fonction API Win32).

SetLastError = false

Indique que l'appelé n'appellera **pas** SetLastError (fonction API Win32), vous ne recevrez donc pas d'informations d'erreur.

- Lorsque SetLastError n'est pas défini, il est défini sur false (valeur par défaut).
- Vous pouvez obtenir le code d'erreur à l'aide de la méthode Marshal.GetLastWin32Error:

Exemple:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

Si vous essayez d'ouvrir mutex qui n'existe pas, GetLastError retournera **ERROR_FILE_NOT_FOUND** .

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

Les codes d'erreur du système peuvent être trouvés ici:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx

API GetLastError

Il existe une API **GetLastError** native que vous pouvez également utiliser:

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- Lorsque vous appelez l'API Win32 à partir du code managé, vous devez toujours utiliser l' **erreur Marshal.GetLastWin32Error** .

Voici pourquoi:

Entre votre appel Win32 qui définit l'erreur (appelle SetLastError), le CLR peut également appeler d'autres appels Win32 pouvant appeler **SetLastError** , ce comportement peut remplacer votre valeur d'erreur. Dans ce scénario, si vous appelez **GetLastError,** vous pouvez obtenir une erreur non valide.

Si vous définissez SetLastError = true , le CLR récupère le code d'erreur avant d'exécuter d'autres appels Win32.

Objet épinglé

GC (Garbage Collector) est responsable du nettoyage de nos déchets.

Alors que **GC** nettoie nos déchets, il supprime les objets inutilisés du tas géré, ce qui provoque une fragmentation du tas. Lorsque le **GC** est terminé avec la suppression, il effectue une compression de tas (défragmintation) qui consiste à déplacer des objets sur le tas.

Comme **GC** n'est pas déterministe, lors du passage d'une référence / d'un pointeur d'objet géré à un code natif, **GC** peut intervenir à tout moment, s'il survient juste après l'appel Inerop. être déplacé sur le tas géré - par conséquent, nous obtenons une référence non valide du côté géré.

Dans ce scénario, vous devez **épingler** l'objet avant de le transmettre au code natif.

Objet épinglé

L'objet épinglé est un objet qui n'est pas autorisé à se déplacer par GC.

Poignée épinglée Gc

Vous pouvez créer un objet pin en utilisant la méthode **Gc.Alloc**

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- L'obtention d'un objet **GCHandle** épinglé à un objet géré marque un objet spécifique comme un objet qui ne peut pas être déplacé par **GC** , jusqu'à ce que la poignée soit libérée.

Exemple:

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finaly
    {
        handle.Free()
    }
}
```

Précautions

- Lorsque vous épinglez (surtout les plus gros), l'objet essaye de libérer le **GcHandle** épinglé aussi rapidement que possible, car il interrompt la défragmentation du tas.
- Si vous oubliez de libérer **GcHandle,** rien ne se passera. Faites-le dans une section de code sécurisée (telle que finaly)

Structures de lecture avec le maréchal

La classe Marshal contient une fonction nommée **PtrToStructure** , cette fonction nous donne la possibilité de lire des structures par un pointeur non géré.

La fonction **PtrToStructure** a beaucoup de surcharges, mais toutes ont la même intention.

PtrToStructure générique:

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T -type de structure.

ptr - Un pointeur sur un bloc de mémoire non géré.

Exemple:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- Si vous traitez des objets gérés en lisant des structures natives, n'oubliez pas d'épingler votre objet :)

```
T Read<T>(byte[] buffer)
   {
       T result = default(T);

       var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

       try
       {
           result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
       }
       finally
       {
           gch.Free();
       }

       return result;
   }
```

Lire Interopérabilité en ligne: https://riptutorial.com/fr/csharp/topic/3278/interoperabilite

Chapitre 93: Interpolation de chaîne

Syntaxe

- $ "content {expression} content"
- $ "content {expression: format} content"
- $ "content {expression} {{content entre accolades}} content}"
- $ "content {expression: format} {{content entre accolades}} content}"

Remarques

L'interpolation de chaînes est un raccourci pour la méthode `string.Format()` qui facilite la création de chaînes avec des valeurs de variable et d'expression à l'intérieur.

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);  // returns "Hello, World"
var newWay = $"Hello, {name}!";                   // returns "Hello, World"
```

Examples

Expressions

Les expressions complètes peuvent également être utilisées dans les chaînes interpolées.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Démo en direct sur .NET Fiddle

Format des dates en chaînes

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

Vous pouvez également utiliser la méthode `DateTime.ToString` pour formater l'objet `DateTime` . Cela produira la même sortie que le code ci-dessus.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Sortie:

C'est le 11 novembre 2015, faites un voeu!

Démo en direct sur .NET Fiddle

Démo en direct à l'aide de DateTime.ToString

> **Note:** `MM` représente les mois et les `mm` pour les minutes. Soyez très prudent lorsque vous utilisez ceux-ci, car des erreurs peuvent introduire des bugs qui peuvent être difficiles à détecter.

Utilisation simple

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

Dans les coulisses

En interne ce

```
$"Hello, {name}!"
```

Sera compilé à quelque chose comme ça:

```
string.Format("Hello, {0}!", name);
```

Rembourrage de la sortie

La chaîne peut être formatée pour accepter un paramètre de remplissage qui spécifie le nombre de positions de caractères que la chaîne insérée utilisera:

```
${value, padding}
```

> **REMARQUE:** Les valeurs de remplissage positives indiquent le remplissage à gauche et les valeurs de remplissage négatives indiquent le remplissage à droite.

Rembourrage Gauche

Un remplissage de gauche de 5 (ajoute 3 espaces avant la valeur du nombre, il prend donc un total de 5 positions de caractères dans la chaîne résultante).

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

Sortie:

```
The answer to life, the universe and everything is    42.
```

Démo en direct sur .NET Fiddle

Rembourrage Droit

Le remplissage à droite, qui utilise une valeur de remplissage négative, ajoutera des espaces à la fin de la valeur actuelle.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42    .";
//                                                           ^^^^^
System.Console.WriteLine(str);
```

Sortie:

```
The answer to life, the universe and everything is 42    .
```

Démo en direct sur .NET Fiddle

Rembourrage avec spécificateurs de format

Vous pouvez également utiliser des spécificateurs de formatage existants en conjonction avec un remplissage.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//                                                           ^^^^^
```

Démo en direct sur .NET Fiddle

Mise en forme des nombres dans les chaînes

Vous pouvez utiliser deux points et la syntaxe de format numérique standard pour contrôler le formatage des nombres.

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;
```

```
var prefixedIfNecessary = $"{integerValue:D5}";
// String value is "00057"
```

Démo en direct sur .NET Fiddle

Lire Interpolation de chaîne en ligne: https://riptutorial.com/fr/csharp/topic/22/interpolation-de-chaine

Chapitre 94: La mise en réseau

Syntaxe

- TcpClient (hôte de chaîne, int port);

Remarques

Vous pouvez obtenir le `NetworkStream` partir d'un `TcpClient` avec `client.GetStream()` et le transmettre à un `StreamReader/StreamWriter` pour accéder à leurs méthodes de lecture et d'écriture asynchrone.

Examples

Client de communication TCP de base

Cet exemple de code crée un client TCP, envoie "Hello World" via la connexion de socket, puis écrit la réponse du serveur sur la console avant de fermer la connexion.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivilent but safer
than calling Close();
```

Télécharger un fichier depuis un serveur Web

Télécharger un fichier à partir d'Internet est une tâche très courante requise par presque toutes les applications que vous êtes susceptible de créer.

Pour ce faire, vous pouvez utiliser la classe " System.Net.WebClient ".

L'utilisation la plus simple de ceci, en utilisant le modèle "using", est montrée ci-dessous:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

Dans cet exemple, il utilise "using" pour s'assurer que votre client Web est correctement nettoyé une fois terminé, et transfère simplement la ressource nommée depuis l'URL du premier paramètre vers le fichier nommé sur votre disque dur local dans le second. paramètre.

Le premier paramètre est de type " System.Uri ", le second paramètre est de type " System.String "

Vous pouvez également utiliser cette fonction en tant que forme asynchrone, de sorte qu'elle se déclenche et effectue le téléchargement en arrière-plan, tandis que votre application utilise autre chose, l'utilisation de cet appel revêt une importance majeure dans les applications modernes. pour garder votre interface utilisateur réactive.

Lorsque vous utilisez les méthodes Async, vous pouvez connecter des gestionnaires d'événements qui vous permettent de surveiller la progression, de manière à pouvoir, par exemple, mettre à jour une barre de progression, par exemple:

```
var webClient = new WebClient())
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

Un point important à retenir si vous utilisez les versions d'Async cependant, et c'est "Soyez très prudent lorsque vous les utilisez dans un" en utilisant la "syntaxe".

La raison en est assez simple. Une fois que vous appelez la méthode du fichier de téléchargement, elle reviendra immédiatement. Si vous avez ceci dans un bloc using, vous retournerez puis quitterez ce bloc, et disposerez immédiatement de l'objet de classe, et annulerez ainsi votre téléchargement en cours.

Si vous utilisez la méthode "using" pour effectuer un transfert asynchrone, assurez-vous de rester dans le bloc englobant jusqu'à la fin du transfert.

Async TCP Client

L'utilisation `async/await` dans les applications C # simplifie le multi-threading. Voici comment vous pouvez utiliser `async/await` waiting en conjonction avec un TcpClient.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
```

```
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // Asynchronsly attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivilent but safer than calling Close();
```

Client UDP de base

Cet exemple de code crée un client UDP, puis envoie "Hello World" sur le réseau au destinataire prévu. Un écouteur ne doit pas nécessairement être actif, car UDP est sans connexion et diffusera le message indépendamment. Une fois le message envoyé, le travail des clients est terminé.

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
     using (var client = new UdpClient())
     {
         IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
         client.Connect(ep);
         client.Send(data, data.Length);
     }
}
catch (Exception ex)
{
     Console.WriteLine(ex.ToString());
}
```

Vous trouverez ci-dessous un exemple d'un auditeur UDP en complément du client ci-dessus. Il restera en permanence à l'écoute du trafic sur un port donné et écrira simplement ces données sur la console. Cet exemple contient un indicateur de contrôle ' `done` ' qui n'est pas défini en interne et s'appuie sur quelque chose pour définir ceci afin de permettre de mettre fin à l'écouteur et de le quitter.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}",
listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World"
sent from above client
    }
}
```

Lire La mise en réseau en ligne: https://riptutorial.com/fr/csharp/topic/1352/la-mise-en-reseau

Chapitre 95: Lecture et écriture de fichiers .zip

Syntaxe

1. statique publique ZipArchive OpenRead (string archiveFileName)

Paramètres

Paramètre	Détails
archiveFileName	Le chemin d'accès à l'archive à ouvrir, spécifié en tant que chemin relatif ou absolu. Un chemin relatif est interprété comme relatif au répertoire de travail en cours.

Examples

Ecrire dans un fichier zip

Pour écrire un nouveau fichier .zip:

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("========================");
        }
    }
}
```

Écriture de fichiers Zip en mémoire

L'exemple suivant renverra les données d' `byte[]` d'un fichier zippé contenant les fichiers qui lui sont fournis, sans devoir accéder au système de fichiers.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
```

```
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file
with this name
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
        }
        //ZipArchive must be disposed before the MemoryStream has data
        return ms.ToArray();
    }
}
```

Obtenir des fichiers à partir d'un fichier Zip

Cet exemple obtient une liste de fichiers à partir des données binaires d'archive zip fournies:

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}
```

L'exemple suivant montre comment ouvrir une archive zip et extraire tous les fichiers .txt dans un dossier

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
```

```
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}
```

Lire Lecture et écriture de fichiers .zip en ligne: https://riptutorial.com/fr/csharp/topic/6709/lecture-et-ecriture-de-fichiers--zip

Chapitre 96: Les attributs

Examples

Création d'un attribut personnalisé

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using
System.AttributeUsage Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

Utiliser un attribut

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Lecture d'un attribut

La méthode `GetCustomAttributes` renvoie un tableau d'attributs personnalisés appliqués au membre. Après avoir récupéré ce tableau, vous pouvez rechercher un ou plusieurs attributs spécifiques.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Ou itérer à travers eux

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`GetCustomAttribute` méthode d'extension `GetCustomAttribute` de `System.Reflection.CustomAttributeExtensions` extrait un attribut personnalisé d'un type spécifié, il peut être appliqué à n'importe quel `MemberInfo` .

```
var attribute = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute` également une signature générique pour spécifier le type d'attribut à rechercher.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

Argument booléen `inherit` peut être transmis à ces deux méthodes. Si cette valeur est définie sur `true` les ancêtres de l'élément seront également inspectés.

Attribut DebuggerDisplay

L'ajout de l'attribut `DebuggerDisplay` modifie la façon dont le débogueur affiche la classe lorsqu'il est survolé.

Les expressions enveloppées dans `{}` seront évaluées par le débogueur. Cela peut être une propriété simple comme dans l'exemple suivant ou une logique plus complexe.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```

L'ajout de `,nq` avant le crochet de fermeture supprime les guillemets lors de la sortie d'une chaîne.

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

Même si les expressions générales sont autorisées dans `{}` elles ne sont pas recommandées. L'attribut `DebuggerDisplay` sera écrit dans la métadonnée de l'assemblage sous forme de chaîne. Les expressions dans `{}` ne sont pas vérifiées pour la validité. Ainsi, un attribut `DebuggerDisplay` contenant une logique plus complexe que par exemple une arithmétique simple pourrait fonctionner correctement en C #, mais la même expression évaluée dans VB.NET ne sera probablement pas syntaxiquement valide et produira une erreur lors du débogage.

Une façon de rendre `DebuggerDisplay` plus agnostique en `DebuggerDisplay` langage consiste à écrire l'expression dans une méthode ou une propriété et à l'appeler à la place.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

   private string DebuggerDisplay()
    {
        return $"{StringProperty} - {IntProperty}"";
    }
}
```

On peut souhaiter que `DebuggerDisplay` toutes ou certaines propriétés et lors du débogage et de l'inspection du type d'objet.
L'exemple ci-dessous entoure également la méthode d'assistance avec `#if DEBUG` car `DebuggerDisplay` est utilisé dans les environnements de débogage.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

#if DEBUG
   private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
Type:{this.GetType()}";
    }
    #endif
}
```

Attributs d'information de l'appelant

Les attributs d'information de l'appelant peuvent être utilisés pour transmettre des informations sur l'invocateur à la méthode invoquée. La déclaration ressemble à ceci:

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
```

```
                        [CallerMemberName]string callerMemberName = "",
                        [CallerLineNumber]int callerLineNumber = 0,
                        [CallerFilePath]string callerFilePath = "")
{
    //perform logging
}
```

Et l'invocation ressemble à ceci:

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

Notez que seul le premier paramètre est passé explicitement à la méthode `LogException` alors que les autres seront fournis au moment de la compilation avec les valeurs appropriées.

Le paramètre `callerMemberName` recevra la valeur `"Save"` - le nom de la méthode d'appel.

Le paramètre `callerLineNumber` recevra le numéro de la ligne sur `LogException` appel de méthode `LogException` est écrit.

Et le paramètre 'callerFilePath' recevra le chemin complet du fichier dans lequel la méthode `Save` est déclarée.

Lecture d'un attribut depuis l'interface

Il n'y a pas de moyen simple d'obtenir des attributs à partir d'une interface, car les classes n'héritent pas des attributs d'une interface. Chaque fois que vous implémentez une interface ou que vous remplacez des membres dans une classe dérivée, vous devez déclarer à nouveau les attributs. Donc, dans l'exemple ci-dessous, le résultat serait `True` dans les trois cas.

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }
```

```
    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True

            var attribute2 =
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}
```

Une façon de récupérer les attributs d'interface consiste à les rechercher via toutes les interfaces implémentées par une classe.

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

Attribut obsolète

System.Obsolete est un attribut utilisé pour marquer un type ou un membre qui a une meilleure version et ne doit donc pas être utilisé.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

Si la classe ci-dessus est utilisée, le compilateur affichera l'avertissement "Cette classe est obsolète. Utilisez plutôt SomeOtherClass".

Lire Les attributs en ligne: https://riptutorial.com/fr/csharp/topic/1062/les-attributs

Chapitre 97: Les délégués

Remarques

Résumé

Un **type de délégué** est un type représentant une signature de méthode particulière. Une instance de ce type fait référence à une méthode particulière avec une signature correspondante. Les paramètres de méthode peuvent avoir des types de délégué, et donc une méthode à transmettre une référence à une autre méthode, qui peut ensuite être appelée

Types de délégué intégrés: `Action<...>` , `Predicate<T>` et `Func<...,TResult>`

L'espace de noms `System` contient les délégués `Action<...>` , `Predicate<T>` et `Func<...,TResult>` , où "..." représente entre 0 et 16 paramètres de type générique (pour 0 paramètre, `Action` est non générique).

`Func` représente des méthodes avec un type de retour correspondant à `TResult` , et `Action` représente des méthodes sans valeur de retour (void). Dans les deux cas, les paramètres de type générique supplémentaires correspondent, dans l'ordre, aux paramètres de la méthode.

`Predicate` représente la méthode avec le type de retour booléen, T est le paramètre d'entrée.

Types de délégué personnalisés

Les types de délégué nommés peuvent être déclarés à l'aide du mot clé `delegate` .

Invoquer des délégués

Les délégués peuvent être appelés en utilisant la même syntaxe que les méthodes: le nom de l'instance du délégué, suivi des parenthèses contenant des paramètres.

Affectation aux délégués

Les délégués peuvent être affectés aux manières suivantes:

- Assigner une méthode nommée
- Affectation d'une méthode anonyme à l'aide d'un lambda
- Affectation d'une méthode nommée à l'aide du mot clé `delegate` .

Combinaison de délégués

Plusieurs objets délégués peuvent être affectés à une instance de délégué à l'aide de l'opérateur `+` . L'opérateur `-` peut être utilisé pour supprimer un délégué de composant d'un autre délégué.

Examples

Références sous-jacentes des délégués de méthodes nommées

Lors de l'attribution de méthodes nommées aux délégués, ils font référence au même objet sous-jacent si:

- Ils sont la même méthode d'instance, sur la même instance d'une classe
- Ils sont la même méthode statique sur une classe

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

Déclaration d'un type de délégué

La syntaxe suivante crée un type de `delegate` nommé `NumberInOutDelegate` , représentant une méthode qui prend un `int` et retourne un `int` .

```
public delegate int NumberInOutDelegate(int input);
```

Cela peut être utilisé comme suit:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

L' `example` instance de délégué est exécuté de la même manière que la méthode `Square` . Une instance de délégué agit littéralement en tant que délégué pour l'appelant: l'appelant appelle le délégué, puis le délégué appelle la méthode cible. Cette indirection dissocie l'appelant de la méthode cible.

Vous pouvez déclarer un type de délégué **générique** et, dans ce cas, vous pouvez spécifier que le type est covariant (`out`) ou contravariant (`in`) dans certains des arguments de type. Par exemple:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Comme pour les autres types génériques, les types de délégué génériques peuvent avoir des contraintes, par exemple `where TFrom : struct, IConvertible where TTo : new()` .

Évitez la co-et la contravariance pour les types de délégué qui doivent être utilisés pour les délégués multidiffusion, tels que les types de gestionnaires d'événements. En effet, la concaténation (`+`) peut échouer si le type d'exécution est différent du type à la compilation en raison de la variance. Par exemple, évitez:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Au lieu de cela, utilisez un type générique invariant:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Sont également pris en charge les délégués où certains paramètres sont modifiés par `ref` ou `out` , comme dans:

```
public delegate bool TryParser<T>(string input, out T result);
```

(exemple: `TryParser<decimal> example = decimal.TryParse;`) ou délégué où le dernier paramètre a le modificateur `params` . Les types de délégué peuvent avoir des paramètres facultatifs (valeurs par défaut de fourniture). Les types de délégué peuvent utiliser des types de pointeur comme `int*` ou `char*` dans leurs signatures ou leurs types de retour (utilisez le mot-clé `unsafe`). Un type de délégué et ses paramètres peuvent comporter des attributs personnalisés.

Le func , Action et prédicat types de délégué

L'espace de noms Système contient les types de délégué `Func<..., TResult>` avec entre 0 et 15 paramètres génériques, renvoyant le type `TResult` .

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but
the first as parameters of that type
    Console.WriteLine(output);
}
```

L'espace de noms Système contient également des types de délégués `Action<...>` avec un nombre différent de paramètres génériques (de 0 à 16). Il est similaire à `Func<T1, .., Tn>` , mais il renvoie toujours un `void` .

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type
arguments
}
```

`Predicate<T>` est également une forme de `Func` mais il renverra toujours `bool` . Un prédicat est un moyen de spécifier un critère personnalisé. En fonction de la valeur de l'entrée et de la logique définie dans le prédicat, elle retournera soit `true` soit `false` . `Predicate<T>` se comporte donc comme `Func<T, bool>` et les deux peuvent être initialisés et utilisés de la même manière.

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");

// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

Le choix d'utiliser ou non `Predicate<T>` ou `Func<T, bool>` est vraiment une question d'opinion. `Predicate<T>` est sans doute plus expressif de l'intention de l'auteur, tandis que `Func<T, bool>` est susceptible d'être familier à une plus grande proportion de développeurs C #.

En plus de cela, il existe certains cas où une seule des options est disponible, en particulier lors de l'interaction avec une autre API. Par exemple, `List<T>` et `Array<T>` prennent généralement `Predicate<T>` pour leurs méthodes, tandis que la plupart des extensions LINQ n'acceptent que `Func<T, bool>` .

Affectation d'une méthode nommée à un délégué

Les méthodes nommées peuvent être attribuées aux délégués avec des signatures correspondantes:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne` prend un `int` et retourne un `int` , sa signature correspond au délégué `Func<int,int>` . `Example.AddOne` peut être directement affecté à `addOne` car ils ont des signatures correspondantes.

Égalité des délégués

L'appel de `.Equals()` sur un délégué se compare par égalité de référence:

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

Ces règles s'appliquent également lors de l'exécution de `+=` ou `-=` sur un délégué multidiffusion, par exemple lors de l'inscription et de la désinscription à des événements.

Assigner à un délégué par lambda

Les Lambdas peuvent être utilisés pour créer des méthodes anonymes à affecter à un délégué:

```
Func<int,int> addOne = x => x+1;
```

Notez que la déclaration explicite de type est requise lors de la création d'une variable de cette manière:

```
var addOne = x => x+1; // Does not work
```

Passer des délégués en tant que paramètres

Les délégués peuvent être utilisés comme pointeurs de fonctions typés:

```
class FuncAsParameters
{
  public void Run()
  {
    DoSomething(ErrorHandler1);
    DoSomething(ErrorHandler2);
  }

  public bool ErrorHandler1(string message)
  {
    Console.WriteLine(message);
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public bool ErrorHandler2(string message)
  {
    // ...Write message to file...
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public void DoSomething(Func<string, bool> errorHandler)
  {
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
      if (!errorHandler("Some error occurred!"))
      {
        // The handler decided we can't continue
        return;
      }
    }
  }
}
```

Combiner des délégués (délégués à la multidiffusion)

Addition `+` et soustraction `-` opérations peuvent être utilisées pour combiner des instances de délégué. Le délégué contient une liste des délégués assignés.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

Dans cet exemple, `d3` est une combinaison de délégués `d1` et `d2` . Par conséquent, lorsqu'il est appelé, le programme `System.Int32` chaînes `1` et `System.Int32` .

Combinaison de délégués avec **des** types de retour **non vides** :

Si un délégué multidiffusion a un type de retour non `nonvoid` , l'appelant reçoit la valeur de retour de la dernière méthode à appeler. Les méthodes précédentes sont toujours appelées, mais leurs

valeurs de retour sont ignorées.

```
    class Program
    {
        public delegate int Transformer(int x);

        static void Main(string[] args)
        {
            Transformer t = Square;
            t += Cube;
            Console.WriteLine(t(2));  // O/P 8
        }

        static int Square(int x) { return x * x; }

        static int Cube(int x) { return x*x*x; }
    }
```

`t(2)` appellera d'abord `Square` puis `Cube` . La valeur de retour de Square est ignorée et la valeur renvoyée de la dernière méthode, c.-à-d. `Cube` est conservé.

Appel de sécurité multicast délégué

Vous avez toujours voulu appeler un délégué multidiffusion mais vous souhaitez que la liste complète des invocations soit appelée même si une exception se produit dans la chaîne. Ensuite, vous avez de la chance, j'ai créé une méthode d'extension qui ne fait que cela, en lançant une `AggregateException` uniquement une fois l'exécution de la liste complète terminée:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del,params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}

public class Test
{
    public delegate void SampleDelegate();
```

```
    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}
```

Cela produit:

```
Target 2 executed
Target 1 executed
```

L'appel direct, sans `SaveInvoke` , n'exécutera que Target 2.

Fermeture à l'intérieur d'un délégué

Les fermetures sont des méthodes anonymes en ligne qui ont la capacité d'utiliser des variables de méthode `Parent` et d'autres méthodes anonymes définies dans la portée du parent.

> Essentiellement, une fermeture est un bloc de code qui peut être exécuté ultérieurement, mais qui conserve l'environnement dans lequel il a été créé pour la première fois, c'est-à-dire qu'il peut toujours utiliser les variables locales de la méthode qui l'a créé. la méthode a terminé son exécution. **- Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();

}
```

Exemple tiré des fermetures dans .NET .

Encapsulation des transformations dans les funcs

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

Dans l'esprit du codage propre, l'encapsulation de vérifications et de transformations comme celle ci-dessus en tant que Func peut faciliter la lecture et la compréhension de votre code. Bien que l'exemple ci-dessus soit très simple, que se passe-t-il s'il y a plusieurs propriétés DateTime ayant chacune leurs règles de validation différentes et que nous voulions vérifier différentes combinaisons? Des fonctions simples, à une ligne, ayant chacune une logique de retour établie peuvent être à la fois lisibles et réduire la complexité apparente de votre code. Considérez les appels Func ci-dessous et imaginez à quel point le code encombrerait la méthode:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Lire Les délégués en ligne: https://riptutorial.com/fr/csharp/topic/1194/les-delegues

Chapitre 98: Les itérateurs

Remarques

Un itérateur est une méthode, un accesseur ou un opérateur qui exécute une itération personnalisée sur un tableau ou une classe de collection à l'aide du mot-clé de rendement.

Examples

Exemple d'itérateur numérique simple

Un cas d'usage courant pour les itérateurs est d'effectuer certaines opérations sur une collection de nombres. L'exemple ci-dessous montre comment chaque élément d'un tableau de nombres peut être imprimé individuellement sur la console.

Cela est possible car les tableaux implémentent l'interface `IEnumerable` , permettant aux clients d'obtenir un itérateur pour le tableau à l'aide de la méthode `GetEnumerator()` . Cette méthode retourne un *énumérateur* , qui est un curseur en lecture seule, en avant uniquement sur chaque nombre du tableau.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Sortie

```
1
2
3
4
5
```

Il est également possible d'obtenir les mêmes résultats en utilisant une déclaration `foreach` :

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Création d'itérateurs à l'aide du rendement

Les itérateurs *produisent des* enquêteurs. En C #, les énumérateurs sont produits en définissant des méthodes, des propriétés ou des indexeurs contenant `yield` instructions de `yield` .

La plupart des méthodes renverront le contrôle à leur appel via des instructions de `return` normales, qui éliminent tous les états locaux de cette méthode. En revanche, les méthodes qui utilisent `yield` instructions de `yield` leur permettent de renvoyer plusieurs valeurs à l'appelant sur demande tout en *conservant* l'état local entre ces valeurs. Ces valeurs renvoyées constituent une séquence. Il existe deux types d'énoncés de `yield` utilisés dans les itérateurs:

- `yield return` , qui retourne le contrôle à l'appelant mais conserve l'état. L'appelé continuera l'exécution de cette ligne quand le contrôle lui sera renvoyé.
- `yield break` , qui fonctionne de manière similaire à une instruction de `return` normale - cela signifie la fin de la séquence. Les instructions de `return` normales elles-mêmes sont illégales dans un bloc itérateur.

Cet exemple ci-dessous illustre une méthode d'itérateur pouvant être utilisée pour générer la séquence de Fibonacci :

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

Cet itérateur peut ensuite être utilisé pour produire un énumérateur de la séquence de Fibonacci pouvant être consommé par une méthode d'appel. Le code ci-dessous montre comment les dix premiers termes de la séquence de Fibonacci peuvent être énumérés:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Sortie

```
1
1
2
3
5
8
13
21
34
55
```

Lire Les itérateurs en ligne: https://riptutorial.com/fr/csharp/topic/4243/les-iterateurs

Chapitre 99: Les méthodes

Examples

Déclaration d'une méthode

Chaque méthode a une signature unique composée d'un accesseur (`public` , `private` , ...), d'un modificateur facultatif (`abstract`), d'un nom et, si nécessaire, de paramètres de méthode. Notez que le type de retour ne fait pas partie de la signature. Un prototype de méthode ressemble à ceci:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` peut être `public` , `protected` , `pirvate` ou par défaut `internal` .

`OptionalModifier` peut être `static override virtual abstract static new` ou `sealed` .

`ReturnType` peut être `void` sans retour ou peut être tout type de celles de base, comme `int` aux classes complexes.

une méthode peut avoir certains ou aucun paramètre d'entrée. pour définir les paramètres d'une méthode, vous devez les déclarer comme des déclarations de variables normales (comme `int a`), et pour plus d'un paramètre, vous devez utiliser des virgules entre eux (comme `int a, int b`).

Les paramètres peuvent avoir des valeurs par défaut. pour cela, vous devez définir une valeur pour le paramètre (comme `int a = 0`). Si un paramètre a une valeur par défaut, la définition de la valeur d'entrée est facultative.

L'exemple de méthode suivant renvoie la somme de deux entiers:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Appeler une méthode

Appeler une méthode statique:

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Appeler une méthode statique et stocker sa valeur de retour:

```
string input = System.Console.ReadLine();
```

Appel d'une méthode d'instance:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Appeler une méthode générique

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

Paramètres et arguments

Une méthode peut déclarer un nombre quelconque de paramètres (dans cet exemple, `i` , `s` et `o` sont les paramètres):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Les paramètres peuvent être utilisés pour transmettre des valeurs dans une méthode, afin que la méthode puisse les utiliser. Cela peut être tout type de travail, comme l'impression des valeurs ou la modification de l'objet référencé par un paramètre ou le stockage des valeurs.

Lorsque vous appelez la méthode, vous devez transmettre une valeur réelle pour chaque paramètre. À ce stade, les valeurs que vous transmettez réellement à l'appel de méthode sont appelées Arguments:

```
DoSomething(x, "hello", new object());
```

Types de retour

Une méthode peut renvoyer soit rien (`void`), soit une valeur d'un type spécifié:

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

Si votre méthode spécifie une valeur de retour, la méthode *doit* renvoyer une valeur. Vous faites

cela en utilisant la déclaration de `return` . Une fois qu'une déclaration de `return` a été atteinte, elle retourne la valeur spécifiée et tout code après qu'il ne sera plus exécuté (les exceptions sont `finally` blocs, qui seront toujours exécutés avant le retour de la méthode).

Si votre méthode ne retourne rien (`void`), vous pouvez toujours utiliser l'instruction `return` sans une valeur si vous souhaitez retourner immédiatement à la méthode. À la fin d'une telle méthode, une déclaration de `return` serait cependant inutile.

Exemples d'instructions de `return` valides:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Lancer une exception peut mettre fin à l'exécution de la méthode sans renvoyer de valeur. En outre, il existe des blocs d'itérateurs, où les valeurs de retour sont générées à l'aide du mot clé yield, mais ce sont des cas particuliers qui ne seront pas expliqués à ce stade.

Paramètres par défaut

Vous pouvez utiliser les paramètres par défaut si vous souhaitez fournir l'option permettant de supprimer les paramètres:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

Lorsque vous appelez une telle méthode et omettez un paramètre pour lequel une valeur par défaut est fournie, le compilateur insère cette valeur par défaut pour vous.

Gardez à l'esprit que les paramètres avec les valeurs par défaut doivent être écrits **après les** paramètres sans valeurs par défaut.

```
static void SaySomething(string say, string what = "ehh") {
        //Correct
        Console.WriteLine(say + what);
    }

static void SaySomethingElse(string what = "ehh", string say) {
        //Incorrect
        Console.WriteLine(say + what);
    }
```

ATTENTION : Comme cela fonctionne de cette façon, les valeurs par défaut peuvent être

problématiques dans certains cas. Si vous modifiez la valeur par défaut d'un paramètre de méthode et ne recompilez pas tous les appelants de cette méthode, ces appelants utiliseront toujours la valeur par défaut qui était présente lors de la compilation, ce qui pourrait entraîner des incohérences.

Surcharge de méthode

Définition: Lorsque plusieurs méthodes portant le même nom sont déclarées avec des paramètres différents, on parle de surcharge de méthode. La surcharge de méthode représente généralement des fonctions identiques mais écrites pour accepter différents types de données en tant que paramètres.

Facteurs affectant

- Nombre d'arguments
- Type d'arguments
- Type de retour **

Considérons une méthode nommée `Area` qui exécutera des fonctions de calcul, qui acceptera divers arguments et renverra le résultat.

Exemple

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

Cette méthode accepte un argument et retourne une chaîne, si nous appelons la méthode avec un entier (disons `5`), la sortie sera `"Area of Square is 25"` .

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

De même, si nous transmettons deux valeurs doubles à cette méthode, la sortie sera le produit des deux valeurs et sera du type double. Cela peut être utilisé pour la multiplication aussi bien que pour trouver l'aire des rectangles

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

Cela peut être utilisé spécialement pour trouver l'aire du cercle, qui accepte une valeur double (`radius`) et renvoie une autre valeur double comme surface.

Chacune de ces méthodes peut être appelée normalement sans conflit - le compilateur examinera les paramètres de chaque appel de méthode pour déterminer quelle version de `Area` doit être

utilisée.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

** Notez que le type de retour *seul* ne peut pas différencier deux méthodes. Par exemple, si nous avions deux définitions pour Area qui avaient les mêmes paramètres, comme ceci:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

Si nous avons besoin que notre classe utilise les mêmes noms de méthode qui renvoient des valeurs différentes, nous pouvons supprimer les problèmes d'ambiguïté en implémentant une interface et en définissant explicitement son utilisation.

```
public interface IAreaCalculatorString {

    public string Area(double width, double height);

}

public class AreaCalculator : IAreaCalculatorString {

    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
```

Méthode anonyme

Les méthodes anonymes fournissent une technique pour transmettre un bloc de code en tant que paramètre délégué. Ce sont des méthodes avec un corps, mais pas de nom.

```
delegate int IntOp(int lhs, int rhs);
```

```
class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };
```

```
        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;

        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}
```

Des droits d'accès

```
// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual  void MyMethod()

// internal: access is limited within the current assembly
internal  void MyMethod()

//private: access is limited only within the same class
private  void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the
containing class.
protected internal void MyMethod()
```

Lire Les méthodes en ligne: https://riptutorial.com/fr/csharp/topic/60/les-methodes

Chapitre 100: Les opérateurs

Introduction

En C #, un opérateur est un élément de programme appliqué à un ou plusieurs opérandes dans une expression ou une instruction. Les opérateurs qui prennent une opérande, tels que l'opérateur d'incrémentation (++) ou new, sont appelés opérateurs unaires. Les opérateurs qui prennent deux opérandes, tels que les opérateurs arithmétiques (+, -, *, /), sont appelés opérateurs binaires. Un opérateur, l'opérateur conditionnel (? :), prend trois opérandes et est le seul opérateur ternaire en C #.

Syntaxe

- opérateur public OperandType statique operatorSymbol (opérande OperandType1)
- opérateur public OperandType statique operatorSymbol (opérande OperandType1, opérande OperandType22)

Paramètres

Paramètre	Détails
opérateurSymbole	L'opérateur étant surchargé, par exemple +, -, /, *
OperandType	Le type qui sera renvoyé par l'opérateur surchargé.
opérande1	Le premier opérande à utiliser pour effectuer l'opération.
opérande2	Le deuxième opérande à utiliser pour effectuer l'opération lors d'opérations binaires.
déclarations	Code facultatif nécessaire pour effectuer l'opération avant de renvoyer le résultat.

Remarques

Tous les opérateurs sont définis comme `static methods` et ils ne sont pas `virtual` et ils ne sont pas hérités.

Priorité de l'opérateur

Tous les opérateurs ont une "priorité" particulière selon le groupe auquel appartient l'opérateur (les opérateurs du même groupe ont la même priorité). Ce qui signifie que certains opérateurs seront appliqués avant les autres. Ce qui suit est une liste de groupes (contenant leurs opérateurs respectifs) classés par priorité (le plus élevé en premier):

- **Opérateurs Primaires**
    - `ab` - Accès membre.
    - `a?.b` - Null accès conditionnel membre.
    - `->` - Déréférencement du pointeur associé à l'accès des membres.
    - `f(x)` - invocation de fonction.
    - `a[x]` - Indexeur.
    - `a?[x]` - Indexeur conditionnel nul.
    - `x++` - Incrément Postfix.
    - `x--` - décrémentation `x--` .
    - `new` - Type instanciation.
    - `default(T)` - Retourne la valeur initialisée par défaut du type `T`
    - `typeof` - Retourne l'objet `Type` de l'opérande.
    - `checked` - Active la vérification du dépassement numérique.
    - `unchecked` - Désactive la vérification numérique des dépassements.
    - `delegate` - Déclare et retourne une instance de délégué.
    - `sizeof` - Retourne la taille en octets du type opérande.
- **Opérateurs Unaires**
    - `+x` - Retourne `x` .
    - `-x` - Négation numérique.
    - `!x` - Négation logique.
    - `~x` - Complément binaire / déclare les destructeurs.
    - `++x` - Incrément de préfixe.
    - `--x` - Préfixe décrémenté.
    - `(T)x` - Type coulée.
    - `await` - attend une `Task` .
    - `&x` - Retourne l'adresse (pointeur) de `x` .
    - `*x` - Déréférencement de pointeur.
- **Opérateurs Multiplicatifs**
    - `x * y` - Multiplication.
    - `x / y` - Division.
    - `x % y` - Module.
- **Opérateurs Additifs**
    - `x + y` - Ajout.
    - `x - y` - Soustraction.
- **Opérateurs de décalage binaire**
    - `x << y` - bits de décalage restants.
    - `x >> y` - Décale les bits à droite.
- **Opérateurs relationnels / tests de type**

    - `x < y` - moins que.
    - `x > y` - Supérieur à.
    - `x <= y` - Inférieur ou égal à.
    - `x >= y` - supérieur ou égal à.
    - `is` - Type compatibilité.
    - `as` - Type conversion.

- **Opérateurs d'égalité**

    - `x == y` - égalité.
    - `x != y` - Non égal.

- **Opérateur logique ET**

    - `x & y` - logique / bit à bit ET.

- **Opérateur logique XOR**

    - `x ^ y` - Logique / binaire XOR.

- **Opérateur OR logique**

    - `x | y` - logique / bit à bit OU.

- **Conditionnel ET Opérateur**

    - `x && y` - ET logique en court-circuit.

- **Opérateur conditionnel**

    - `x || y` - OU logique en court-circuit.

- **Opérateur de coalescence nulle**

    - `x ?? y` - Retourne `x` s'il n'est pas nul; sinon, retourne `y` .

- **Opérateur conditionnel**

    - `x ? y : z` - Évalue / retourne `y` si `x` est vrai; sinon, évalue `z` .

Contenu connexe

- Opérateur de coalescence nulle
- Opérateur Null-Conditionnel
- nom de l'opérateur

Examples

Opérateurs surchargeables

C # permet aux types définis par l'utilisateur de surcharger les opérateurs en définissant des fonctions membres statiques à l'aide du mot clé `operator` .
L'exemple suivant illustre une implémentation de l'opérateur `+` .

Si nous avons une classe `Complex` qui représente un nombre complexe:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

Et nous voulons ajouter l'option permettant d'utiliser l'opérateur `+` pour cette classe. c'est à dire:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

Nous devrons surcharger l'opérateur `+` pour la classe. Ceci est fait en utilisant une fonction statique et le mot-clé d' `operator` :

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Les opérateurs tels que `+` , `-` , `*` , `/` peuvent tous être surchargés. Cela inclut également les opérateurs qui ne renvoient pas le même type (par exemple, `==` et `!=` Peuvent être surchargés, malgré le retour des booléens). La règle ci-dessous relative aux paires est également appliquée ici.

Les opérateurs de comparaison doivent être surchargés par paires (par exemple, si `<` est surchargé `>` il doit également être surchargé).

Une liste complète des opérateurs surchargeables (ainsi que des opérateurs non surchargeables et des restrictions placées sur certains opérateurs surchargeables) peut être consultée sur MSDN - Overloadable Operators (Guide de programmation C #) .

7.0

la surcharge de l' `operator is` été introduite avec le mécanisme de correspondance de modèle de C # 7.0. Pour plus de détails, voir Correspondance de motif

Étant donné un type `Cartesian` défini comme suit

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

Un `operator is` surchargeable peut par exemple être défini pour `Polar` coordonnées `Polar`

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

qui peut être utilisé comme ça

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(L'exemple est tiré de la documentation Roslyn Pattern Matching Documentation)

Opérateurs relationnels

Équivaut à

Vérifie si les opérandes fournis (arguments) sont égaux

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

Contrairement à Java, l'opérateur de comparaison d'égalité fonctionne en mode natif avec les chaînes.

L'opérateur de comparaison d'égalité fonctionnera avec des opérandes de types différents si une distribution implicite existe de l'un à l'autre. Si aucune distribution implicite appropriée n'existe, vous pouvez appeler une conversion explicite ou utiliser une méthode pour convertir en un type compatible.

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

Contrairement à Visual Basic.NET, l'opérateur de comparaison d'égalité est différent de l'opérateur d'affectation d'égalité.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

Ne pas confondre avec l'opérateur d'affectation (`=`).

Pour les types de valeur, l'opérateur renvoie `true` si les deux opérandes ont la même valeur. Pour les types de référence, l'opérateur renvoie `true` si les deux opérandes sont égaux en *référence* (et non en valeur). Une exception est que les objets de chaîne seront comparés avec une égalité de valeur.

Pas égal

Vérifie si les opérandes fournis *ne* sont *pas* égaux.

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

Cet opérateur renvoie effectivement le résultat opposé à celui de l'opérateur égal (`==`)

Plus grand que

Vérifie si le premier opérande est supérieur au deuxième opérande.

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

Moins que

Vérifie si le premier opérande est inférieur au deuxième opérande.

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.
```

```
var x = 12;
var y = 22;
x < y    //Returns true.
y < x    //Returns false.
```

Supérieur à égal à

Vérifie si le premier opérande est supérieur à la seconde opérande.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

Moins que égal à

Vérifie si le premier opérande est inférieur au deuxième opérande.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

Opérateurs de court-circuit

Par définition, les opérateurs booléens en court-circuit n'évalueront le deuxième opérande que si le premier opérande ne peut pas déterminer le résultat global de l'expression.

Cela signifie que si vous utilisez l'opérateur && comme *firstCondition && secondCondition,* il évaluera *secondCondition* uniquement lorsque *firstCondition* est true et ofcource le résultat global sera vrai seulement si *firstOperand* et *secondOperand* sont évalués à true. Ceci est utile dans de nombreux scénarios, par exemple imaginez que vous vouliez vérifier que votre liste comporte plus de trois éléments, mais vous devez également vérifier si la liste a été initialisée pour ne pas s'exécuter dans *NullReferenceException* . Vous pouvez y parvenir comme ci-dessous:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count> 3 ne sera pas coché jusqu'à ce que myList! = null soit rencontré.

Logique ET

`&&` est la contrepartie de court-circuit de l'opérateur booléen AND (`&`) standard.

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

OU logique

`||` est la contrepartie de court-circuit de l'opérateur OR booléen standard (`|`).

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

Exemple d'utilisation

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

taille de

Renvoie un `int` contenant la taille d'un type * en octets.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

** Ne supporte que certains types primitifs dans un contexte sécurisé.*

Dans un contexte non sécurisé, `sizeof` peut être utilisé pour renvoyer la taille d'autres types et structures primitifs.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Surcharge des opérateurs d'égalité

La surcharge des opérateurs d'égalité ne suffit pas. Dans des circonstances différentes, tous les éléments suivants peuvent être appelés:

1. `object.Equals` et `object.GetHashCode`
2. `IEquatable<T>.Equals` (facultatif, permet d'éviter la boxe)
3. `operator ==` et `operator !=` (optionnel, permet d'utiliser des opérateurs)

Lors de la substitution de `Equals` , `GetHashCode` doit également être remplacé. Lorsque vous implémentez `Equals` , il existe de nombreux cas particuliers: comparer des objets d'un type différent, en les comparant à soi-même, etc.

Lorsque NOT surchargé, la méthode `Equals` et l'opérateur `==` se comportent différemment pour les classes et les structures. Pour les classes, seules les références sont comparées, et pour les structures, les valeurs des propriétés sont comparées par réflexion, ce qui peut avoir un impact négatif sur les performances. `==` ne peut pas être utilisé pour comparer des structures à moins qu'elles ne soient surchargées.

En règle générale, l'opération d'égalité doit obéir aux règles suivantes:

- Ne pas *jeter d'exceptions* .
- Réflexivité: `A` toujours égal à `A` (peut ne pas être vrai pour les valeurs `NULL` dans certains systèmes).
- Transitvity: si `A` est égal à `B` et `B` est égal à `C` , alors `A` est égal à `C`
- Si `A` est égal à `B` , alors `A` et `B` ont des codes de hachage égaux.
- Indépendance de l'arbre d'héritage: si `B` et `C` sont des instances de `Class2` héritées de `Class1` : `Class1.Equals(A,B)` doivent toujours renvoyer la même valeur que l'appel à `Class2.Equals(A,B)` .

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }
```

```
    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Opérateurs membres de classe: Accès membres

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

Opérateurs membres de classe: accès conditionnel nul aux membres

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

Opérateurs membres de classe: invocation de fonction

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

Opérateurs membres de classe: indexation d'objets agrégés

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

Opérateurs membres de classe: indexation conditionnelle nulle

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

Opérateur "exclusif ou"

L'opérateur d'un "exclusif ou" (pour XOR court) est: ^

Cet opérateur renvoie true lorsqu'un, mais un seul, des bools fournis est vrai.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

Opérateurs de transfert de bits

Les opérateurs de décalage permettent aux programmeurs d'ajuster un entier en déplaçant tous ses bits vers la gauche ou la droite. Le diagramme suivant montre l'effet du décalage d'une valeur vers la gauche par un chiffre.

Décalage à gauche

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

Décalage de droite

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

Opérateurs de diffusion implicite et de distribution explicite

C # permet aux types définis par l'utilisateur de contrôler l'affectation et la diffusion via l'utilisation des mots clés `explicit` et `implicit` . La signature de la méthode prend la forme:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

La méthode ne peut pas prendre plus d'arguments, ni être une méthode d'instance. Il peut cependant accéder à tous les membres privés de type défini dans le document.

Un exemple à la fois d'une distribution `implicit` et `explicit` :

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
```

```
    {
        return im._pixels;
    }
}
```

Permettant la syntaxe de distribution suivante:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

Les opérateurs de distribution peuvent travailler dans les deux sens, en allant *de* votre type *à* votre type:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

Enfin, le mot-clé `as` , qui peut être utilisé pour la conversion au sein d'une hiérarchie de types, n'est **pas** valide dans cette situation. Même après avoir défini une distribution `explicit` ou `implicit` , vous ne pouvez pas faire:

```
ColorImage cm = myBinaryImage as ColorImage;
```

Cela générera une erreur de compilation.

Opérateurs binaires avec affectation

C # a plusieurs opérateurs qui peuvent être combinés avec un signe `=` pour évaluer le résultat de l'opérateur, puis attribuer le résultat à la variable d'origine.

Exemple:

```
x += y
```

est le même que

```
x = x + y
```

Opérateurs d'affectation:

- `+=`

- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `^=`
- `<<=`
- `>>=`

? : Opérateur ternaire

Renvoie l'une des deux valeurs en fonction de la valeur d'une expression booléenne.

Syntaxe:

```
condition ? expression_if_true : expression_if_false;
```

Exemple:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

L'opérateur ternaire est associé à droite, ce qui permet d'utiliser des expressions ternaires composées. Ceci est fait en ajoutant des équations ternaires supplémentaires dans la position vraie ou fausse d'une équation ternaire parente. Des précautions doivent être prises pour assurer la lisibilité, mais cela peut être utile dans certaines circonstances.

Dans cet exemple, une opération ternaire composé évalue une `clamp` fonction et retourne la valeur courante si elle est comprise dans l'intervalle, le `min` valeur si elle est inférieure à la plage, ou le `max` de valeur si elle est supérieure à la plage.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Les opérateurs ternaires peuvent également être imbriqués, tels que:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

Lors de l'écriture d'instructions ternaires composées, il est courant d'utiliser des parenthèses ou des indentations pour améliorer la lisibilité.

Les types de *expression_if_true* et *expression_if_false* doivent être identiques ou il doit y avoir une conversion implicite de l'un à l'autre.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

Le type et les conditions de conversion s'appliquent également à vos propres classes.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit
conversion from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough
to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate
to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

Type de

Obtient l'objet `System.Type` pour un type.

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

Pour obtenir le type d'exécution, utilisez la méthode `GetType` pour obtenir le `System.Type` de l'instance actuelle.

Opérateur `typeof` prend un nom de type en paramètre, spécifié lors de la compilation.

```
public class Animal {}
public class Dog : Animal {}
```

```
var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

Opérateur par défaut

Type de valeur (où T: struct)

Les types de données primitifs intégrés, tels que `char` , `int` et `float` , ainsi que les types définis par l'utilisateur déclarés avec `struct` ou `enum` . Leur valeur par défaut est `new T()` :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Type de référence (où T: classe)

Tout type de `class` , `interface` , tableau ou délégué. Leur valeur par défaut est `null` :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

nom de l'opérateur

Renvoie une chaîne qui représente le nom non qualifié d'une `variable` , d'un `type` ou d'un `member` .

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

L'opérateur `nameof` été introduit dans C # 6.0. Il est évalué à la compilation et la valeur de chaîne renvoyée est insérée en ligne par le compilateur. Il peut donc être utilisé dans la plupart des cas où la chaîne constante peut être utilisée (par exemple, les étiquettes de `case` dans une instruction `switch` , les attributs, etc. .). Cela peut être utile dans des cas comme lever et enregistrer des exceptions, des attributs, des liens d'action MVC, etc.

? (Opérateur conditionnel nul)

6,0

Introduit dans C # 6.0 , l'opérateur conditionnel nul `?.` renverra immédiatement `null` si l'expression sur son côté gauche évalue à `null` , au lieu de lancer une `NullReferenceException` . Si son côté gauche donne une valeur non `null` , il est traité comme une normale `.` opérateur. Notez que, comme il peut renvoyer `null` , son type de retour est toujours un type nullable. Cela signifie que pour un type struct ou primitif, il est enveloppé dans un `Nullable<T>` .

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

Cela est pratique lorsque vous tirez des événements. Normalement, vous devez placer l'appel d'événement dans une instruction if en vérifiant la valeur `null` et déclencher l'événement par la suite, ce qui introduit la possibilité d'une situation de concurrence. En utilisant l'opérateur conditionnel Null, ceci peut être résolu de la manière suivante:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Incrémentation et décrémentation du postfixe et du préfixe

L'incrément de postfix `x++` ajoutera `1` à `x`

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix décrément `x--` soustraira un

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` est appelé incrément de préfixe, il incrémente la valeur de x et retourne x alors que `x++` renvoie la valeur de x, puis s'incrémente

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

tandis que

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

les deux sont couramment utilisés dans la boucle

```
for(int i = 0; i < 10; i++)
{
```

```
}
```

=> Opérateur Lambda

3.0

L'opérateur `=>` a la même priorité que l'opérateur d'affectation `=` et est associé à droite.

Il est utilisé pour déclarer des expressions lambda et est également largement utilisé avec les requêtes LINQ :

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

Lorsqu'il est utilisé dans des extensions LINQ ou des requêtes, le type des objets peut généralement être ignoré car il est déduit par le compilateur:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

La forme générale de l'opérateur lambda est la suivante:

```
(input parameters) => expression
```

Les paramètres de l'expression lambda sont spécifiés avant `=>` opérateur, et l'expression / statement / block à exécuter est à droite de l'opérateur:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

Cet opérateur peut être utilisé pour définir facilement des délégués, sans écrire de méthode explicite:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

au lieu de

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Opérateur d'affectation '='

L'opérateur d'affectation `=` définit la valeur de l'opérande de gauche à la valeur de l'opérande de droite et renvoie cette valeur:

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

?? Opérateur de coalescence nulle

L'opérateur Null-Coalescing `??` renverra le côté gauche lorsqu'il n'est pas nul. S'il est nul, il renverra le côté droit.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

Le `??` l'opérateur peut être enchaîné, ce qui permet de supprimer les contrôles `if` .

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Lire Les opérateurs en ligne: https://riptutorial.com/fr/csharp/topic/18/les-operateurs

Chapitre 101: Linq to Objects

Introduction

LINQ to Objects fait référence à l'utilisation de requêtes LINQ avec toute collection IEnumerable.

Examples

Comment LINQ to Object exécute les requêtes

Les requêtes LINQ ne s'exécutent pas immédiatement. Lorsque vous créez la requête, vous stockez simplement la requête pour une exécution ultérieure. Ce n'est que lorsque vous demandez réellement à effectuer une itération de la requête que la requête est exécutée (par exemple, dans une boucle for, lorsque vous appelez ToList, Count, Max, Average, First, etc.)

Ceci est considéré comme une *exécution différée* . Cela vous permet de construire la requête en plusieurs étapes, en la modifiant potentiellement en fonction d'instructions conditionnelles, puis de l'exécuter ultérieurement, une fois que vous avez demandé le résultat.

Vu le code:

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

L'exemple ci-dessus stocke uniquement la requête dans `query` variable de `query` . Il n'exécute pas la requête elle-même.

L'instruction `foreach` force l'exécution de la requête:

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

Certaines méthodes LINQ déclenchent également l'exécution de la requête, `Count` , `First` , `Max` , `Average` . Ils renvoient des valeurs uniques. `ToList` et `ToArray` collectent les résultats et les transforment respectivement en une liste ou en un tableau.

Sachez qu'il est possible de parcourir plusieurs fois la requête si vous appelez plusieurs fonctions LINQ sur la même requête. Cela pourrait vous donner des résultats différents à chaque appel. Si vous souhaitez uniquement travailler avec un ensemble de données, veillez à le sauvegarder dans une liste ou un tableau.

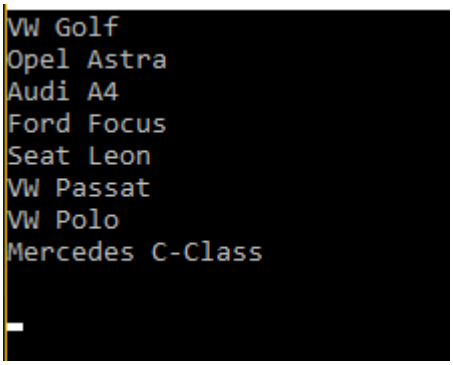
Utiliser LINQ to Objects dans C

Une simple requête SELECT dans Linq

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

Dans l'exemple ci-dessus, un tableau de chaînes (cars) est utilisé comme une collection d'objets à interroger à l'aide de LINQ. Dans une requête LINQ, la clause from vient en premier pour introduire la source de données (cars) et la variable range (car). Lorsque la requête est exécutée, la variable range servira de référence à chaque élément successif dans les voitures. Comme le compilateur peut déduire le type de voiture, vous n'avez pas à le spécifier explicitement

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:

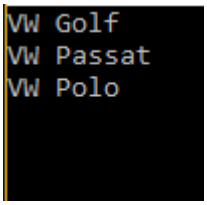


SELECT avec une clause WHERE

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

La clause WHERE est utilisée pour interroger le tableau de chaînes (cars) pour rechercher et renvoyer un sous-ensemble de tableau qui satisfait à la clause WHERE.

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:



Génération d'une liste ordonnée

```
var list = from car in cars
           orderby car ascending
           select car;
```

Parfois, il est utile de trier les données renvoyées. La clause orderby entraînera le tri des éléments en fonction du comparateur par défaut pour le type en cours de tri.

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:

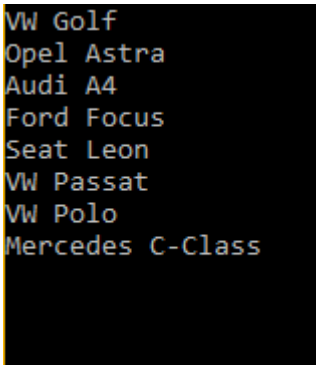
Travailler avec un type personnalisé

Dans cet exemple, une liste saisie est créée, renseignée puis interrogée

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
        var car6 = new Car("VW Passat", 97586);
```

```
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}
```

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:

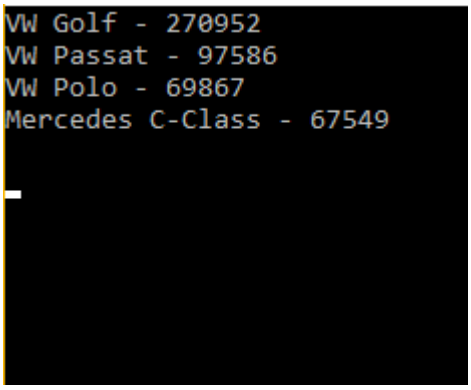


Jusqu'à présent, les exemples ne semblent pas étonnants, car on peut simplement parcourir le tableau pour faire la même chose. Cependant, avec les quelques exemples ci-dessous, vous pouvez voir comment créer des requêtes plus complexes avec LINQ to Objects et obtenir plus avec beaucoup moins de code.

Dans l'exemple ci-dessous, nous pouvons sélectionner des voitures qui ont été vendues à plus de 60000 unités et les classer en fonction du nombre d'unités vendues:

```
var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```

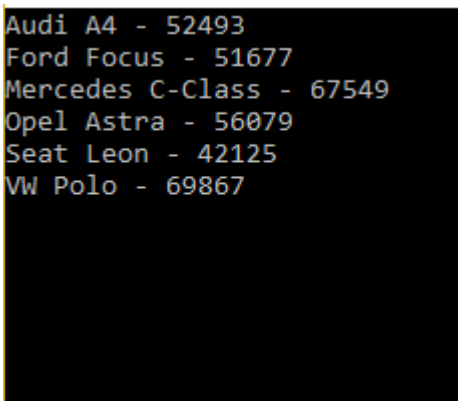
Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:



Dans l'exemple ci-dessous, nous pouvons sélectionner les voitures qui ont vendu un nombre impair d'unités et les classer par ordre alphabétique sur leur nom:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:



Lire Linq to Objects en ligne: https://riptutorial.com/fr/csharp/topic/9405/linq-to-objects

Chapitre 102: LINQ to XML

Examples

Lire le XML en utilisant LINQ to XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

Pour lire ce fichier XML en utilisant LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

Pour accéder à un élément unique

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
```

```
    Console.WriteLine(employee.Element("Name").Value);
}
```

Pour accéder à plusieurs éléments

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
    employee.Element("Name").Value,
    employee.Element("EmpId").Value);
}
```

Pour accéder à tous les éléments ayant un attribut spécifique

```
XElement xelement = XElement.Load("Employees.xml");
var name = from nm in xelement.Root.Elements("Employee")
       where (string)nm.Element("Sex") == "Female"
       select nm;
Console.WriteLine("Details of Female Employees:");
foreach (XElement xEle in name)
Console.WriteLine(xEle);
```

Pour accéder à un élément spécifique ayant un attribut spécifique

```
XElement xelement = XElement.Load("..\\..\\Employees.xml");
var homePhone = from phoneno in xelement.Root.Elements("Employee")
            where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
            select phoneno;
Console.WriteLine("List HomePhone Nos.");
foreach (XElement xEle in homePhone)
{
    Console.WriteLine(xEle.Element("Phone").Value);
}
```

Lire LINQ to XML en ligne: https://riptutorial.com/fr/csharp/topic/2773/linq-to-xml

Chapitre 103: Lire et comprendre les empilements

Introduction

Une trace de pile est une aide précieuse lors du débogage d'un programme. Vous obtiendrez une trace de pile lorsque votre programme lève une exception et parfois lorsque le programme se termine de manière anormale.

Examples

Trace de trace pour une exception NullReferenceException dans Windows Forms

Créons un petit morceau de code qui lance une exception:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

Si nous l'exécutons, nous obtenons les exceptions et traces de pile suivantes:

```
System.NullReferenceException: "Object reference not set to an instance of an object."
   at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

La trace de la pile continue comme ça, mais cette partie suffira à nos fins.

En haut de la pile, nous voyons la ligne:

> à WindowsFormsApplication1.Form1.button1_Click (expéditeur d'objet, EventArgs e) dans F: \ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.cs: ligne 29

C'est la partie la plus importante. Il nous indique la ligne *exacte* où l'Exception s'est produite: ligne 29 dans Form1.cs.
Donc, c'est là que vous commencez votre recherche.

La deuxième ligne est

> à System.Windows.Forms.Control.OnClick (EventArgs e)

C'est la méthode qui a appelé `button1_Click` . Nous savons maintenant que `button1_Click` , où l'erreur s'est produite, a été appelée depuis `System.Windows.Forms.Control.OnClick` .

Nous pouvons continuer comme ça; la troisième ligne est

> à System.Windows.Forms.Button.OnClick (EventArgs e)

C'est à son tour le code appelé `System.windows.Forms.Control.OnClick` .

La trace de la pile est la liste des fonctions appelées jusqu'à ce que votre code rencontre l'Exception. Et en suivant cela, vous pouvez déterminer le chemin d'exécution suivi par votre code jusqu'à ce qu'il rencontre des problèmes!

Notez que la trace de la pile inclut les appels du système .Net; vous n'avez normalement pas besoin de suivre tout le code de Microsofts `System.Windows.Forms` pour savoir ce qui a mal tourné, seul le code qui appartient à votre propre application.

Alors, pourquoi est-ce appelé "trace de pile"?
Parce que chaque fois qu'un programme appelle une méthode, il enregistre sa localisation. Il a une structure de données appelée "pile", où il vide son dernier emplacement.
Si la méthode est exécutée, elle regarde la pile pour voir où elle se trouvait avant d'appeler la méthode - et continue à partir de là.

Ainsi, la pile permet à l'ordinateur de savoir où il s'est arrêté, avant d'appeler une nouvelle méthode.

Mais cela sert aussi d'aide au débogage. Comme un détective qui trace les étapes qu'un criminel a prises pour commettre son crime, un programmeur peut utiliser la pile pour suivre les étapes d'un programme avant qu'il ne s'écroule.

Lire Lire et comprendre les empilements en ligne: https://riptutorial.com/fr/csharp/topic/8923/lire-et-comprendre-les-empilements

Chapitre 104: Littéraux

Syntaxe

- **bool:** vrai ou faux
- **octet:** Aucun, littéral entier implicitement converti à partir d'int
- **sbyte:** Aucun, littéral entier implicitement converti à partir d'int
- **char:** Enveloppez la valeur avec des guillemets simples
- **décimal:** M ou m
- **double:** D, d ou un nombre réel
- **float:** F ou f
- **int:** None, valeur par défaut pour les valeurs intégrales dans la plage de int
- **uint:** U, u ou des valeurs intégrales dans la plage de uint
- **long:** L, l ou des valeurs intégrales dans la plage de long
- **ulong:** UL, ul, Ul, uL, LU, lu, Lu, lU ou des valeurs entières dans la plage de ulong
- **short:** Aucun, littéral entier implicitement converti à partir d'int
- **ushort:** Aucun, littéral entier implicitement converti à partir d'int
- **string:** Enveloppez la valeur avec des guillemets doubles, éventuellement précédés de `@`
- **null** : Le littéral `null`

Examples

int littéraux

`int` littéraux `int` sont définis en utilisant simplement des valeurs intégrales dans la plage de `int` :

```
int i = 5;
```

littéraux uint

`uint` littéraux `uint` sont définis en utilisant le suffixe `U` ou `u` , ou en utilisant une valeur intégrale dans la plage de `uint` :

```
uint ui = 5U;
```

littéraux de chaîne

`string` littéraux de `string` sont définis en entourant la valeur de guillemets `"` :

```
string s = "hello, this is a string literal";
```

Les littéraux de chaîne peuvent contenir des séquences d'échappement. Voir les séquences d'échappement de chaîne

En outre, C # prend en charge les littéraux de chaîne verbatim (voir Chaînes verbatim). Celles-ci sont définies en encapsulant la valeur avec des guillemets doubles `"` , et en l'ajoutant à `@` . Les séquences d'échappement sont ignorées dans les littéraux de chaîne textuelle, et tous les caractères d'espacement sont inclus:

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

littéraux char

`char` littéraux `char` sont définis en encapsulant la valeur avec des guillemets simples `'` :

```
char c = 'h';
```

Les littéraux de caractères peuvent contenir des séquences d'échappement. Voir les séquences d'échappement de chaîne

Un littéral de caractère doit comporter exactement un caractère (après évaluation de toutes les séquences d'échappement). Les littéraux de caractères vides ne sont pas valides. Le caractère par défaut (retourné par `default(char)` ou `new char()`) est `'\0'` , ou le caractère NULL (à ne pas confondre avec le `null` des références et littérale null).

littéraux octets

`byte` type d' `byte` n'a pas de suffixe littéral. Les littéraux entiers sont implicitement convertis à partir de l' `int` :

```
byte b = 127;
```

littéraux sbyte

`sbyte` type `sbyte` n'a pas de suffixe littéral. Les littéraux entiers sont implicitement convertis à partir de l' `int` :

```
sbyte sb = 127;
```

littéraux décimaux

`decimal` littéraux `decimal` sont définis en utilisant le suffixe M ou m sur un nombre réel:

```
decimal m = 30.5M;
```

doubles littéraux

`double` littéraux sont définis en utilisant le suffixe D ou d, ou en utilisant un nombre réel:

```
double d = 30.5D;
```

float littéraux

`float` littéraux `float` sont définis en utilisant le suffixe F ou f, ou en utilisant un nombre réel:

```
float f = 30.5F;
```

littéraux longs

`long` littéraux `long` sont définis en utilisant le suffixe `L` ou `l` , ou en utilisant une valeur intégrale dans la plage de `long` :

```
long l = 5L;
```

ulong littéral

`ulong` littéraux `ulong` sont définis en utilisant le suffixe `UL` , `ul` , `Ul` , `uL` , `LU` , `lu` , `Lu` ou `lU` , ou en utilisant une valeur intégrale dans la plage de `ulong` :

```
ulong ul = 5UL;
```

littéral court

type `short` n'a pas de littéral. Les littéraux entiers sont implicitement convertis à partir de l' `int` :

```
short s = 127;
```

Ushort littéral

`ushort` type `ushort` n'a pas de suffixe littéral. Les littéraux entiers sont implicitement convertis à partir de l' `int` :

```
ushort us = 127;
```

littéraux bool

`bool` littéraux booléens sont `true` ou `false` ;

```
bool b = true;
```

Lire Littéraux en ligne: https://riptutorial.com/fr/csharp/topic/2655/litteraux

Chapitre 105: Manipulation de cordes

Examples

Changer la casse des caractères dans une chaîne

La classe `System.String` prend en charge un certain nombre de méthodes pour convertir des caractères majuscules et minuscules dans une chaîne.

- `System.String.ToLowerInvariant` est utilisé pour renvoyer un objet String converti en minuscule.
- `System.String.ToUpperInvariant` est utilisé pour renvoyer un objet String converti en majuscule.

Remarque: La raison d'utiliser les versions *invariantes* de ces méthodes est d'empêcher la production de lettres inattendues spécifiques à la culture. Ceci est expliqué ici en détail .

Exemple:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Notez que vous *pouvez* choisir de spécifier une **culture** spécifique lors de la conversion en minuscules et en majuscules en utilisant les méthodes String.ToLower (CultureInfo) et String.ToUpper (CultureInfo) en conséquence.

Trouver une chaîne dans une chaîne

En utilisant `System.String.Contains` vous pouvez savoir si une chaîne particulière existe dans une chaîne. La méthode retourne un booléen, true si la chaîne existe, sinon false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

À l'aide de la méthode `System.String.IndexOf` , vous pouvez localiser la position de départ d'une sous-chaîne dans une chaîne existante.
Notez que la position renvoyée est basée sur zéro, une valeur de -1 est renvoyée si la sous-chaîne est introuvable.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

Pour rechercher le premier emplacement à la ***fin*** d'une chaîne, utilisez la méthode `System.String.LastIndexOf` :

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Suppression (rognage) d'un espace blanc d'une chaîne

La méthode `System.String.Trim` peut être utilisée pour supprimer tous les caractères d'espacement de début et de fin d'une chaîne:

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

En outre:

- Pour supprimer l'espace blanc uniquement au *début* d'une chaîne, utilisez: `System.String.TrimStart`
- Pour supprimer les espaces blancs uniquement à la *fin* d'une chaîne, utilisez: `System.String.TrimEnd`

Sous-chaîne pour extraire une partie d'une chaîne.

La méthode `System.String.Substring` peut être utilisée pour extraire une partie de la chaîne.

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

Remplacement d'une chaîne dans une chaîne

À l'aide de la méthode `System.String.Replace` , vous pouvez remplacer une partie d'une chaîne par une autre chaîne.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Toutes les occurrences de la chaîne de recherche sont remplacées:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` peut également être utilisé pour *supprimer une* partie d'une chaîne en spécifiant une chaîne vide comme valeur de remplacement:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Fractionner une chaîne en utilisant un délimiteur

Utilisez la méthode `System.String.Split` pour renvoyer un tableau de chaînes contenant des sous-chaînes de la chaîne d'origine, divisées en fonction d'un délimiteur spécifié:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Sortie:

Un
Deux
Trois
Quatre

Concaténer un tableau de chaînes en une seule chaîne

La méthode `System.String.Join` permet de concaténer tous les éléments d'un tableau de chaînes, en utilisant un séparateur spécifié entre chaque élément:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Concaténation de chaînes

La concaténation de chaînes peut être effectuée à l'aide de la méthode `System.String.Concat` ou (beaucoup plus facile) à l'aide de l'opérateur `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Lire Manipulation de cordes en ligne: https://riptutorial.com/fr/csharp/topic/3599/manipulation-de-cordes

Chapitre 106: Méthodes d'extension

Syntaxe

- public static ReturnType MyExtensionMethod (cette cible TargetType)
- public static ReturnType MyExtensionMethod (cette cible TargetType, TArg1 arg1, ...)

Paramètres

Paramètre	Détails
ce	Le premier paramètre d'une méthode d'extension doit toujours être précédé du mot `this` clé `this` , suivi de l'identifiant avec lequel se référer à l'instance "actuelle" de l'objet que vous étendez.

Remarques

Les méthodes d'extension sont le sucre syntaxique qui permet d'appeler des méthodes statiques sur des instances d'objet comme si elles étaient membres du type lui-même.

Les méthodes d'extension nécessitent un objet cible explicite. Vous devrez utiliser le mot `this` clé `this` pour accéder à la méthode depuis le type étendu lui-même.

Les méthodes d'extensions doivent être déclarées statiques et doivent vivre dans une classe statique.

Quel espace de noms?

Le choix de l'espace de noms pour votre classe de méthode d'extension est un compromis entre visibilité et découverte.

L' option la plus souvent mentionnée est d'avoir un espace de noms personnalisé pour vos méthodes d'extension. Cependant, cela impliquera un effort de communication afin que les utilisateurs de votre code sachent que les méthodes d'extension existent et où les trouver.

Une alternative consiste à choisir un espace de noms tel que les développeurs découvriront vos méthodes d'extension via Intellisense. Donc, si vous voulez étendre la classe `Foo` , il est logique de placer les méthodes d'extension dans le même espace de noms que `Foo` .

Il est important de comprendre que **rien ne vous empêche d'utiliser l'espace de noms "quelqu'un d'autre"** : ainsi, si vous souhaitez étendre `IEnumerable` , vous pouvez ajouter votre méthode d'extension dans l'espace de noms `System.Linq` .

Ce n'est pas *toujours* une bonne idée. Par exemple, dans un cas spécifique, vous souhaiterez peut-être étendre un type commun (`bool IsApproxEqualTo(this double value, double other)` par

exemple), mais ne pas polluer l'ensemble du `System` . Dans ce cas, il est préférable de choisir un espace de noms local spécifique.

Enfin, il est également possible de mettre les méthodes d'extension dans *aucun espace de nommage* !

Une bonne question de référence: comment gérez-vous les espaces de noms de vos méthodes d'extension?

Applicabilité

Des précautions doivent être prises lors de la création de méthodes d'extension pour s'assurer qu'elles conviennent à tous les intrants possibles et ne concernent pas uniquement des situations spécifiques. Par exemple, il est possible d'étendre des classes système telles que `string` , ce qui rend votre nouveau code disponible pour **n'importe quelle** chaîne. Si votre code doit exécuter une logique spécifique à un domaine sur un format de chaîne spécifique à un domaine, une méthode d'extension ne conviendrait pas car sa présence pourrait induire les appelants à travailler avec d'autres chaînes du système.

La liste suivante contient les caractéristiques de base et les propriétés des méthodes d'extension

1. Ce doit être une méthode statique.
2. Il doit être situé dans une classe statique.
3. Il utilise le mot-clé "this" comme premier paramètre avec un type dans .NET et cette méthode sera appelée par une instance de type donnée du côté client.
4. Il a également montré par VS intellisense. Quand on appuie sur le point . après une instance de type, il est dans VS intellisense.
5. Une méthode d'extension doit se trouver dans le même espace de noms que celui utilisé ou vous devez importer l'espace de noms de la classe par une instruction using.
6. Vous pouvez donner n'importe quel nom à la classe qui a une méthode d'extension, mais la classe doit être statique.
7. Si vous souhaitez ajouter de nouvelles méthodes à un type et que vous ne disposez pas du code source correspondant, la solution consiste à utiliser et à implémenter des méthodes d'extension de ce type.
8. Si vous créez des méthodes d'extension ayant les mêmes méthodes de signature que le type que vous étendez, les méthodes d'extension ne seront jamais appelées.

Examples

Méthodes d'extension - aperçu

Les méthodes d'extension ont été introduites dans C # 3.0. Les méthodes d'extension étendent et ajoutent un comportement aux types existants sans créer un nouveau type dérivé, recompiler ou modifier le type d'origine. *Ils sont particulièrement utiles lorsque vous ne pouvez pas modifier la source d'un type que vous souhaitez améliorer.* Des méthodes d'extension peuvent être créées pour les types de systèmes, les types définis par des tiers et les types que vous avez définis vous-

même. La méthode d'extension peut être appelée comme s'il s'agissait d'une méthode membre du type d'origine. Cela permet le **chaînage de méthode** utilisé pour implémenter une **interface Fluent** .

Une méthode d'extension est créée en ajoutant une **méthode statique** à une **classe statique** distincte du type d'origine étendu. La classe statique contenant la méthode d'extension est souvent créée dans le seul but de conserver les méthodes d'extension.

Les méthodes d'extension prennent un premier paramètre spécial qui désigne le type d'origine étendu. Ce premier paramètre est décoré avec le mot - clé `this` (ce qui constitue une utilisation particulière et distincte de `this` en C # -it doit être comprise comme différent de l'utilisation de `this` qui permet référence aux membres de l'instance de l' objet).

Dans l'exemple suivant, le type d'origine en cours d'extension est la `string` classe. `String` a été étendue par une méthode `Shorten()` , qui fournit la fonctionnalité supplémentaire de raccourcissement. La classe statique `StringExtensions` a été créée pour contenir la méthode d'extension. La méthode d'extension `Shorten()` montre qu'il s'agit d'une extension de `string` via le premier paramètre spécialement marqué. Pour montrer que la méthode `Shorten()` étend la `string` , le premier paramètre est marqué avec `this` . Par conséquent, la signature complète du premier paramètre est `this string text` , où `string` est le type d'origine en cours d'extension et `text` est le nom du paramètre choisi.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

Démo en direct sur .NET Fiddle

L'objet transmis en tant que *premier argument d'une méthode d'extension* (qui est accompagné du mot `this` clé `this`) est l'instance à laquelle la méthode d'extension est appelée.

Par exemple, lorsque ce code est exécuté:

```
"some string".Shorten(5);
```

Les valeurs des arguments sont les suivantes:

```
text: "some string"
length: 5
```

Notez que les méthodes d'extension ne sont utilisables que si elles se trouvent dans le même espace de nommage que leur définition, si l'espace de nom est importé explicitement par le code à l'aide de la méthode d'extension ou si la classe d'extension est sans espace de noms. Les directives du framework .NET recommandent de placer les classes d'extension dans leur propre espace de noms. Cependant, cela peut entraîner des problèmes de découverte.

Cela n'entraîne aucun conflit entre les méthodes d'extension et les bibliothèques utilisées, sauf si les espaces de noms susceptibles d'entrer en conflit sont explicitement intégrés. Par exemple, les extensions LINQ :

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
    }
}
```

Démo en direct sur .NET Fiddle

Depuis C # 6.0, il est également possible de placer une directive `using static` dans la *classe* contenant les méthodes d'extension. Par exemple, en `using static System.Linq.Enumerable;` . Cela rend les méthodes d'extension de cette classe particulière disponibles sans amener d'autres types du même espace de noms dans la portée.

Lorsqu'une méthode de classe avec la même signature est disponible, le compilateur le hiérarchise sur l'appel de la méthode d'extension. Par exemple:

```
class Test
{
   public void Hello()
   {
       Console.WriteLine("From Test");
   }
}

static class TestExtensions
{
    public static void Hello(this Test test)
```

```
    {
        Console.WriteLine("From extension method");
    }
}

class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

Démo en direct sur .NET Fiddle

Notez que s'il y a deux fonctions d'extension avec la même signature et que l'une d'entre elles se trouve dans le même espace de nommage, alors celle-ci sera prioritaire. D'un autre côté, si les deux sont accessibles en `using` , alors une erreur de compilation s'ensuivra avec le message:

> **L'appel est ambigu entre les méthodes ou propriétés suivantes**

Notez que la commodité syntaxique d'appeler une méthode d'extension via `originalTypeInstance.ExtensionMethod()` est une commodité facultative. La méthode peut également être appelée de manière traditionnelle, de sorte que le premier paramètre spécial est utilisé comme paramètre de la méthode.

C'est-à-dire les deux travaux suivants:

```
//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

Utiliser explicitement une méthode d'extension

Les méthodes d'extension peuvent également être utilisées comme méthodes classiques de classe statique. Cette façon d'appeler une méthode d'extension est plus détaillée, mais elle est nécessaire dans certains cas.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Usage:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

Quand appeler les méthodes d'extension en tant que méthodes statiques

Il existe encore des scénarios où vous devez utiliser une méthode d'extension comme méthode statique:

- Résoudre un conflit avec une méthode membre. Cela peut se produire si une nouvelle version d'une bibliothèque introduit une nouvelle méthode de membre avec la même signature. Dans ce cas, la méthode membre sera préférée par le compilateur.
- Résoudre les conflits avec une autre méthode d'extension avec la même signature. Cela peut se produire si deux bibliothèques incluent des méthodes d'extension similaires et que les espaces de noms des deux classes avec des méthodes d'extension sont utilisés dans le même fichier.
- Passer la méthode d'extension en tant que groupe de méthodes dans le paramètre délégué.
- Faire votre propre liaison via `Reflection` .
- En utilisant la méthode d'extension dans la fenêtre Immédiat dans Visual Studio.

En utilisant statique

Si une directive `using static` est utilisée pour amener les membres statiques d'une classe statique dans la portée globale, les méthodes d'extension sont ignorées. Exemple:

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

Si vous supprimez le modificateur `this` du premier argument de la méthode `Shorten` , la dernière ligne sera compilée.

Vérification nulle

Les méthodes d'extension sont des méthodes statiques qui se comportent comme des méthodes d'instance. Cependant, contrairement à ce qui se passe lors de l'appel d'une méthode d'instance sur une référence `null` , lorsqu'une méthode d'extension est appelée avec une référence `null` , elle ne lance pas une `NullReferenceException` . Cela peut être très utile dans certains scénarios.

Par exemple, considérez la classe statique suivante:

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}
```

```
string nullString = null;
string emptyString = nullString.EmptyIfNull();// will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null
```

Démo en direct sur .NET Fiddle

Les méthodes d'extension ne peuvent voir que les membres publics (ou internes) de la classe étendue

```
public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}
```

Les méthodes d'extension ne sont qu'un sucre syntaxique et ne font pas partie de la classe qu'elles étendent. Cela signifie qu'ils ne peuvent pas briser l' encapsulation, ils ont seulement accès aux `public` (ou lorsque mis en œuvre dans le même ensemble, `internal`) champs, propriétés et méthodes.

Méthodes d'extension génériques

Tout comme d'autres méthodes, les méthodes d'extension peuvent utiliser des génériques. Par exemple:

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

et l'appel serait comme:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

Voir la démo

De même pour plusieurs arguments de type:

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

L'appeler serait comme:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

Voir la démo

Vous pouvez également créer des méthodes d'extension pour les types partiellement liés dans les types multi-génériques:

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```

L'appeler serait comme:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

Voir la démo

Vous pouvez également spécifier des contraintes de type avec `where` :

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Code d'appel:

```
int number = 5;
var IsDefault = number.IsDefault();
```

Voir la démo

Méthodes d'extension réparties selon le type statique

Le type statique (à la compilation) est utilisé plutôt que le type dynamique (type d'exécution) pour correspondre aux paramètres.

```
public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
    {
        return "Derived";
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;
```

```
        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}
```

Démo en direct sur .NET Fiddle

En outre, l'envoi basé sur le type statique ne permet pas d'appeler une méthode d'extension sur un objet `dynamic` :

```
public class Person
{
    public string Name { get; set; }
}

public static class ExtenionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

Les méthodes d'extension ne sont pas prises en charge par le code dynamique.

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
}

static class StringExtensions
```

```
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

> La raison pour laquelle [appeler des méthodes d'extension à partir d'un code dynamique] ne fonctionne pas est que, dans les méthodes d'extension de code non dynamiques, vous effectuez une recherche complète de toutes les classes connues du compilateur pour une classe statique qui correspond à une méthode d'extension. . La recherche va dans l'ordre en fonction de l'imbrication de l'espace de noms et disponible à l' `using` directives dans chaque espace de noms.
>
> Cela signifie que pour obtenir une invocation de méthode d'extension dynamique résolue correctement, le DLR doit en quelque sorte savoir *au moment de l'exécution* quelles étaient toutes les imbrications et directives d' `using` espace de noms *dans votre code source* . Nous n'avons pas de mécanisme pratique pour encoder toutes ces informations sur le site d'appel. Nous avons envisagé d'inventer un tel mécanisme, mais nous avons décidé qu'il s'agissait d'un coût trop élevé et que nous risquions d'être trop risqués.

La source

Méthodes d'extension en tant qu'encapsuleurs fortement typés

Les méthodes d'extension peuvent être utilisées pour écrire des wrappers fortement typés pour les objets de type dictionnaire. Par exemple un cache, `HttpContext.Items` à cetera ...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

Cette approche supprime la nécessité d'utiliser des littéraux de chaîne en tant que clés sur toute la base de code, ainsi que la nécessité de les convertir au type requis pendant l'opération de lecture. Dans l'ensemble, il crée un moyen plus sécurisé et fortement typé d'interagir avec des objets faiblement typés tels que les dictionnaires.

Méthodes d'extension pour le chaînage

Lorsqu'une méthode d'extension renvoie une valeur ayant le même type que `this` argument, elle peut être utilisée pour "chaîner" un ou plusieurs appels de méthode avec une signature compatible. Cela peut être utile pour les types scellés et / ou primitifs, et permet la création d'API dites "fluides" si les noms des méthodes se lisent comme un langage humain naturel.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

Ou comme ça

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6};
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

Méthodes d'extension en combinaison avec des interfaces

Il est très pratique d'utiliser des méthodes d'extension avec des interfaces car l'implémentation peut être stockée en dehors de la classe et tout ce qu'il faut pour ajouter des fonctionnalités à la classe est de décorer la classe avec l'interface.

```
public interface IInterface
{
   string Do()
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
      //does something with IInterface instance
    }
```

```
}

public class Classy : IInterface
{
   // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
   // provided you import the namespace containing the extension method
   public Do(){
       return this.DoWith();
   }
}
```

utiliser comme:

```
 var classy = new Classy();
 classy.Do(); // will call the extension
 classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

IList Exemple de méthode d'extension: comparaison de 2 listes

Vous pouvez utiliser la méthode d'extension suivante pour comparer le contenu de deux instances IList <T> du même type.

Par défaut, les éléments sont comparés en fonction de leur ordre dans la liste et des éléments eux-mêmes. `isOrdered` false au paramètre `isOrdered` , vous ne comparerez que les éléments eux-mêmes, quel que soit leur ordre.

Pour que cette méthode fonctionne, le type générique (`T`) doit remplacer les méthodes `Equals` et `GetHashCode` .

Usage:

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2);//this gives false
list1.Compare(list2, false);//this gives true. they are equal when the order is disregarded
```

Méthode:

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                 (l1 == null && l2 != null) ||
```

```
                (l1 != null && l2 == null) ||
                (!l1.Equals(l2)))
            {
                return false;
            }
        }
        return true;
    }
    else
    {
        List<T> list2Copy = new List<T>(list2);
        //Can be done with Dictionary without O(n^2)
        for (int i = 0; i < list1.Count; i++)
        {
            if (!list2Copy.Remove(list1[i]))
                return false;
        }
        return true;
    }
}
```

Méthodes d'extension avec énumération

Les méthodes d'extension sont utiles pour ajouter des fonctionnalités aux énumérations.

Une utilisation courante consiste à implémenter une méthode de conversion.

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }
    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

Vous pouvez maintenant convertir rapidement votre valeur enum en un type différent. Dans ce cas un bool.

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

Vous pouvez également utiliser des méthodes d'extension pour ajouter des méthodes de type propriété.

```
public enum Element
```

```
{
    Hydrogen,
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
    Nitrogen,
    Oxygen
    //Etc
}

public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen:  return 1.00794;
            case Element.Helium:    return 4.002602;
            case Element.Lithium:   return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:     return 10.811;
            case Element.Carbon:    return 12.0107;
            case Element.Nitrogen:  return 14.0067;
            case Element.Oxygen:    return 15.9994;
            //Etc
        }
        return double.Nan;
    }
}

var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

Les extensions et les interfaces permettent ensemble le code DRY et les fonctionnalités de type mixin

Les méthodes d'extension vous permettent de simplifier vos définitions d'interface en n'incluant que les fonctionnalités essentielles de l'interface et en définissant des méthodes de commodité et des surcharges comme méthodes d'extension. Les interfaces avec moins de méthodes sont plus faciles à implémenter dans les nouvelles classes. Garder les surcharges comme des extensions plutôt que de les inclure dans l'interface vous évite de copier du code standard dans chaque implémentation, vous aidant ainsi à conserver votre code SEC. Ceci est en fait similaire au modèle de mixin que C # ne supporte pas.

Les extensions de `System.Linq.Enumerable` à `IEnumerable<T>` sont un bon exemple. `IEnumerable<T>` requiert uniquement la classe d'implémentation pour implémenter deux méthodes: `GetEnumerator()` générique et non générique. Mais `System.Linq.Enumerable` fournit d'innombrables utilitaires utiles en tant qu'extensions permettant une consommation claire et concise d' `IEnumerable<T>` .

Ce qui suit est une interface très simple avec des surcharges de commodité fournies en tant qu'extensions.

```
public interface ITimeFormatter
{
```

```
   string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
public class SecondsTimeFormatter : ITimeFormatter
{
   public string Format(TimeSpan span)
   {
       return $"{(int)span.TotalSeconds}s";
   }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is rougly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}
```

Méthodes d'extension pour la gestion de cas particuliers

Les méthodes d'extension peuvent être utilisées pour "masquer" le traitement de règles métier inélégantes qui nécessiteraient autrement d'encombrer une fonction d'appel avec les instructions if / then. Ceci est similaire et analogue à la gestion des valeurs NULL avec les méthodes d'extension. Par exemple,

```
public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty
but are known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() :
cake;
    }
}
```

```
Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake);//Eat the cake, confident that it is not a lie.
```

Utilisation de méthodes d'extension avec des méthodes statiques et des

rappels

Pensez à utiliser des méthodes d'extension comme des fonctions qui encapsulent un autre code. Voici un excellent exemple qui utilise à la fois une méthode statique et une méthode d'extension pour envelopper la construction Try Catch. Faites votre code Bullet Proof ...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
```

```
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
            {
                //error is the already logged and handled exception
                //determine the base result
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

Méthodes d'extension sur les interfaces

Une caractéristique utile des méthodes d'extension est que vous pouvez créer des méthodes communes pour une interface. Normalement, une interface ne peut pas avoir d'implémentations partagées, mais avec des méthodes d'extension possibles.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
```

```
    }
}
```

Dans cet exemple, la méthode `FeetDriven` peut être utilisée sur n'importe quel `IVehicle` . Cette logique de cette méthode s'appliquerait à tous les `IVehicle` , de sorte qu'il est possible de le faire de manière à ce qu'il n'y ait pas de définition `FeetDriven` dans l' `IVehicle` qui serait mise en œuvre de la même manière pour tous les enfants.

Utilisation de méthodes d'extension pour créer de belles classes de mappeurs

Nous pouvons créer de meilleures classes de mappeurs avec des méthodes d'extension, Supposons que je dispose de classes DTO comme

```
public class UserDTO
{
       public AddressDTO Address { get; set; }
}

public class AddressDTO
{
       public string Name { get; set; }
}
```

et je dois mapper aux classes de modèle de vue correspondantes

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

alors je peux créer ma classe de mappeur comme ci-dessous

```
public static class ViewModelMapper
{
      public static UserViewModel ToViewModel(this UserDTO user)
      {
            return user == null ?
                null :
                new UserViewModel()
                {
                    Address = user.Address.ToViewModel()
                    // Job = user.Job.ToViewModel(),
                    // Contact = user.Contact.ToViewModel() .. and so on
                };
      }

      public static AddressViewModel ToViewModel(this AddressDTO userAddr)
      {
```

```
            return userAddr == null ?
                null :
                new AddressViewModel()
                {
                    Name = userAddr.Name
                };
      }
}
```

Puis enfin je peux invoquer mon mappeur comme ci-dessous

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
        };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

La beauté ici est que toutes les méthodes de mappage ont un nom commun (ToViewModel) et que nous pouvons le réutiliser de plusieurs manières

Utilisation de méthodes d'extension pour créer de nouveaux types de collection (par exemple, DictList)

Vous pouvez créer des méthodes d'extension pour améliorer la convivialité des collections imbriquées telles qu'un `Dictionary` avec une valeur `List<T>` .

Considérez les méthodes d'extension suivantes:

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
            where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection>
dict, TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }
```

```
        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

vous pouvez utiliser les méthodes d'extension comme suit:

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15

dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False
```

Voir la démo

Lire Méthodes d'extension en ligne: https://riptutorial.com/fr/csharp/topic/20/methodes-d-extension

Chapitre 107: Méthodes DateTime

Examples

DateTime.Add (TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

DateTime.AddDays (Double)

Ajoutez des jours dans un objet dateTime.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

Vous pouvez également soustraire des jours en passant une valeur négative:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

DateTime.AddHours (Double)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

DateTime.AddMilliseconds (Double)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
```

```
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

DateTime.Compare (DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

DateTime.DaysInMonth (Int32, Int32)

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

DateTime.AddYears (Int32)

Ajouter des années à l'objet dateTime:

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
   Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
```

```
   Console.WriteLine("{0,2} year(s) from now: {1:d}",
                     ctr, baseDate.AddYears(ctr));
```

Fonctions pures d'avertissement lors de l'utilisation de DateTime

Wikipedia définit actuellement une fonction pure comme suit:

1. La fonction évalue toujours la même valeur de résultat avec la même valeur d'argument. La valeur de résultat de la fonction ne peut dépendre d'aucune information ou état masqué pouvant changer pendant l'exécution du programme ou entre différentes exécutions du programme, ni dépendre d'une entrée externe des périphériques d'E / S.
2. L'évaluation du résultat ne provoque aucun effet secondaire ou sortie sémantiquement observable, tel que la mutation d'objets mutables ou la sortie vers des dispositifs d'E / S

En tant que développeur, vous devez être au courant des méthodes pures et vous tomberez souvent sur ces méthodes dans de nombreux domaines. Une des choses que j'ai vu qui mord beaucoup de développeurs débutants est de travailler avec les méthodes de classe DateTime. Beaucoup d'entre eux sont purs et si vous n'êtes pas au courant, vous pouvez être surpris. Un exemple:

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

Compte tenu de l'exemple ci-dessus, on peut s'attendre à ce que le résultat imprimé sur console soit le «26/12/2016», mais en réalité vous vous retrouvez avec la même date. C'est parce que AddDays est une méthode pure et n'affecte pas la date d'origine. Pour obtenir le résultat attendu, vous devez modifier l'appel AddDays comme suit:

```
        sample = sample.AddDays(1);
```

DateTime.Parse (String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

DateTime.TryParse (String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent
and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};
```

```
foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

Parse et TryParse avec informations culturelles

Vous voudrez peut-être l'utiliser lors de l'analyse des DateTimes à partir de différentes cultures (langues) , l'exemple suivant analyse la date néerlandaise.

```
DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}
```

Exemple de Parse:

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

DateTime comme initialiseur dans for-loop

```
// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}
```

Itérer sur un `TimeSpan` fonctionne de la même manière.

DateTime ToString, ToShortDateString, ToLongDateString et ToString formatés

```
using System;
```

```
public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

        Console.WriteLine(date.ToString());        //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy"));    //Outputs: 31/12/2016
    }
}
```

Date actuelle

Pour obtenir la date actuelle, utilisez la propriété `DateTime.Today` . Cela renvoie un objet `DateTime` avec la date du jour. Lorsque ceci est alors converti `.ToString()` cela se fait par défaut dans la localité de votre système.

Par exemple:

```
Console.WriteLine(DateTime.Today);
```

Ecrit la date du jour, dans votre format local sur la console.

DateTime Formating

Formatage de date et heure standard

DateTimeFormatInfo spécifie un ensemble de spécificateurs pour la mise en forme simple de la date et de l'heure. Chaque spécificateur correspond à un modèle de format DateTimeFormatInfo particulier.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"                             ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"                            ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"                          LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"              LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"   FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                            MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                         YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"      RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                 SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
UniversalSortableDateTime
```

Formatage de date / heure personnalisé

Il existe des spécificateurs de format personnalisés suivants:

- `y` (année)
- `M` (mois)
- `d` (jour)
- `h` (heure 12)
- `H` (heure 24)
- `m` (minute)
- `s` (second)
- `f` (deuxième fraction)
- `F` (deuxième fraction, les zéros de fin sont tronqués)
- `t` (PM ou AM)
- `z` (fuseau horaire).

```
var year =       String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"  year
var month =      String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August"  month
var day =        String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =       String.Format("{0:h hh H HH}",     dt); // "6 06 18 18"       hour 12/24
var minute =     String.Format("{0:m mm}",          dt); // "50 50"            minute
var secound =    String.Format("{0:s ss}",          dt); // "23 23"            second
var fraction =   String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300"    sec.fraction
var fraction2 =  String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"       without zeroes
var period =     String.Format("{0:t tt}",          dt); // "P PM"             A.M. or P.M.
var zone =       String.Format("{0:z zz zzz}",      dt); // "+0 +00 +00:00"    time zone
```

Vous pouvez également utiliser le séparateur de date `/` (barre oblique) et le séparateur de temps `:` (deux points).

Pour exemple de code

Pour plus d'informations MSDN .

DateTime.ParseExact (String, String, IFormatProvider)

Convertit la représentation sous forme de chaîne spécifiée d'une date et d'une heure en son équivalent DateTime en utilisant le format et les informations de format spécifiques à la culture spécifiés. Le format de la représentation sous forme de chaîne doit correspondre exactement au format spécifié.

Convertir une chaîne de format spécifique en équivalent DateTime

Disons que nous avons une chaîne de DateTime spécifique à la culture `08-07-2016 11:30:12 PM` comme `MM-dd-yyyy hh:mm:ss tt` le format et nous voulons convertir en équivalent `DateTime` objet

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

Convertir une chaîne de date et heure en objet `DateTime` équivalent sans format de culture

spécifique

Disons que nous avons une chaîne DateTime dans `dd-MM-yy hh:mm:ss tt` format `dd-MM-yy hh:mm:ss tt` et que nous voulons la convertir en objet `DateTime` équivalent, sans aucune information de culture spécifique

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

Convertir une chaîne de date et heure en objet DateTime équivalent sans aucun format de culture spécifique avec un format différent

Disons que nous avons une chaîne de date, exemple comme '23 -12-2016 'ou '12 / 23/2016' et que nous voulons la convertir en objet `DateTime` équivalent, sans aucune information de culture spécifique

```
   string date =  '23-12-2016' or date = 12/23/2016';
   string[] formats = new string[] {"dd-MM-yyyy","MM/dd/yyyy"}; // even can add more possible
formats.
   DateTime date = DateTime.ParseExact(date,formats,
CultureInfo.InvariantCulture,DateTimeStyles.None);
```

REMARQUE: `System.Globalization` doit être ajouté pour la classe CultureInfo

DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime)

Convertit la représentation sous forme de chaîne spécifiée d'une date et d'une heure en son équivalent DateTime en utilisant le format, les informations de format spécifiques à la culture et le style spécifiés. Le format de la représentation sous forme de chaîne doit correspondre exactement au format spécifié. La méthode renvoie une valeur qui indique si la conversion a réussi.

Par exemple

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

Analyser la date sans drapeaux de style.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

```
// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Utilisez des formats personnalisés avec M et MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analyser une chaîne avec des informations de fuseau horaire.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analyser une chaîne représentant UTC.

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None,
out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
  Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Les sorties

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

Lire Méthodes DateTime en ligne: https://riptutorial.com/fr/csharp/topic/1587/methodes-datetime

Chapitre 108: Microsoft.Exchange.WebServices

Examples

Récupérer les paramètres d'absence de l'utilisateur spécifié

Commençons par créer un objet `ExchangeManager` , où le constructeur se connectera aux services pour nous. Il a également une méthode `GetOofSettings` , qui retournera l'objet `OofSettings` pour l'adresse électronique spécifiée:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password =
WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return
redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

Nous pouvons maintenant appeler cela ailleurs comme ceci:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

Mettre à jour les paramètres d'absence du bureau de l'utilisateur

En utilisant la classe ci-dessous, nous pouvons nous connecter à Exchange, puis définir les paramètres d' `UpdateUserOof` du bureau d'un utilisateur spécifique avec `UpdateUserOof` :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Mettez à jour les paramètres utilisateur avec les éléments suivants:

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
```

```
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage,
externalMessage, theUser);
```

Notez que vous pouvez formater les messages à l'aide de balises `html` standard.

Lire Microsoft.Exchange.WebServices en ligne:
https://riptutorial.com/fr/csharp/topic/4863/microsoft-exchange-webservices

Chapitre 109: Mise en cache

Examples

MemoryCache

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

Lire Mise en cache en ligne: https://riptutorial.com/fr/csharp/topic/4383/mise-en-cache

Chapitre 110: Mise en œuvre d'un modèle de conception de décorateur

Remarques

Avantages d'utiliser Decorator:

- vous pouvez ajouter de nouvelles fonctionnalités à l'exécution dans différentes configurations
- bonne alternative pour l'héritage
- le client peut choisir la configuration qu'il veut utiliser

Examples

Simuler une cafétéria

Le décorateur est l'un des modèles de conception structurelle. Il est utilisé pour ajouter, supprimer ou modifier le comportement d'un objet. Ce document vous apprendra à utiliser correctement Decorator DP.

Laissez-moi vous en expliquer l'idée sur un exemple simple. Imaginez que vous êtes maintenant dans Starbobs, célèbre société de café. Vous pouvez passer une commande pour n'importe quel café que vous voulez - avec de la crème et du sucre, avec de la crème et de la garniture et bien plus encore! Mais, la base de toutes les boissons est le café - boisson sombre et amère, vous pouvez modifier. Écrivons un programme simple qui simule une machine à café.

Premièrement, nous devons créer et abstraire un cours qui décrit notre boisson de base:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Maintenant, créons des extras, comme le sucre, le lait et la garniture. Les classes créées doivent implémenter `AbstractCoffee` - elles vont le décorer:

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }
    public override string ShowCoffee()
```

```
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}
public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
        else return "Topping";
    }
}
```

Maintenant, nous pouvons créer notre café préféré:

```
public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee);  //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());

    }
}
```

L'exécution du code produira la sortie suivante:

> Café avec garniture au sucre et au lait

Lire Mise en œuvre d'un modèle de conception de décorateur en ligne: https://riptutorial.com/fr/csharp/topic/4798/mise-en-ouvre-d-un-modele-de-conception-de-decorateur

Chapitre 111: Mise en œuvre d'un modèle de conception de poids mouche

Examples

Mise en œuvre de la carte dans le jeu RPG

La masselotte est l'un des modèles de conception structurelle. Il est utilisé pour réduire la quantité de mémoire utilisée en partageant autant de données que possible avec des objets similaires. Ce document vous apprendra à utiliser correctement Flyweight DP.

Laissez-moi vous en expliquer l'idée sur un exemple simple. Imaginez que vous travaillez sur un jeu de rôle et que vous devez charger un fichier volumineux contenant des caractères. Par exemple:

- `#` est de l'herbe. Vous pouvez marcher dessus.
- `$` est le point de départ
- `@` c'est du rock. Vous ne pouvez pas marcher dessus.
- `%` est le coffre au trésor

Exemple de carte:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@#############@@@@@@#######@#$@@@

@#############@@@@@@#######@###@@@

@########%#######@############@@@

@#############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Comme ces objets ont des caractéristiques similaires, vous n'avez pas besoin de créer un objet distinct pour chaque champ de carte. Je vais vous montrer comment utiliser le poids mouche.

Définissons une interface que nos champs implémenteront:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Maintenant, nous pouvons créer des classes qui représentent nos champs. Nous devons également les identifier en quelque sorte (j'ai utilisé une énumération):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

Comme je l'ai dit, nous n'avons pas besoin de créer d'instance distincte pour chaque champ. Nous devons créer un **référentiel** de champs. L'essence de Flyweight DP est que nous créons dynamiquement un objet uniquement si nous en avons besoin et qu'il n'existe pas encore dans notre repo, ou le retourne s'il existe déjà. Écrivons une classe simple qui va gérer cela pour nous:

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
            default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); //add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
```

```
    }
    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}
```

Génial! Maintenant, nous pouvons tester notre code:

```
public class Program
{
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}
```

Le résultat dans la console doit être:

> Création d'une nouvelle instance de Grass
>
> Création d'une nouvelle instance de Rock

Mais pourquoi l'herbe n'apparaît-elle qu'une seule fois si on veut l'obtenir deux fois? C'est parce que la première fois que nous appelons `GetField` , l'occurrence herbe n'existe pas dans notre **référentiel** , elle est donc créée, mais la prochaine fois que nous aurons besoin d'herbe, elle existe déjà.

Lire Mise en œuvre d'un modèle de conception de poids mouche en ligne: https://riptutorial.com/fr/csharp/topic/4619/mise-en-ouvre-d-un-modele-de-conception-de-poids-mouche

Chapitre 112: Mise en œuvre singleton

Examples

Singleton statiquement initialisé

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

Cette implémentation est thread-safe car dans ce cas `instance` objet `instance` est initialisé dans le constructeur statique. Le CLR garantit déjà que tous les constructeurs statiques sont exécutés en mode thread-safe.

L' `instance` mutation n'est pas une opération thread-safe, par conséquent l'attribut `readonly` garantit l'immuabilité après l'initialisation.

Singleton paresseux et thread-safe (à l'aide du verrouillage à double contrôle)

Cette version thread-safe d'un singleton était nécessaire dans les premières versions de .NET où `static` initialisation `static` n'était pas garantie pour être thread-safe. Dans les versions plus modernes du framework, un singleton statiquement initialisé est généralement préféré car il est très facile de faire des erreurs d'implémentation dans le modèle suivant.

```
public sealed class ThreadSafeSingleton
{
   private static volatile ThreadSafeSingleton instance;
   private static object lockObject = new Object();

   private ThreadSafeSingleton()
   {
   }

   public static ThreadSafeSingleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (lockObject)
            {
               if (instance == null)
               {
                  instance = new ThreadSafeSingleton();
               }
            }
         }

         return instance;
```

```
        }
    }
}
```

Notez que la vérification `if (instance == null)` est effectuée deux fois: une fois avant l'acquisition du verrou et une fois par la suite. Cette implémentation serait toujours thread-safe même sans la première vérification NULL. Cependant, cela signifierait qu'un verrou serait acquis à *chaque fois que* l'instance est demandée, ce qui pourrait nuire à la performance. La première vérification NULL est ajoutée afin que le verrou ne soit pas acquis, sauf si nécessaire. La deuxième vérification NULL garantit que seul le premier thread à acquérir le verrou crée alors l'instance. Les autres threads trouveront l'instance à remplir et passeront à la suite.

Paresseux, Singleton (utilisant Lazy)

Le type .Net 4.0 Lazy garantit l'initialisation d'objet thread-safe, donc ce type pourrait être utilisé pour créer Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

Utiliser `Lazy<T>` fera en sorte que l'objet ne soit instancié que lorsqu'il est utilisé quelque part dans le code d'appel.

Un usage simple sera comme:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Démo en direct sur .NET Fiddle

Paresseux, singleton thread-safe (pour .NET 3.5 ou version antérieure, implémentation alternative)

Parce que dans .NET 3.5 et versions antérieures, vous n'avez pas la classe `Lazy<T>` , vous utilisez le modèle suivant:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {
    }

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

Ceci est inspiré du post du blog de Jon Skeet .

Parce que le `Nested` l'instanciation de l'instance singleton classe est imbriquée privé et ne sera pas déclenchée en accédant à d' autres membres de la `Sigleton` classe (comme une propriété publique en lecture seule, par exemple).

Élimination de l'instance Singleton lorsqu'elle n'est plus nécessaire

La plupart des exemples montrent l'instanciation et la conservation d'un objet `LazySingleton` jusqu'à la fin de l'application propriétaire, même si l'application n'a plus besoin de cet objet. Une solution consiste à implémenter `IDisposable` et à définir l'instance d'objet sur null comme suit:

```
public class LazySingleton : IDisposable
{
    private static volatile Lazy<LazySingleton> _instance;
    private static volatile int _instanceCount = 0;
    private bool _alreadyDisposed = false;

public static LazySingleton Instance
{
    get
    {
        if (_instance == null)
            _instance = new Lazy<LazySingleton>(() => new LazySingleton());
        _instanceCount++;
        return _instance.Value;
    }
}

private LazySingleton() { }
```

```
// Public implementation of Dispose pattern callable by consumers.
public void Dispose()
{
    if (--_instanceCount == 0) // No more references to this object.
    {
       Dispose(true);
       GC.SuppressFinalize(this);
    }
}

// Protected implementation of Dispose pattern.
protected virtual void Dispose(bool disposing)
{
    if (_alreadyDisposed) return;

    if (disposing)
    {
        _instance = null; // Allow GC to dispose of this instance.
        // Free any other managed objects here.
    }

    // Free any unmanaged objects here.
    _alreadyDisposed = true;
}
```

Le code ci-dessus dispose de l'instance avant la fin de l'application, mais uniquement si les consommateurs appellent `Dispose()` sur l'objet après chaque utilisation. Comme il n'y a aucune garantie que cela se produira ou une manière de le forcer, rien ne garantit que l'instance sera jamais éliminée. Mais si cette classe est utilisée en interne, il est plus facile de s’assurer que la méthode `Dispose()` est appelée après chaque utilisation. Un exemple suit:

```
public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}
```

Veuillez noter que cet exemple n'est **pas compatible avec les threads** .

Lire Mise en œuvre singleton en ligne: https://riptutorial.com/fr/csharp/topic/1192/mise-en-ouvre-singleton

Chapitre 113: Modèles de conception créative

Remarques

Les modèles de création visent à séparer un système de la façon dont ses objets sont créés, composés et représentés. Ils augmentent la flexibilité du système en termes de quoi, qui, comment et quand créer un objet. Les modèles de création encapsulent les connaissances sur les classes utilisées par un système, mais cachent les détails de la création et de la mise en forme des instances de ces classes. Les programmeurs ont compris que la composition de systèmes avec héritage rend ces systèmes trop rigides. Les motifs de création sont conçus pour briser ce couplage étroit.

Examples

Motif Singleton

Le modèle Singleton est conçu pour restreindre la création d'une classe à une seule instance.

Ce modèle est utilisé dans un scénario où il est logique de n'avoir qu'un élément comme:

- une classe unique qui orchestre les interactions des autres objets, ex. Classe de gestionnaire
- ou une classe qui représente une ressource unique et unique, ex. Composant d'enregistrement

L'une des méthodes les plus courantes pour implémenter le modèle Singleton **consiste** à `CreateInstance()` une **méthode de fabrique** statique telle que `CreateInstance()` ou `GetInstance()` (ou une propriété statique en C #, `Instance`), conçue pour renvoyer toujours la même instance.

Le premier appel à la méthode ou à la propriété crée et renvoie l'instance Singleton. Par la suite, la méthode retourne toujours la même instance. De cette façon, il n'y a qu'une seule instance de l'objet singleton.

Il est possible d'empêcher la création d'instances via `new` en rendant le constructeur de classe `private`.

Voici un exemple de code typique pour l'implémentation d'un modèle Singleton en C #:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
```

```
    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

Pour illustrer ce modèle, le code ci-dessous vérifie si une instance identique du Singleton est renvoyée lorsque la propriété Instance est appelée plusieurs fois.

```
class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}
```

Note: cette implémentation n'est pas thread-safe.

Pour voir plus d'exemples, y compris comment rendre ce thread sûr, visitez: Mise en œuvre Singleton

Les singletons sont conceptuellement similaires à une valeur globale et provoquent des problèmes et des problèmes de conception similaires. Pour cette raison, le modèle Singleton est largement considéré comme un anti-pattern.

Visitez "Qu'est-ce qui est si mauvais à propos de Singletons?" pour plus d'informations sur les problèmes qui surviennent avec leur utilisation.

En C #, vous avez la possibilité de créer une classe `static` , ce qui rend tous les membres statiques et la classe ne peut pas être instanciée. Compte tenu de cela, il est courant de voir les classes statiques utilisées à la place du modèle Singleton.

Pour les différences clés entre les deux, consultez le modèle C # Singleton versus la classe statique .

Modèle de méthode d'usine

La méthode d'usine est l'un des modèles de conception créative. Il est utilisé pour résoudre le problème de la création d'objets sans spécifier le type de résultat exact. Ce document vous apprendra à utiliser correctement la méthode d'usine DP.

Laissez-moi vous en expliquer l'idée sur un exemple simple. Imaginez que vous travaillez dans une usine qui produit trois types d'appareils: un ampèremètre, un voltmètre et un compteur de résistance. Vous écrivez un programme pour un ordinateur central qui créera le périphérique sélectionné, mais vous ne connaissez pas la décision finale de votre patron sur ce qu'il doit produire.

Créons une interface `IDevice` avec des fonctions communes à tous les périphériques:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Maintenant, nous pouvons créer des classes qui représentent nos appareils. Ces classes doivent implémenter l'interface `IDevice` :

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
```

```
{
    private Random r = null;
    public VoltMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}
```

Maintenant, nous devons définir la méthode d'usine. Créons la classe `DeviceFactory` avec la méthode statique à l'intérieur:

```
public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}
```

Génial! Testons notre code:

```
public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.VOLT);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
```

```
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.OHM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();
    }
}
```

Voici l'exemple de sortie que vous pourriez voir après avoir exécuté ce code:

AmMeter allume ...

36

6

33

43

24

AmMeter clignote des lumières en disant au revoir!

VoltMeter s'allume ...

102

-61

85

138

36

VoltMeter clignote des lumières disant au revoir!

OhmMeter allume ...

723828

368536

685412

800266

578595

OhmMeter clignote des lumières disant au revoir!

Motif de constructeur

Séparez la construction d'un objet complexe de sa représentation afin que le même processus de construction puisse créer différentes représentations et offre un haut niveau de contrôle sur l'assemblage des objets.

Dans cet exemple, illustre le modèle Builder dans lequel différents véhicules sont assemblés de manière progressive. La boutique utilise VehicleBuilders pour construire une variété de véhicules dans une série d'étapes séquentielles.

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

namespace GangOfFour.Builder
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      VehicleBuilder builder;

      // Create shop with vehicle builders
      Shop shop = new Shop();

      // Construct and display vehicles
      builder = new ScooterBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new CarBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new MotorCycleBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
```

```
{
  // Builder uses a complex series of steps
  public void Construct(VehicleBuilder vehicleBuilder)
  {
    vehicleBuilder.BuildFrame();
    vehicleBuilder.BuildEngine();
    vehicleBuilder.BuildWheels();
    vehicleBuilder.BuildDoors();
  }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class VehicleBuilder
{
  protected Vehicle vehicle;

  // Gets vehicle instance
  public Vehicle Vehicle
  {
    get { return vehicle; }
  }

  // Abstract build methods
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
  public MotorCycleBuilder()
  {
    vehicle = new Vehicle("MotorCycle");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "MotorCycle Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}
```

```
/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
  public CarBuilder()
  {
    vehicle = new Vehicle("Car");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Car Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "2500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "4";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "4";
  }
}

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
  public ScooterBuilder()
  {
    vehicle = new Vehicle("Scooter");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Scooter Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "50 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
```

```
    }
  }

  /// <summary>
  /// The 'Product' class
  /// </summary>
  class Vehicle
  {
    private string _vehicleType;
    private Dictionary<string,string> _parts =
      new Dictionary<string,string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
      this._vehicleType = vehicleType;
    }

    // Indexer
    public string this[string key]
    {
      get { return _parts[key]; }
      set { _parts[key] = value; }
    }

    public void Show()
    {
      Console.WriteLine("\n----------------------------");
      Console.WriteLine("Vehicle Type: {0}", _vehicleType);
      Console.WriteLine(" Frame : {0}", _parts["frame"]);
      Console.WriteLine(" Engine : {0}", _parts["engine"]);
      Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
      Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
  }
}
```

Sortie

Type de véhicule: Cadre de scooter: Cadre de scooter
Moteur: aucun
Roues: 2
#Portes: 0

Type de véhicule: voiture
Cadre: cadre de voiture
Moteur: 2500 cc
Roues: 4
#Portes: 4

Type de véhicule: MotorCycle
Cadre: Cadre MotorCycle
Moteur: 500 cc

Roues: 2
#Portes: 0

Motif Prototype

Spécifiez le type d'objets à créer à l'aide d'une instance de prototype et créez de nouveaux objets en copiant ce prototype.

Dans cet exemple, illustre le modèle Prototype dans lequel les nouveaux objets Couleur sont créés en copiant les couleurs préexistantes définies par l'utilisateur du même type.

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

namespace GangOfFour.Prototype
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Prototype Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ColorManager colormanager = new ColorManager();

      // Initialize with standard colors
      colormanager["red"] = new Color(255, 0, 0);
      colormanager["green"] = new Color(0, 255, 0);
      colormanager["blue"] = new Color(0, 0, 255);

      // User adds personalized colors
      colormanager["angry"] = new Color(255, 54, 0);
      colormanager["peace"] = new Color(128, 211, 128);
      colormanager["flame"] = new Color(211, 34, 20);

      // User clones selected colors
      Color color1 = colormanager["red"].Clone() as Color;
      Color color2 = colormanager["peace"].Clone() as Color;
      Color color3 = colormanager["flame"].Clone() as Color;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Prototype' abstract class
  /// </summary>
  abstract class ColorPrototype
  {
    public abstract ColorPrototype Clone();
  }

  /// <summary>
```

```
  /// The 'ConcretePrototype' class
  /// </summary>
  class Color : ColorPrototype
  {
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
      this._red = red;
      this._green = green;
      this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
      Console.WriteLine(
        "Cloning color RGB: {0,3},{1,3},{2,3}",
        _red, _green, _blue);

      return this.MemberwiseClone() as ColorPrototype;
    }
  }

  /// <summary>
  /// Prototype manager
  /// </summary>
  class ColorManager
  {
    private Dictionary<string, ColorPrototype> _colors =
      new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
      get { return _colors[key]; }
      set { _colors.Add(key, value); }
    }
  }
}
```

Sortie:

Couleur de clonage RVB: 255, 0, 0

Couleur de clonage RVB: 128,211,128

Couleur de clonage RVB: 211, 34, 20

Motif d'usine abstraite

Fournir une interface pour créer des familles d'objets associés ou dépendants sans spécifier leurs classes concrètes.

Dans cet exemple montre la création de différents mondes animaux pour un jeu informatique

utilisant différentes usines. Bien que les animaux créés par les usines du continent soient différents, les interactions entre les animaux restent les mêmes.

```
using System;

namespace GangOfFour.AbstractFactory
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }


  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }

  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class AfricaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Lion();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
```

```
/// </summary>
class AmericaFactory : ContinentFactory
{
  public override Herbivore CreateHerbivore()
  {
    return new Bison();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
  public override void Eat(Herbivore h)
```

```
    {
      // Eat Bison
      Console.WriteLine(this.GetType().Name +
        " eats " + h.GetType().Name);
    }
  }

  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

Sortie:

Lion mange des gnous

Le loup mange le bison

Lire Modèles de conception créative en ligne: https://riptutorial.com/fr/csharp/topic/6654/modeles-de-conception-creative

Chapitre 114: Modèles de conception structurelle

Introduction

Les modèles de conception structurelle sont des modèles qui décrivent comment les objets et les classes peuvent être combinés et forment une grande structure et qui facilitent la conception en identifiant un moyen simple de réaliser des relations entre des entités. Il existe sept modèles structurels décrits. Ils sont comme suit: adaptateur, pont, composite, décorateur, façade, poids mouche et proxy

Examples

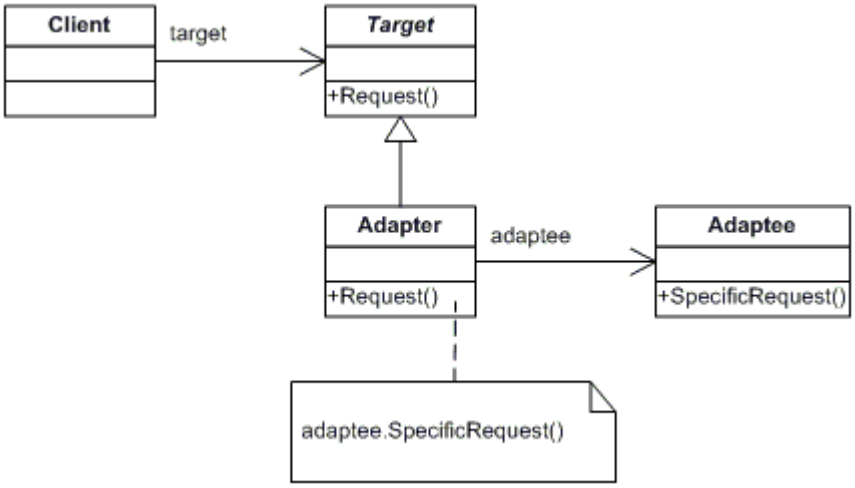
Modèle de conception d'adaptateur

> **«Adapter»,** comme son nom l'indique, est l'objet qui permet à deux interfaces incompatibles entre elles de communiquer entre elles.
>
> **Par exemple:** si vous achetez un iPhone 8 (ou tout autre produit Apple), vous avez besoin de beaucoup d'adaptateurs. Parce que l'interface par défaut ne prend pas en charge l'audio jac ou USB. Avec ces adaptateurs, vous pouvez utiliser des écouteurs avec des fils ou utiliser un câble Ethernet normal. Ainsi, *"deux interfaces incompatibles entre elles communiquent entre elles"* .
>
> **Donc, en termes techniques, cela signifie:** Convertir l'interface d'une classe en une autre interface attendue par un client. L'adaptateur permet aux classes de fonctionner ensemble, ce qui ne pourrait pas être dû à des interfaces incompatibles. Les classes et objets participant à ce modèle sont:

Le modèle d'adaptateur sort 4 éléments

1. **ITarget:** C'est l'interface utilisée par le client pour atteindre les fonctionnalités.
2. **Adaptee:** C'est la fonctionnalité que le client souhaite mais son interface n'est pas compatible avec le client.
3. **Client:** il s'agit de la classe qui souhaite obtenir des fonctionnalités en utilisant le code d'adaptee.
4. **Adapter:** C'est la classe qui implémenterait ITarget et appellerait le code Adaptee que le client veut appeler.

UML



Premier code Exemple (exemple théorique) .

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

Deuxième exemple de code (mise en œuvre du monde réel)

```
/// <summary>
///  Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
```

```
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not
compatible with the client.
/// </summary>
public class CompanyEmplyees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];

        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee’s
code (list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a thirt party and you do'n have any control over it.
     * But it requires a Emplyee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the clietn list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("######### Employee List ##########");
        foreach (var item in employee)
        {
            Console.Write(item);
        }

    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code
which the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmplyees, ITarget
```

```
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}
```

Quand utiliser

- Autoriser un système à utiliser des classes d'un autre système incompatible avec celui-ci.
- Permettre la communication entre les systèmes nouveaux et existants, indépendants les uns des autres
- Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter sont le meilleur exemple de Pattern Adapter.

Lire Modèles de conception structurelle en ligne:
https://riptutorial.com/fr/csharp/topic/9764/modeles-de-conception-structurelle

Chapitre 115: Modificateurs d'accès

Remarques

Si le modificateur d'accès est omis,

- les classes sont par défaut `internal`
- les méthodes sont deault `private`
- les getters et les setters héritent du modificateur de la propriété, par défaut c'est `private`

Les modificateurs d'accès sur les configurateurs ou les accesseurs de propriétés ne peuvent que restreindre l'accès, pas l'élargir: `public string someProperty {get; private set;}`

Examples

Publique

Le mot-clé `public` rend une classe (y compris les classes imbriquées), une propriété, une méthode ou un champ accessible à chaque consommateur:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

privé

Le mot-clé `private` marque les propriétés, les méthodes, les champs et les classes imbriquées à utiliser dans la classe uniquement:

```
public class Foo()
{
    private string someProperty { get; set; }
```

```
    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}
```

interne

Le mot-clé interne rend une classe (y compris les classes imbriquées), la propriété, la méthode ou le champ disponible pour chaque consommateur du même assembly:

```
internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}
```

Cela peut être cassé pour permettre à un assembly de test d'accéder au code via l'ajout de code au fichier AssemblyInfo.cs:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

protégé

Le champ des marques de mot-clé `protected` , les propriétés des méthodes et les classes

imbriquées à utiliser dans la même classe et les classes dérivées uniquement:

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

protégé interne

Le champ de mots-clés `protected internal` , les méthodes, les propriétés et les classes imbriquées à utiliser dans le même assembly ou les classes dérivées d'un autre assembly:

Assemblée 1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty  { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
```

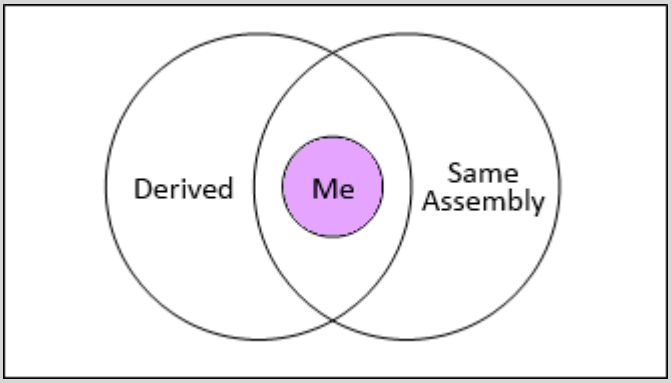
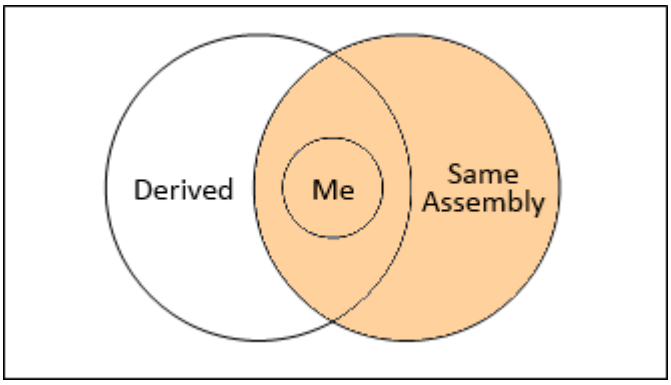
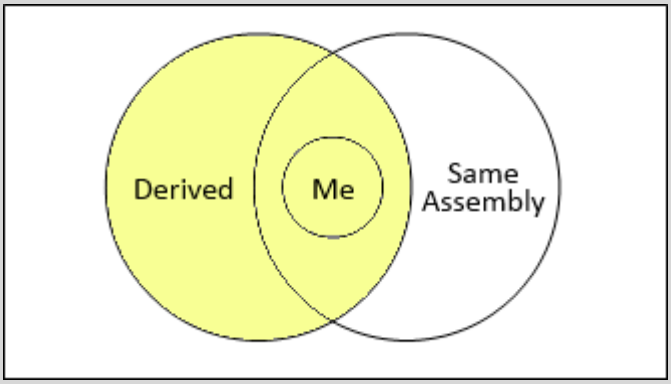
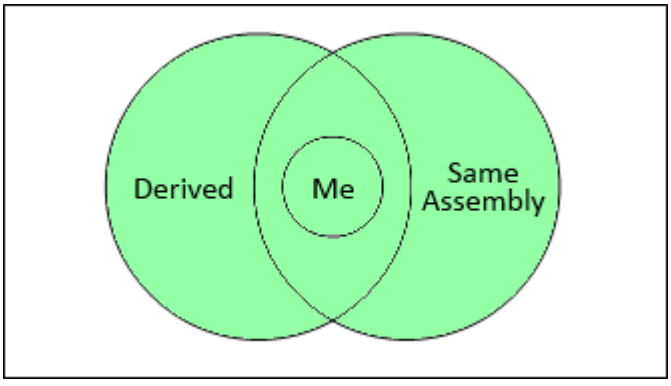
```
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

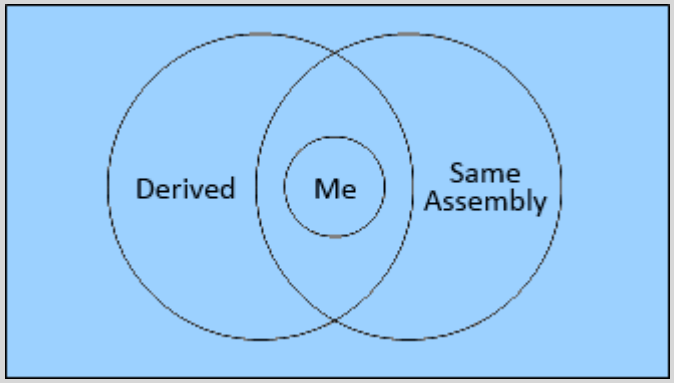
Assemblée 2

```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }

}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

Diagrammes de modificateurs d'accès

Voici tous les modificateurs d’accès dans les diagrammes de venn, du plus limitant au plus accessible:

Modificateur d'accès	Diagramme
privé	
interne	
protégé	
protégé interne	

Modificateur d'accès	Diagramme
Publique	

Vous trouverez ci-dessous plus d'informations.

Lire Modificateurs d'accès en ligne: https://riptutorial.com/fr/csharp/topic/960/modificateurs-d-acces

Chapitre 116: Mots clés

Introduction

Les mots - clés sont des identificateurs réservés et prédéfinis ayant une signification particulière pour le compilateur. Ils ne peuvent pas être utilisés comme identificateurs dans votre programme sans le préfixe `@` . Par exemple, `@if` est un identifiant légal mais pas le mot `if` clé `if` .

Remarques

C # possède une collection prédéfinie de "mots clés" (ou mots réservés) qui ont chacun une fonction spéciale. Ces mots ne peuvent pas être utilisés comme identificateurs (noms pour les variables, les méthodes, les classes, etc.) à moins qu'ils ne soient préfixés par `@` .

- `abstract`
- `as`
- `base`
- `bool`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `checked`
- `class`
- `const`
- `continue`
- `decimal`
- `default`
- `delegate`
- `do`
- `double`
- `else`
- `enum`
- `event`
- `explicit`
- `extern`
- `false`
- `finally`
- `fixed`
- `float`
- `for`
- `foreach`
- `goto`
- `if`
- `implicit`
- `in`
- `int`
- `interface`
- `internal`
- `is`

- lock
- long
- namespace
- new
- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using (directive)
- using (statement)
- virtual
- void
- volatile
- when
- while

En dehors de cela, C # utilise également certains mots-clés pour fournir une signification spécifique au code. Ils sont appelés mots-clés contextuels. Les mots-clés contextuels peuvent être utilisés comme identificateurs et n'ont pas besoin d'être préfixés par `@` lorsqu'ils sont utilisés comme identificateurs.

- add
- alias
- ascending
- async
- await
- descending
- dynamic

- from
- get
- global
- group
- into
- join
- let
- nameof
- orderby
- partial
- remove
- select
- set
- value
- var
- where
- yield

Examples

pilealloc

Le mot-clé `stackalloc` crée une région de mémoire sur la pile et renvoie un pointeur sur le début de cette mémoire. La mémoire allouée par pile est automatiquement supprimée lorsque la portée dans laquelle elle a été créée est fermée.

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
ptr[2] = 232;
...
```

Utilisé dans un contexte dangereux.

Comme avec tous les pointeurs en C #, il n'y a pas de limites à vérifier les lectures et les affectations. La lecture au-delà des limites de la mémoire allouée aura des résultats imprévisibles - elle peut accéder à un emplacement arbitraire dans la mémoire ou provoquer une exception de violation d'accès.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

La mémoire allouée par pile est automatiquement supprimée lorsque la portée dans laquelle elle a été créée est fermée. Cela signifie que vous ne devez jamais renvoyer la mémoire créée avec stackalloc ou la stocker au-delà de la durée de vie de l'étendue.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
     //ptr is now an invalid pointer, using it in any way will have
     //unpredictable results. This is exactly the same as accessing beyond
     //the bounds of the pointer.
     var ptr = Leak();
}
```

`stackalloc` ne peut être utilisé que lors de la déclaration *et de l'* initialisation de variables. Ce qui suit n'est *pas* valide:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Remarques:

`stackalloc` ne doit être utilisé que pour des optimisations de performances (pour le calcul ou l'interopérabilité). Cela est dû au fait que:

- Le ramasse-miettes n'est pas nécessaire car la mémoire est allouée sur la pile plutôt que sur le tas - la mémoire est libérée dès que la variable est hors de portée
- Il est plus rapide d'allouer de la mémoire sur la pile plutôt que sur le tas
- Augmenter les chances de succès du cache sur le processeur en raison de la localisation des données

volatil

L'ajout du mot-clé `volatile` à un champ indique au compilateur que la valeur du champ peut être modifiée par plusieurs threads distincts. Le principal objectif du mot-clé `volatile` est d'empêcher les optimisations du compilateur qui supposent uniquement un accès par thread unique. L'utilisation de `volatile` garantit que la valeur du champ est la valeur la plus récente disponible et que la valeur n'est pas soumise à la mise en cache des valeurs non volatiles.

Il est recommandé de marquer *chaque variable* pouvant être utilisée par plusieurs threads comme `volatile` pour éviter tout comportement inattendu dû à des optimisations en arrière-plan. Considérons le bloc de code suivant:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;
```

```
        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Dans le bloc de code ci-dessus, le compilateur lit les instructions `x = 5` et `y = x + 10` et détermine que la valeur de `y` finira toujours par 15. Ainsi, il optimisera la dernière instruction comme `y = 15` . Cependant, la variable `x` est en fait un champ `public` et la valeur de `x` peut être modifiée à l'exécution via un thread différent agissant séparément sur ce champ. Considérons maintenant ce bloc de code modifié. Notez que le champ `x` est maintenant déclaré comme `volatile` .

```
public class Example
{
    public volatile int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Maintenant, le compilateur recherche les utilisations en *lecture* du champ `x` et s'assure que la valeur actuelle du champ est toujours extraite. Cela garantit que même si plusieurs threads lisent et écrivent dans ce champ, la valeur actuelle de `x` est toujours extraite.

`volatile` ne peut être utilisé que sur les champs de la `class` es ou de la `struct` s. Ce qui suit n'est *pas* valide :

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile` ne peut être appliqué qu'aux champs des types suivants:

- types de référence ou paramètres de type génériques connus pour être des types de référence
- types primitifs tels que `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `char` , `float` et `bool`
- types d' `sbyte` basés sur `byte` , `sbyte` , `short` , `ushort` , `int` ou `uint`
- `IntPtr` et `UIntPtr`

Remarques:

- Le modificateur `volatile` est généralement utilisé pour un champ accessible par plusieurs threads sans utiliser l'instruction de verrouillage pour sérialiser l'accès.
- Le mot clé `volatile` peut être appliqué aux types de référence
- Le mot-clé `volatile` ne fonctionnera pas sur les primitives 64 bits sur une plate-forme 32 bits atomique. Les opérations verrouillées telles que `Interlocked.Read` et `Interlocked.Exchange` doivent toujours être utilisées pour un accès multithread sécurisé sur ces plates-formes.

fixé

L'instruction fixe corrige la mémoire dans un emplacement. Les objets en mémoire sont généralement déplacés, ce qui rend le ramassage des ordures possible. Mais lorsque nous utilisons des pointeurs non sécurisés vers des adresses de mémoire, cette mémoire ne doit pas être déplacée.

- Nous utilisons l'instruction fixed pour garantir que le garbage collector ne déplace pas les données de chaîne.

Variables fixes

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

Utilisé dans un contexte dangereux.

Taille de tableau fixe

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` ne peut être utilisé que sur les champs d'une `struct` (doit également être utilisé dans un contexte non sécurisé).

défaut

Pour les classes, interfaces, délégués, tableaux, nullables (tels que int?) Et les types de pointeurs, la `default(TheType)` renvoie `null` :

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

Pour structs et enums, `default(TheType)` renvoie la même chose que `new TheType()` :

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` peut être particulièrement utile lorsque `T` est un paramètre générique pour lequel aucune contrainte n'est présente pour décider si `T` est un type de référence ou un type de valeur, par exemple:

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

lecture seulement

Le mot clé `readonly` est un modificateur de champ. Lorsqu'une déclaration de champ inclut un modificateur `readonly` , les affectations à ce champ ne peuvent avoir lieu que dans le cadre de la déclaration ou dans un constructeur de la même classe.

Le mot clé `readonly` est différent du mot clé `const` . Un champ `const` ne peut être initialisé qu'à la déclaration du champ. Un champ `readonly` peut être initialisé à la déclaration ou dans un constructeur. Par conséquent, les champs en `readonly` peuvent avoir des valeurs différentes selon le constructeur utilisé.

Le mot-clé `readonly` est souvent utilisé lors de l'injection de dépendances.

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";

    Person(string name)
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

> Note: La déclaration d'un champ *readonly* n'implique pas l' *immuabilité* . Si le champ est un *type de référence,* le **contenu** de l'objet peut être modifié. *Readonly* est généralement utilisé pour empêcher que l'objet soit **écrasé** et assigné uniquement lors de l' **instanciation** de cet objet.
>
> Remarque: Dans le constructeur, un champ en lecture seule peut être réaffecté

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

comme

Le mot clé `as` est un opérateur similaire à une *distribution* . Si un transtypage n'est pas possible, utilisez `as` produit `null` plutôt que de provoquer une `InvalidCastException` .

`expression as type` est équivalent à `expression is type ? (type)expression : (type)null` avec la mise en garde `as` est valable uniquement sur les conversions de référence, les conversions nullables et les conversions de boxe. Les conversions définies par l'utilisateur *ne* sont *pas* prises en charge. un casting régulier doit être utilisé à la place.

Pour l'expansion ci-dessus, le compilateur génère du code tel que l' `expression` ne sera évaluée qu'une seule fois et utilisera une vérification de type dynamique unique (contrairement aux deux dans l'exemple ci-dessus).

`as` peut être utile lorsque vous attendez un argument pour faciliter plusieurs types. Plus précisément, il accorde à l'utilisateur plusieurs options - plutôt que de vérifier chaque possibilité

avec `is` avant de lancer, ou simplement de jeter et de rattraper les exceptions. Il est recommandé d'utiliser «as» lors de la conversion / vérification d'un objet, ce qui entraînera une seule pénalité de déballage. Utiliser `is` pour vérifier, alors le lancer entraînera deux pénalités de unboxing.

Si un argument est censé être une instance d'un type spécifique, une distribution régulière est préférable car son objectif est plus clair pour le lecteur.

Comme un appel à `as` peut produire une valeur `null` , vérifiez toujours le résultat pour éviter une `NullReferenceException` .

Exemple d'utilisation

```
object something = "Hello";
Console.WriteLine(something as string);        //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);          //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

Démo en direct sur .NET Fiddle

Exemple équivalent sans utiliser `as` :

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

Ceci est utile lors de la `Equals` fonction `Equals` dans les classes personnalisées.

```
class MyCustomClass
{

    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;

        // if it is null it may be really null
        // or it may be of a different type
        if (Object.ReferenceEquals(null, customObject))
        {
            // If it is null then it is not equal to this instance.
            return false;
        }

        // Other equality controls specific to class
    }

}
```

est

Vérifie si un objet est compatible avec un type donné, par exemple si un objet est une instance du type `BaseInterface` ou un type dérivant de `BaseInterface` :

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

Si l'intention de la distribution est d'utiliser l'objet, il est préférable d'utiliser le `as` mot - clé »

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True - valid use of 'is'
Console.WriteLine(d is BaseClass);     // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); //prefered method since you incur only one unboxing penalty
}
```

Mais, à partir de C # 7 `pattern matching` fonctionnalité de `pattern matching` étend l'opérateur is pour rechercher un type et déclarer une nouvelle variable en même temps. Même partie de code avec C # 7:

7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

Type de

Renvoie le `Type` d'un objet sans qu'il soit nécessaire de l'instancier.

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
```

```
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

const

`const` est utilisé pour représenter des valeurs qui **ne changeront jamais** pendant la durée de vie du programme. Sa valeur est constante à la **compilation** , par opposition au mot clé `readonly` , dont la valeur est constante à l'exécution.

Par exemple, comme la vitesse de la lumière ne changera jamais, nous pouvons la stocker dans une constante.

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

C'est essentiellement la même chose que d'avoir une `return mass * 299792458 * 299792458` , car le compilateur remplacera directement `c` par sa valeur constante.

Par conséquent, `c` ne peut pas être modifié une fois déclaré. Ce qui suit produira une erreur de compilation:

```
const double c = 299792458;  // Speed of light

c = 500;  //compile-time error
```

Une constante peut être préfixée avec les mêmes modificateurs d'accès que les méthodes:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` membres `const` sont `static` par nature. Cependant, l'utilisation de `static` explicitement n'est pas autorisée.

Vous pouvez également définir des constantes locales à la méthode:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

Celles-ci ne peuvent pas être précédées d'un mot-clé `private` ou `public` , car elles sont implicitement locales à la méthode dans laquelle elles sont définies.

Tous les types ne peuvent pas être utilisés dans une déclaration `const` . Les types de valeur

autorisés sont les types prédéfinis `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` et tous les types `enum` . Essayer de déclarer des membres `const` avec d'autres types de valeur (tels que `TimeSpan` ou `Guid`) échouera à la compilation.

Pour la `string` type de référence prédéfinie spéciale, les constantes peuvent être déclarées avec n'importe quelle valeur. Pour tous les autres types de référence, les constantes peuvent être déclarées mais doivent toujours avoir la valeur `null` .

Comme les valeurs `const` sont connues à la compilation, elles sont autorisées en tant qu'étiquettes de `case` dans une instruction `switch` , en tant qu'arguments standard pour les paramètres facultatifs, en tant qu'arguments pour attribuer des spécifications, etc.

Si les valeurs `const` sont utilisées dans différents assemblys, il faut faire attention au contrôle de version. Par exemple, si l'assembly A définit un `public const int MaxRetries = 3;` , et l'assembly B utilise cette constante, puis si la valeur de `MaxRetries` est modifiée ultérieurement à `5` dans l'assembly A (qui est ensuite `MaxRetries`), cette modification ne sera pas effective dans l'assembly B, *sauf si l'* assembly B est également recompilé (avec une référence à la nouvelle version de A).

Pour cette raison, si une valeur peut changer dans les révisions futures du programme et si la valeur doit être visible publiquement, ne déclarez pas cette valeur `const` sauf si vous savez que tous les assemblys dépendants seront recompilés chaque fois que quelque chose est modifié. L'alternative consiste à utiliser `static readonly` au lieu de `const` , qui est résolu à l'exécution.

espace de noms

Le mot-clé `namespace` est une structure d'organisation qui nous aide à comprendre comment une base de code est organisée. Les espaces de noms en C # sont des espaces virtuels plutôt que dans un dossier physique.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

Les espaces de noms en C # peuvent également être écrits en syntaxe chaînée. Ce qui suit est équivalent à ci-dessus:

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

essayer, attraper, enfin lancer

`try` , `catch` , `finally` , et `throw` vous permettent de gérer les exceptions dans votre code.

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block
that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try
has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}
```

Remarque: Le mot-clé `return` peut être utilisé dans le bloc `try` , et le bloc `finally` sera toujours exécuté (juste avant le retour). Par exemple:

```
try
{
    connection.Open();
    return connection.Get(query);
}
```

```
finally
{
    connection.Close();
}
```

L'instruction `connection.Close()` sera exécutée avant que le résultat de `connection.Get(query)` soit renvoyé.

continuer

Transmettez immédiatement le contrôle à la prochaine itération de la construction de boucle englobante (for, foreach, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Sortie:

5
6
7
8
9

Démo en direct sur .NET Fiddle

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Sortie:

une
b
c
ré

Démo en direct sur .NET Fiddle

ref, out

Les mots-clés `ref` et `out` provoquent un argument par référence, et non par valeur. Pour les types de valeur, cela signifie que la valeur de la variable peut être modifiée par l'appelé.

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

Pour les types de référence, l'instance dans la variable peut non seulement être modifiée (comme c'est le cas sans la `ref`), mais elle peut également être complètement remplacée:

```
Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now
```

La principale différence entre le mot-clé `out` et `ref` est que `ref` exige que la variable soit initialisée par l'appelant, tandis que `out` passe cette responsabilité à l'appelé.

Pour utiliser un paramètre `out` , la définition de la méthode et la méthode appelante doivent utiliser explicitement le mot clé `out` .

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number);  // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number);  // number = 34

void AddOneByRef(ref int value)
{
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

Démo en direct sur .NET Fiddle

Ce qui suit ne compile *pas* , car `out` paramètres `out` doivent avoir une valeur assignée avant que la méthode ne retourne (elle devrait être compilée avec `ref`):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

utiliser un mot clé comme modificateur générique

`out` mot clé `out` peut également être utilisé dans les paramètres de type générique lors de la définition des interfaces génériques et des délégués. Dans ce cas, le mot-clé `out` indique que le paramètre type est covariant.

> La covariance vous permet d'utiliser un type plus dérivé que celui spécifié par le paramètre générique. Cela permet la conversion implicite des classes implémentant des interfaces variantes et la conversion implicite des types de délégué. La covariance et la contravariance sont prises en charge pour les types de référence, mais elles ne sont pas prises en charge pour les types de valeur. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

coché, non coché

Les mots clés `checked` et `unchecked` `checked` définissent la manière dont les opérations gèrent les dépassements mathématiques. Le "dépassement" dans le contexte des mots-clés `checked` et `unchecked` se produit lorsqu'une opération arithmétique de type entier génère une valeur dont l'amplitude est supérieure à celle que le type de données cible peut représenter.

Lorsque le débordement se produit dans un bloc `checked` (ou lorsque le compilateur est configuré pour utiliser l'arithmétique vérifiée globalement), une exception est émise pour avertir d'un comportement indésirable. Pendant ce temps, dans un bloc `unchecked` , le dépassement de capacité est silencieux: aucune exception n'est levée et la valeur est simplement contournée par la limite opposée. Cela peut conduire à des bogues subtils et difficiles à trouver.

Comme la plupart des opérations arithmétiques sont effectuées sur des valeurs trop petites ou trop grandes pour déborder, la plupart du temps, il n'est pas nécessaire de définir explicitement un bloc comme `checked` . Des précautions doivent être prises lors du calcul arithmétique sur des entrées non limitées susceptibles de provoquer un débordement, par exemple lors du calcul arithmétique dans des fonctions récursives ou lors de la saisie d'une entrée utilisateur.

Ni `checked` ni `unchecked` n'affecte les opérations arithmétiques en virgule flottante.

Lorsqu'un bloc ou une expression est déclaré `unchecked` , toutes les opérations arithmétiques qu'il contient sont autorisées à déborder sans provoquer d'erreur. Un exemple où ce comportement est *souhaité* serait le calcul d'une somme de contrôle, où la valeur est autorisée à "boucler" pendant le calcul:

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
```

```
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

L'une des utilisations les plus courantes de la `object.GetHashCode() unchecked` est l'implémentation d'une substitution personnalisée pour `object.GetHashCode()` , un type de somme de contrôle. Vous pouvez voir l'utilisation du mot clé dans les réponses à cette question: Quel est le meilleur algorithme pour un System.Object.GetHashCode surchargé? .

Lorsqu'un bloc ou une expression est déclaré comme `checked` , toute opération arithmétique qui provoque un dépassement entraîne une `OverflowException` .

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Les deux cochés et non cochés peuvent être sous forme de bloc et d'expression.

Les blocs cochés et non contrôlés n'affectent pas les méthodes appelées, seuls les opérateurs appelés directement dans la méthode actuelle. Par exemple, les `Enum.ToObject()` , `Convert.ToInt32()` et les opérateurs définis par l'utilisateur ne sont pas affectés par les contextes vérifiés / non vérifiés personnalisés.

***Remarque** : Le comportement par défaut du débordement par défaut (coché ou non) peut être modifié dans les **propriétés** du **projet** ou via le commutateur de ligne de commande **/ checked [+ | -]** . Il est courant de vérifier les opérations pour les versions de débogage et de les désactiver pour les versions validées. Les mots clés `checked` et `unchecked` `checked` `unchecked` seraient alors utilisés que lorsqu'une approche par défaut ne s'applique pas et que vous avez besoin d'un comportement explicite pour garantir l'exactitude.*

aller à

`goto` peut être utilisé pour accéder à une ligne spécifique du code, spécifiée par une étiquette.

`goto` tant que:

Étiquette:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

Démo en direct sur .NET Fiddle

Déclaration de cas:

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}
```

Démo en direct sur .NET Fiddle

Ceci est particulièrement utile pour exécuter plusieurs comportements dans une instruction switch, car C # ne prend pas en charge les blocs de cas en cascade .

Réessayer d'exception

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

Démo en direct sur .NET Fiddle

Semblable à beaucoup de langues, l'utilisation du mot-clé goto est déconseillée sauf les cas ci-dessous.

Les utilisations valides de `goto` qui s'appliquent à C #:

- Cas de chute dans la déclaration de changement.
- Pause à plusieurs niveaux. LINQ peut souvent être utilisé à la place, mais ses performances sont généralement moins bonnes.

- Désallocation de ressources lors de l'utilisation d'objets de bas niveau non emballés. En C #, les objets de bas niveau doivent généralement être regroupés dans des classes distinctes.
- Machines à états finis, par exemple, analyseurs syntaxiques; utilisé en interne par les machines asynchrones / attendues générées par le compilateur.

enum

Le mot clé `enum` indique au compilateur que cette classe hérite de la classe abstraite `Enum` , sans que le programmeur ait à l'hériter explicitement. `Enum` est un descendant de `ValueType` , destiné à être utilisé avec un ensemble distinct de constantes nommées.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

Vous pouvez éventuellement spécifier une valeur spécifique pour chacun (ou certains d'entre eux):

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

Dans cet exemple, j'ai omis une valeur pour 0, c'est généralement une mauvaise pratique. Une `enum` aura toujours une valeur par défaut produite par conversion explicite `(YourEnumType) 0` , où `YourEnumType` est votre type d' `enume` déclaré. Sans une valeur de 0 définie, un `enum` n'aura pas de valeur définie à l'initiation.

Le type sous-jacent de `enum` est `int` , vous pouvez changer le type sous-jacent en n'importe quel type entier, y compris `byte` , `sbyte` , `short` , `ushort` , `int` , `uint` , `long` et `ulong` . Vous trouverez ci-dessous un enum avec le type sous-jacent `byte` :

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Notez également que vous pouvez convertir vers / à partir du type sous-jacent simplement avec une distribution:

```
int value = (int)NotableYear.EndOfWwI;
```

Pour ces raisons, vous devriez toujours vérifier si une `enum` est valide lorsque vous exposez les fonctions de la bibliothèque:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

base

Le mot clé de **`base`** est utilisé pour accéder aux membres d'une classe de base. Il est couramment utilisé pour appeler les implémentations de base des méthodes virtuelles ou pour spécifier quel constructeur de base doit être appelé.

Choisir un constructeur

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in
Child's constructor
        Console.WriteLine(message);
    }
}
```

Implémentation de la base d'appel de la méthode virtuelle

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

Il est possible d'utiliser le mot clé base pour appeler une implémentation de base à partir de n'importe quelle méthode. Cela lie l'appel de méthode directement à l'implémentation de base, ce qui signifie que même si les nouvelles classes enfant remplacent une méthode virtuelle, l'implémentation de base sera toujours appelée. Cela doit donc être utilisé avec prudence.

```
public class Parent
{
    public virtual int VirtualMethod()
```

```
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());

        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());

        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}
```

pour chaque

`foreach` est utilisé pour parcourir les éléments d'un tableau ou les éléments d'une collection qui implémente `IEnumerable` †.

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

Cela va sortir

> "Bonjour le monde!"
> "Comment ça va aujourd'hui?"
> "Au revoir"

Démo en direct sur .NET Fiddle

Vous pouvez quitter la boucle `foreach` à tout moment en utilisant le mot-clé break ou passer à l'itération suivante à l'aide du mot-clé continue .

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}

// Prints: 1, 3, 4,
```

Démo en direct sur .NET Fiddle

Notez que l'ordre d'itération n'est garanti *que* pour certaines collections telles que les tableaux et la `List` , mais **n'est pas** garanti pour de nombreuses autres collections.

---

† Alors que `IEnumerable` est généralement utilisé pour indiquer des collections énumérables, `foreach` exige uniquement que la collection expose publiquement la méthode de l' `object GetEnumerator()` , qui doit renvoyer un objet exposant la méthode `bool MoveNext()` et l' `object Current { get; }` propriété.

params

`params` permet à un paramètre de méthode de recevoir un nombre variable d'arguments, c'est-à-dire zéro, un ou plusieurs arguments sont autorisés pour ce paramètre.

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

Cette méthode peut maintenant être appelée avec une liste typique d'arguments `int` , ou un tableau d'ints.

```
AddAll(5, 10, 15, 20);                // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

`params` doivent apparaître au plus une fois et, s'ils sont utilisés, ils doivent être en **dernier** dans la liste des arguments, même si le type suivant est différent de celui du tableau.

Soyez prudent lorsque vous surchargez les fonctions lorsque vous utilisez le mot-clé `params` . C # préfère faire correspondre les surcharges plus spécifiques avant de tenter d'utiliser des surcharges avec des `params` . Par exemple si vous avez deux méthodes:

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

Ensuite, la surcharge de 2 arguments spécifique aura la priorité avant d'essayer la surcharge de `params` .

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

Pause

Dans une boucle (for, foreach, do, while) l'instruction `break` interrompt l'exécution de la boucle la plus interne et retourne au code qui la suit. Il peut également être utilisé avec le `yield` dans lequel il spécifie qu'un itérateur a pris fin.

```
for (var i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

Démo en direct sur .NET Fiddle

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

L'instruction break est également utilisée dans les constructions à casse pour sortir d'un cas ou d'un segment par défaut.

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

Dans les instructions switch, le mot clé 'break' est requis à la fin de chaque instruction de cas. Ceci est contraire à certains langages qui permettent de passer à la prochaine déclaration de cas de la série. Des solutions de contournement pour cela incluraient des instructions «goto» ou empiler les instructions «case» de manière séquentielle.

Le code suivant donnera les numéros `0, 1, 2, ..., 9` et la dernière ligne ne sera pas exécutée. `yield break` signifie la fin de la fonction (pas simplement une boucle).

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

Démo en direct sur .NET Fiddle

Notez que contrairement à d'autres langages, il n'existe aucun moyen d'étiqueter une pause particulière en C #. Cela signifie que dans le cas de boucles imbriquées, seule la boucle la plus interne sera arrêtée:

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
```

```
    {
        if (innerItem.ShoudBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
    }
}
```

Si vous souhaitez sortir de la boucle *externe* , vous pouvez utiliser l'une des stratégies suivantes:

- Une déclaration de **goto** pour sortir de toute la structure en boucle.
- Une variable d'indicateur spécifique (`shouldBreak` dans l'exemple suivant) qui peut être vérifiée à la fin de chaque itération de la boucle externe.
- Refactoring le code pour utiliser une instruction `return` dans le corps de la boucle la plus interne ou éviter toute la structure de la boucle imbriquée.

```
bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
            shouldBreak = true;
        }
    }

    if(shouldBreakNow)
    {
        break; // Break out of outer loop if flag was set to true
    }
}

endAllLooping: // label from where control flow will continue
```

abstrait

Une classe marquée avec le mot clé `abstract` ne peut pas être instanciée.

Une classe *doit* être marquée comme abstraite si elle contient des membres abstraits ou si elle hérite des membres abstraits qu'elle ne met pas en œuvre. Une classe *peut* être marquée comme abstraite même si aucun membre abstrait n'est impliqué.

Les classes abstraites sont généralement utilisées comme classes de base lorsqu'une partie de l'implémentation doit être spécifiée par un autre composant.

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}
```

```
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();       // Allowed due to Cat deriving from Animal
cat.MakeSound();              // will print out "Meov meov"

Animal dog = new Dog();       // Allowed due to Dog deriving from Animal
dog.MakeSound();              // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

Une méthode, une propriété ou un événement marqué avec le mot-clé `abstract` indique que l'implémentation de ce membre doit être fournie dans une sous-classe. Comme mentionné ci-dessus, les membres abstraits ne peuvent apparaître que dans les classes abstraites.

```
abstract class Animal
{
   public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

float, double, décimal

flotte

`float` est un alias du type de données .NET `System.Single` . Il permet de stocker des nombres à virgule flottante simple précision IEEE 754. Ce type de données est présent dans `mscorlib.dll` qui est implicitement référencé par chaque projet C # lorsque vous les créez.

Portée approximative: -3,4 × 10^{38} à 3,4 × 10^{38}

Précision décimale: 6-9 chiffres significatifs

Notation :

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> Il convient de noter que le type `float` entraîne souvent des erreurs d'arrondi significatives. Dans les applications où la précision est importante, d'autres types de données doivent être pris en compte.

---

double

`double` est un alias du type de données .NET `System.Double` . Il représente un nombre à virgule flottante de 64 bits à double précision. Ce type de données est présent dans `mscorlib.dll` qui est implicitement référencé dans tout projet C #.

Gamme: ± 5,0 × 10^{-324} à ± 1,7 × 10^{308}

Précision décimale: 15-16 chiffres significatifs

Notation :

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

---

décimal

`decimal` est un alias du type de données .NET `System.Decimal` . Il représente un mot-clé indiquant un type de données 128 bits. Par rapport aux types à virgule flottante, le type décimal a plus de précision et une plage plus petite, ce qui le rend approprié pour les calculs financiers et monétaires. Ce type de données est présent dans `mscorlib.dll` qui est implicitement référencé dans tout projet C #.

Gamme: -7,9 × 10^{28} à 7,9 × 10^{28}

Précision décimale: 28-29 chiffres significatifs

Notation :

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

uint

Un **entier non signé** , ou **uint** , est un type de données numérique qui ne peut contenir que des entiers positifs. Comme son nom l'indique, il représente un entier non signé de 32 bits. Le mot-clé **uint** lui-même est un alias pour le type de système de type commun `System.UInt32` . Ce type de données est présent dans `mscorlib.dll` , qui est implicitement référencé par chaque projet C # lorsque vous les créez. Il occupe quatre octets d'espace mémoire.

Les entiers non signés peuvent contenir n'importe quelle valeur de 0 à 4 294 967 295.

Exemples sur comment et maintenant ne pas déclarer les entiers non signés

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Remarque: selon Microsoft , il est recommandé d'utiliser le type de données **int** autant que possible, car le type de données **uint** n'est pas compatible avec CLS.

ce

Le mot `this` clé `this` fait référence à l'instance actuelle de la classe (objet). On peut ainsi distinguer deux variables portant le même nom, l'une au niveau de la classe (un champ) et l'autre un paramètre (ou une variable locale) d'une méthode.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Les autres utilisations du mot clé sont le chaînage des surcharges de constructeur non statiques :

```
public MyClass(int arg) : this(arg, null)
{
}
```

et écrire des indexeurs :

```
public string this[int idx1, string idx2]
{
    get { /* ... */ }
    set { /* ... */ }
}
```

et déclarer des méthodes d'extension :

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

S'il n'y a pas de conflit avec une variable locale ou d'un paramètre, il est une question de style que ce soit d'utiliser `this` ou non, si `this.MemberOfType` et `MemberOfType` seraient équivalentes dans ce cas. Voir aussi mot clé de `base` .

Notez que si une méthode d'extension doit être appelée sur l'instance actuelle, `this` est nécessaire. Par exemple, si vous êtes dans une méthode non statique d'une classe qui implémente `IEnumerable<>` et que vous souhaitez appeler l'extension `Count` avant, vous devez utiliser:

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

et `this` ne peut pas être omis ici.

pour

Syntaxe: `for (initializer; condition; iterator)`

- La boucle `for` est couramment utilisée lorsque le nombre d'itérations est connu.
- Les instructions de la section d' `initializer` ne sont exécutées qu'une seule fois avant d'entrer dans la boucle.
- La section `condition` contient une expression booléenne évaluée à la fin de chaque itération de boucle pour déterminer si la boucle doit quitter ou doit être réexécutée.
- La section `iterator` définit ce qui se passe après chaque itération du corps de la boucle.

Cet exemple montre comment `for` peut être utilisé pour itérer sur les caractères d'une chaîne:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Sortie:

H
e
l
l
o

Démo en direct sur .NET Fiddle

Toutes les expressions qui définissent une instruction `for` sont facultatives; Par exemple, l'instruction suivante est utilisée pour créer une boucle infinie:

```
for( ; ; )
{
    // Your code here
}
```

La section d' `initializer` peut contenir plusieurs variables, à condition qu'elles soient du même type. La section `condition` peut être composée de toute expression pouvant être évaluée à une `bool` . Et la section `iterator` peut effectuer plusieurs actions séparées par des virgules:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Sortie:

Bonjour
bonjour1
bonjour12

Démo en direct sur .NET Fiddle

tandis que

L'opérateur `while` itération sur un bloc de code jusqu'à ce que la requête conditionnelle soit égale à false ou que le code soit interrompu par une `goto` , `return` , `break` ou `throw` .

Syntaxe pour `while` mot clé:

while (*condition*) { *bloc de code;* }

Exemple:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Sortie:

"Tant qu'il est sur la boucle numéro 1."
"Tant qu'il est sur la boucle numéro 2."
"Tant qu'il est sur la boucle numéro 3."
"Tant qu'il est sur la boucle numéro 4."
"Tant qu'il est sur la boucle numéro 5."

Démo en direct sur .NET Fiddle

Une boucle while est **contrôlée par une entrée** , car la condition est vérifiée **avant** l'exécution du bloc de code joint. Cela signifie que la boucle while n'exécuterait pas ses instructions si la

condition est fausse.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Donner une condition `while` sans la configurer pour qu'elle devienne fausse à un moment donné entraînera une boucle infinie ou sans fin. Dans la mesure du possible, cela doit être évité, cependant, il peut y avoir des circonstances exceptionnelles lorsque vous en avez besoin.

Vous pouvez créer une telle boucle comme suit:

```
while (true)
{
//...
}
```

Notez que le compilateur C # transformera des boucles telles que

```
while (true)
{
// ...
}
```

ou

```
for(;;)
{
// ...
}
```

dans

```
{
:label
// ...
goto label;
}
```

Notez qu'une boucle while peut avoir n'importe quelle condition, aussi complexe soit-elle, à condition qu'elle évalue (ou retourne) une valeur booléenne (bool). Il peut également contenir une fonction qui renvoie une valeur booléenne (une telle fonction est évaluée par le même type qu'une expression telle que «a == x»). Par exemple,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

revenir

> MSDN: L'instruction return termine l'exécution de la méthode dans laquelle elle apparaît et renvoie le contrôle à la méthode appelante. Il peut également renvoyer une valeur facultative. Si la méthode est un type vide, l'instruction return peut être omise.

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}


public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

dans

Le mot clé `in` a trois utilisations:

a) Dans le cadre de la syntaxe d'une instruction `foreach` ou de la syntaxe d'une requête LINQ

```
foreach (var member in sequence)
{
    // ...
}
```

b) Dans le contexte des interfaces génériques et des types de délégué génériques, cela signifie la *contravariance* pour le paramètre de type en question:

```
public interface IComparer<in T>
{
    // ...
}
```

c) Dans le contexte de LINQ, la requête fait référence à la collection interrogée

```
var query = from x in source select new { x.Name, x.ID, };
```

en utilisant

Il existe deux types d' `using` de l' `using` mots clés, à savoir l' `using statement` et l' `using directive` :

1. **en utilisant la déclaration** :

   Le mot-clé `using` garantit que les objets qui implémentent l'interface `IDisposable` sont correctement éliminés après utilisation. Il existe un sujet distinct pour l' instruction using

2. **en utilisant la directive**

La directive `using` a trois utilisations, consultez la page msdn pour la directive using . Il existe un sujet distinct pour la directive d'utilisation .

scellé

Lorsqu'il est appliqué à une classe, le modificateur `sealed` empêche les autres classes d'en hériter.

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

Appliqué à une méthode `virtual` (ou à une propriété virtuelle), le modificateur `sealed` empêche cette méthode (propriété) d'être *remplacée* dans les classes dérivées.

```
public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}

public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}
```

taille de

Utilisé pour obtenir la taille en octets pour un type non géré

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2(Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

statique

Le modificateur `static` est utilisé pour déclarer un membre statique, qui n'a pas besoin d'être instancié pour être accédé, mais est simplement accessible via son nom, par exemple `DateTime.Now` .

`static` peut être utilisé avec des classes, des champs, des méthodes, des propriétés, des opérateurs, des événements et des constructeurs.

Bien qu'une instance d'une classe contienne une copie séparée de tous les champs d'instance de la classe, il n'y a qu'une seule copie de chaque champ statique.

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new A();
        A c = new A();

        Console.WriteLine(A.count); // 3
    }
}
```

`count` est égal au nombre total d'instances d' `A` classe `A`

Le modificateur static peut également être utilisé pour déclarer un constructeur statique pour une classe, pour initialiser des données statiques ou exécuter du code qui doit être appelé une seule fois. Les constructeurs statiques sont appelés avant que la classe ne soit référencée pour la première fois.

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

Une `static class` est marquée avec le mot-clé `static` et peut être utilisée comme conteneur utile pour un ensemble de méthodes qui fonctionnent sur des paramètres, mais ne nécessite pas nécessairement d'être lié à une instance. En raison de la nature `static` de la classe, elle ne peut pas être instanciée, mais peut contenir un `static constructor` . Certaines caractéristiques d'une `static class` incluent:

- Ne peut pas être hérité

- Ne peut pas hériter d'autre chose que `Object`
- Peut contenir un constructeur statique mais pas un constructeur d'instance
- Ne peut contenir que des membres statiques
- Est scellé

Le compilateur est également convivial et permettra au développeur de savoir si des membres de l'instance existent dans la classe. Un exemple serait une classe statique qui convertit les métriques américaines et canadiennes:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

Lorsque les classes sont déclarées statiques:

```
public static class Functions
{
  public static int Double(int value)
  {
    return value + value;
  }
}
```

toutes les fonctions, propriétés ou membres de la classe doivent également être déclarés statiques. Aucune instance de la classe ne peut être créée. Essentiellement, une classe statique vous permet de créer des groupes de fonctions regroupés de manière logique.

Étant donné que C # 6 `static` peut également être utilisé parallèlement à l' `using` pour importer des membres et des méthodes statiques. Ils peuvent être utilisés sans nom de classe.

Ancienne manière, sans `using static` :

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
Console
    }

}
```

Exemple d' `using static`

```
using static System.Console;

public class ConsoleApplication
```

```
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

Désavantages

Bien que les classes statiques puissent être incroyablement utiles, elles comportent leurs propres avertissements:

- Une fois que la classe statique a été appelée, la classe est chargée en mémoire et ne peut pas être exécutée via le ramasse-miettes tant que AppDomain ne contient pas la classe statique.
- Une classe statique ne peut pas implémenter une interface.

int

`int` est un alias pour `System.Int32` , qui est un type de données pour les entiers signés 32 bits. Ce type de données peut être trouvé dans `mscorlib.dll` qui est implicitement référencé par chaque projet C # lorsque vous les créez.

Gamme: -2 147 483 648 à 2 147 483 647

```
int int1 = -10007;
var int2 = 2132012521;
```

longue

Le mot-clé **long** est utilisé pour représenter les entiers 64 bits signés. C'est un alias pour le type de données `System.Int64` présent dans `mscorlib.dll` , qui est implicitement référencé par chaque projet C # lorsque vous les créez.

*Toute variable **longue** peut être déclarée explicitement et implicitement:*

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

Une variable **longue** peut contenir n'importe quelle valeur comprise entre -9,223,372,036,854,775,808 et 9,223,372,036,854,775,807, et peut être utile dans les situations où une variable doit contenir une valeur supérieure aux limites de ce que peuvent contenir les autres variables (telles que la variable **int**).

ulong

Mot-clé utilisé pour les entiers 64 bits non signés. Il représente le type de données `System.UInt64`

trouvé dans `mscorlib.dll` qui est implicitement référencé par chaque projet C # lorsque vous les créez.

Gamme: 0 à 18,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 1544674406360945131 5UL;
```

dynamique

Le mot-clé `dynamic` est utilisé avec des objets typés dynamiquement . Les objets déclarés comme `dynamic` renoncent aux vérifications statiques à la compilation et sont évalués à l'exécution.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

L'exemple suivant utilise la `dynamic` avec la bibliothèque Json.NET de Newtonsoft, afin de lire facilement les données d'un fichier JSON désérialisé.

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

Il existe certaines limitations associées au mot-clé dynamique. L'un d'eux est l'utilisation de méthodes d'extension. L'exemple suivant ajoute une méthode d'extension pour string: `SayHello` .

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

La première approche sera de l'appeler comme d'habitude (comme pour une chaîne):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

Aucune erreur de compilation, mais à l'exécution, vous obtenez une `RuntimeBinderException` . La solution consiste à appeler la méthode d'extension via la classe statique:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

virtuel, remplacement, nouveau

virtuel et remplacement

Le mot-clé `virtual` permet à une méthode, une propriété, un indexeur ou un événement d'être remplacés par des classes dérivées et présente un comportement polymorphe. (Les membres sont non virtuels par défaut en C #)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

Afin de remplacer un membre, le mot clé `override` est utilisé dans les classes dérivées. (Notez que la signature des membres doit être identique)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

Le comportement polymorphe des membres virtuels signifie que, lorsqu'il est appelé, le membre en cours d'exécution est déterminé lors de l'exécution et non lors de la compilation. Le membre dominant dans la classe la plus dérivée de l'objet particulier est une instance exécutée.

En bref, l'objet peut être déclaré de type `BaseClass` au moment de la compilation, mais si à l'exécution il s'agit d'une instance de `DerivedClass` le membre remplacé sera exécuté:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
```

```
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

La substitution d'une méthode est facultative:

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Nouveau

Étant donné que seuls les membres définis comme `virtual` sont remplaçables et polymorphes, une classe dérivée qui redéfinit un membre non virtuel peut entraîner des résultats inattendus.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

Lorsque cela se produit, le membre exécuté est toujours déterminé au moment de la compilation en fonction du type d'objet.

- Si l'objet est déclaré de type `BaseClass` (même si à l'exécution est d'une classe dérivée), alors la méthode de `BaseClass` est exécutée
- Si l'objet est déclaré de type `DerivedClass` alors la méthode de `DerivedClass` est exécutée.

Il s'agit généralement d'un accident (lorsqu'un membre est ajouté au type de base après qu'un membre identique ait été ajouté au type dérivé) et qu'un avertissement de compilation **CS0108** est généré dans ces scénarios.

Si c'était intentionnel, le `new` mot-clé est utilisé pour supprimer l'avertissement du compilateur (et informez les autres développeurs de vos intentions!). le comportement reste le même, le `new` mot-clé supprime simplement l'avertissement du compilateur.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

L'utilisation de la substitution n'est *pas* facultative

Contrairement au C ++, l'utilisation du mot clé `override` n'est *pas* facultative:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

L'exemple ci-dessus provoque également l'avertissement **CS0108** , car `B.Foo()` ne `A.Foo()` pas automatiquement `A.Foo()` . Ajouter une `override` lorsque l'intention est de remplacer la classe de base et de provoquer un comportement polymorphe, d'ajouter de `new` lorsque vous souhaitez un comportement non polymorphe et de résoudre l'appel en utilisant le type statique. Ce dernier doit être utilisé avec prudence, car il peut entraîner une grave confusion.

Le code suivant entraîne même une erreur:

```
public class A
{
    public void Foo()
    {
```

```
    }
}

public class B : A
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

Les classes dérivées peuvent introduire un polymorphisme

Le code suivant est parfaitement valide (bien que rare):

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
        }
    }

    public class B : A
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

Maintenant, tous les objets avec une référence statique de B (et ses dérivés) utilisent le polymorphisme pour résoudre `Foo()` , tandis que les références de A utilisent `A.Foo()` .

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

Les méthodes virtuelles ne peuvent pas être privées

Le compilateur C # est strict en empêchant les constructions insensées. Les méthodes marquées comme `virtual` ne peuvent pas être privées. Parce qu'une méthode privée ne peut pas être vue à partir d'un type dérivé, elle ne peut pas non plus être écrasée. Cela ne compile pas:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

async, attend

Le mot-clé `await` été ajouté dans la version C # 5.0 prise en charge à partir de Visual Studio 2012. Il s'appuie sur TPL (Task Parallel Library), ce qui facilite considérablement le multithreading. Les mots-clés `async` et en `await` sont utilisés par paire dans la même fonction, comme indiqué ci-dessous. Le mot-clé `await` est utilisé pour suspendre l'exécution de la méthode asynchrone actuelle jusqu'à ce que la tâche asynchrone attendue soit terminée et / ou que ses résultats soient renvoyés. Pour utiliser le mot-clé `await` , la méthode qui l'utilise doit être marquée avec le mot-clé `async` .

L'utilisation d' `async` avec `void` est fortement déconseillée. Pour plus d'informations, vous pouvez regarder ici .

Exemple:

```
public async Task DoSomethingAsync()
{
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

Sortie:

> "Lancer un processus inutile ..."
>
> ** ... 1 seconde de retard ... **
>
> "Il a fallu 1000 millisecondes pour exécuter un processus inutile."

Les paires de mots-clés `async` et `await` peuvent être omises si une méthode de retour de `Task` ou de `Task<T>` ne renvoie qu'une seule opération asynchrone.

Plutôt que ceci:

```
public async Task PrintAndDelayAsync(string message, int delay)
```

```
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

Il est préférable de le faire:

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

5.0

En C # 5.0 `await` ne peut pas être utilisé dans `catch` et `finally` .

6,0

Avec C # 6.0, l' `await` peut être utilisée dans `catch` et `finally` .

carboniser

Un caractère est une lettre unique stockée dans une variable. C'est un type de valeur intégré qui prend deux octets d'espace mémoire. Il représente le type de données `System.Char` trouvé dans `mscorlib.dll` qui est implicitement référencé par chaque projet C # lorsque vous les créez.

Il y a plusieurs façons de faire cela.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

Un caractère peut être implicitement converti en `ushort, int, uint, long, ulong, float, double,` ou `decimal` et renverra la valeur entière de ce caractère.

```
ushort u = c;
```

renvoie 99 etc.

Cependant, il n'y a pas de conversions implicites d'autres types en char. Au lieu de cela, vous devez les lancer.

```
ushort u = 99;
 char c = (char)u;
```

fermer à clé

`lock` fournit la sécurité des threads pour un bloc de code, de sorte qu'il ne soit accessible que par un seul thread dans le même processus. Exemple:

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:

Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

Cas d'utilisation:

Chaque fois que vous avez un bloc de code qui peut produire des effets secondaires s'il est exécuté par plusieurs threads en même temps. Le mot clé lock ainsi qu'un **objet de synchronisation partagé** (`_objLock` dans l'exemple) peuvent être utilisés pour empêcher cela.

Notez que `_objLock` ne peut pas être `null` et que plusieurs threads exécutant le code doivent utiliser la même instance d'objet (en en faisant un champ `static` ou en utilisant la même instance de classe pour les deux threads)

Du côté du compilateur, le mot clé lock est un sucre syntaxique remplacé par `Monitor.Enter(_lockObj);` et `Monitor.Exit(_lockObj);` . Donc, si vous remplacez le verrou en entourant le bloc de code avec ces deux méthodes, vous obtiendrez les mêmes résultats. Vous pouvez voir le code réel dans le sucre syntaxique dans C # - exemple de verrouillage

nul

Une variable d'un type référence peut contenir une référence valide à une instance ou une référence null. La référence null est la valeur par défaut des variables de type référence, ainsi que

les types de valeur nullable.

`null` est le mot clé qui représente une référence null.

En tant qu'expression, il peut être utilisé pour affecter la référence null aux variables des types susmentionnés:

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

Les types de valeur non nullables ne peuvent pas recevoir de référence nulle. Toutes les affectations suivantes sont invalides:

```
int a = null;
float b = null;
decimal c = null;
```

La référence null *ne* doit *pas* être confondue avec des instances valides de différents types, telles que:

- une liste vide (`new List<int>()`)
- une chaîne vide (`""`)
- le nombre zéro (`0` , `0f` , `0m`)
- le caractère nul (`'\0'`)

Parfois, il est utile de vérifier si quelque chose est nul ou un objet vide / par défaut. La méthode System.String.IsNullOrEmpty (String) peut être utilisée pour vérifier cela ou vous pouvez implémenter votre propre méthode équivalente.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null
reference. Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

interne

Le mot clé `internal` est un modificateur d'accès pour les types et les membres de type. Les types internes ou les membres sont **accessibles uniquement dans les fichiers du même assembly**

usage:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

La différence entre les différents modificateurs d'accès est clarifiée ici

Modificateurs d'accès

Publique

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 ne peut être accédé que par le code de la même classe ou de la même structure.

protégé

Le type ou le membre est uniquement accessible par code dans la même classe ou structure, ou dans une classe dérivée.

interne

Le type ou le membre est accessible par n'importe quel code du même assembly, mais pas d'un autre assembly.

protégé interne

Le type ou le membre est accessible par n'importe quel code du même assembly ou par toute classe dérivée d'un autre assembly.

Si **aucun modificateur d'accès** n'est défini, un modificateur d'accès par défaut est utilisé. Il y a donc toujours une forme de modificateur d'accès même si ce n'est pas le cas.

où

`where` peut servir à deux fins dans C #: taper la contrainte dans un argument générique et filtrer les requêtes LINQ.

Dans une classe générique, considérons

```
public class Cup<T>
{
    // ...
}
```

T s'appelle un paramètre de type. La définition de classe peut imposer des contraintes sur les

types réels pouvant être fournis pour T.

Les types de contraintes suivants peuvent être appliqués:

- type de valeur
- Type de référence
- constructeur par défaut
- héritage et implémentation

type de valeur

Dans ce cas, seules les `struct` (ceci inclut les types de données «primitifs» tels que `int` , `boolean` etc.) peuvent être fournies.

```
public class Cup<T> where T : struct
{
    // ...
}
```

Type de référence

Dans ce cas, seuls les types de classe peuvent être fournis

```
public class Cup<T> where T : class
{
    // ...
}
```

valeur hybride / type de référence

Il est parfois souhaitable de restreindre les arguments de type à ceux disponibles dans une base de données. Celles-ci sont généralement associées à des types de valeur et à des chaînes. Comme toutes les restrictions de type doivent être respectées, il n'est pas possible de spécifier `where T : struct or string` (ce n'est pas une syntaxe valide). Une solution consiste à restreindre les arguments de type à `IConvertible` qui a des types intégrés de "... Booléen, SByte, Octet, Int16, UInt16, Int32, UInt32, Int64, UInt64, Unique, Double, Décimal, DateHeure, Char et Chaîne. " Il est possible que d'autres objets implémentent IConvertible, bien que cela soit rare dans la pratique.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

constructeur par défaut

Seuls les types contenant un constructeur par défaut seront autorisés. Cela inclut les types de valeur et les classes qui contiennent un constructeur par défaut (sans paramètre)

```
public class Cup<T> where T : new
{
    // ...
```

```
}
```

héritage et implémentation

Seuls les types qui héritent d'une certaine classe de base ou implémentent une interface donnée peuvent être fournis.

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

La contrainte peut même référencer un autre paramètre de type:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Plusieurs contraintes peuvent être spécifiées pour un argument de type:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

Les exemples précédents montrent des contraintes génériques sur une définition de classe, mais les contraintes peuvent être utilisées partout où un argument de type est fourni: classes, structures, interfaces, méthodes, etc.

`where` peut aussi être une clause LINQ. Dans ce cas, il est analogue à `WHERE` en SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
    // prints 2 1 3 2 0
```

externe

Le mot clé `extern` est utilisé pour déclarer les méthodes implémentées en externe. Cela peut être utilisé conjointement avec l'attribut DllImport pour appeler du code non géré à l'aide des services Interop. qui dans ce cas viendra avec `static` modificateur `static`

Par exemple:

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);

    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

Cela utilise la méthode SetForegroundWindow importée de la bibliothèque User32.dll

Cela peut également être utilisé pour définir un alias d'assembly externe. ce qui nous permet de référencer différentes versions de mêmes composants à partir d'un seul assemblage.

Pour référencer deux assemblys avec les mêmes noms de type complets, un alias doit être spécifié 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 depuis un programme, référencez-les en utilisant le mot-clé extern. Par exemple:

```
extern alias GridV1;
extern alias GridV2;
```

bool

Mot-clé pour stocker les valeurs booléennes `true` et `false` . bool est un alias de System.Boolean.

La valeur par défaut d'un bool est false.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

Pour qu'un bool autorise des valeurs nulles, il doit être initialisé en tant que booléen?.

La valeur par défaut d'un bool? est nul

```
bool? a // default value is null
```

quand

Le `when` est un mot clé ajouté dans **C # 6** et il est utilisé pour le filtrage des exceptions.

Avant l'introduction du mot `when` clé `when` , vous auriez pu avoir une clause catch pour chaque type d'exception; avec l'ajout du mot-clé, un contrôle plus fin est désormais possible.

Une expression `when` est attachée à une branche `catch` et seulement si la condition `when` est `true` , la clause `catch` sera exécutée. Il est possible d'avoir plusieurs clauses `catch` avec les mêmes types de classes d'exception, et différentes `when` conditions `when` remplies.

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }


CatchException(Method1);
CatchException(Method2);
```

décoché

Le mot clé `unchecked` empêche le compilateur de rechercher les débordements / sous-flux.

Par exemple:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Sans le mot-clé `unchecked` , aucune des deux opérations d'addition ne sera compilée.

Quand est-ce utile?

Ceci est utile car cela peut accélérer les calculs qui ne déborderont certainement pas, car la vérification du débordement prend du temps ou lorsqu'un comportement de débordement / débordement est souhaité (par exemple, lors de la génération d'un code de hachage).

vide

Le mot réservé `"void"` est un alias de type `System.Void` et a deux utilisations:

1. Déclarez une méthode qui n'a pas de valeur de retour:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

Une méthode avec un type de retour vide peut toujours avoir le mot-clé de `return` dans son corps. Ceci est utile lorsque vous souhaitez quitter l'exécution de la méthode et renvoyer le flux à l'appelant:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Déclarez un pointeur sur un type inconnu dans un contexte non sécurisé.

Dans un contexte non sécurisé, un type peut être un type de pointeur, un type de valeur ou un type de référence. Une déclaration de type pointeur est généralement `type* identifier` , où le type est un type connu - c.-à-d. `int* myInt` , mais peut également être un `void* identifier` , où le type est inconnu

Notez que la déclaration d'un type de pointeur vide est déconseillée par Microsoft.

si, si ... sinon, si ... sinon si

L'instruction `if` est utilisée pour contrôler le flux du programme. Une instruction `if` identifie l'instruction à exécuter en fonction de la valeur d'une expression `Boolean` .

Pour une seule instruction, les `braces` {} sont facultatives mais recommandées.

```
int a = 4;
if(a % 2 == 0)
```

```
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

Le `if` peut aussi avoir une clause `else` , qui sera exécutée si la condition est évaluée à false:

```
int a = 5;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

Le `if` ... `else if` construct vous permet de spécifier plusieurs conditions:

```
int a = 9;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
{
     Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

Important de noter **que si une condition est remplie dans l'exemple ci-dessus, le contrôle ignore d'autres tests et saute à la fin de cette construction si sinon.Par conséquent, l' *ordre* des tests est important si vous utilisez if .. else if construct**

Les expressions booléennes C # utilisent une évaluation de court-circuit . Ceci est important dans les cas où l'évaluation des conditions peut avoir des effets secondaires:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

Il n'y a aucune garantie que `someOtherBooleanMethodWithSideEffects` sera réellement exécuté.

C'est également important dans les cas où des conditions préalables garantissent qu'il est "sûr" d'évaluer les versions ultérieures. Par exemple:

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

La commande est très importante dans ce cas car, si nous inversons la commande:

```
if (someCollection.Count > 0 && someCollection != null) {
```

il va lancer une `NullReferenceException` si `someCollection` est `null` .

faire

L'opérateur do effectue une itération sur un bloc de code jusqu'à ce qu'une requête conditionnelle soit égale à false. La boucle do-while peut également être interrompue par une `goto` , `return` , `break` ou `throw` .

La syntaxe du mot clé `do` est la suivante:

faire { *bloc de code;* } while (*condition*);

Exemple:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Sortie:

"Do est sur la boucle numéro 1."
"Do est sur la boucle numéro 2."
"Do est sur la boucle numéro 3."
"Do est sur la boucle numéro 4."
"Do est sur la boucle numéro 5."

Contrairement à la `while` boucle, la boucle do-while est **sortie contrôlée.** Cela signifie que la boucle do-while exécute ses instructions au moins une fois, même si la condition échoue la première fois.

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

opérateur

La plupart des opérateurs intégrés (y compris les opérateurs de conversion) peuvent être surchargés en utilisant le mot clé d' `operator` avec les modificateurs `public` et `static` .

Les opérateurs se présentent sous trois formes: opérateurs unaires, opérateurs binaires et opérateurs de conversion.

Les opérateurs unaires et binaires nécessitent au moins un paramètre du même type que le type conteneur, et certains nécessitent un opérateur de correspondance complémentaire.

Les opérateurs de conversion doivent convertir vers ou à partir du type englobant.

```
public struct Vector32
{

    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{X}, {Y}}}";

}

public struct Vector64
{
```

```
    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}, {Y}}}";

}
```

Exemple

```
var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2);  // {102, 103}
Console.WriteLine(vector1 - vector2);  // {-72, -25}
```

struct

Un type de `struct` est un type de valeur généralement utilisé pour encapsuler de petits groupes de variables associées, telles que les coordonnées d'un rectangle ou les caractéristiques d'un élément dans un inventaire.

> Les classes sont des types de référence, les structs sont des types de valeur.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }

        public void Display(string name)
        {
            WriteLine(name + ": " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
```

```
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

Les structures peuvent également contenir des constructeurs, des constantes, des champs, des méthodes, des propriétés, des indexeurs, des opérateurs, des événements et des types imbriqués.

Quelques suggestions de MS sur l'utilisation de struct et Quand utiliser la classe:

CONSIDÉRER

définir une structure au lieu d'une classe si les instances de ce type sont petites et généralement de courte durée ou sont généralement incorporées dans d'autres objets.

ÉVITER

définir une structure à moins que le type présente toutes les caractéristiques suivantes:

- Il représente logiquement une valeur unique, similaire aux types primitifs (int, double, etc.)
- Il a une taille d'instance inférieure à 16 octets.
- C'est immuable.
- Il ne sera pas nécessaire de l'encadrer fréquemment.

commutateur

L'instruction `switch` est une instruction de contrôle qui sélectionne une section de commutateur à exécuter dans une liste de candidats. Une instruction de commutateur comprend une ou plusieurs sections de commutateur. Chaque section de commutateur contient une ou plusieurs étiquettes de `case` suivies d'une ou plusieurs instructions. Si aucune étiquette ne contient une valeur correspondante, le contrôle est transféré dans la section `default` , le cas échéant. La gestion des cas n'est pas prise en charge en C # à proprement parler. Cependant, si une ou plusieurs étiquettes de `case` sont vides, l'exécution suivra le code du prochain bloc de `case` contenant du code. Cela permet de regrouper plusieurs étiquettes de `case` avec la même implémentation. Dans l'exemple suivant, si `month` est égal à 12, le code dans le `case 2` sera exécuté car les étiquettes de `case 12 1` et `2` sont regroupées. Si un `case` bloc est pas vide, une `break` doit être présent avant le prochain `case` l' étiquette, sinon le compilateur signaleront une erreur.

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
```

```
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
    default:
        Console.WriteLine("Incorrect month index");
        break;
}
```

Un `case` ne peut être étiqueté que par une valeur connue à la *compilation* (par exemple `1` , `"str"` , `Enum.A`), donc une `variable` n'est pas une étiquette de `case` valide, mais une valeur `const` ou `Enum` est (ainsi que toute valeur littérale).

interface

Une `interface` contient les signatures des méthodes, propriétés et événements. Les classes dérivées définissent les membres car l'interface ne contient que la déclaration des membres.

Une interface est déclarée en utilisant le mot-clé `interface` .

```
interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;

    public Product(decimal price)
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}
```

peu sûr

Le mot clé `unsafe` peut être utilisé dans les déclarations de type ou de méthode ou pour déclarer un bloc en ligne.

Le but de ce mot clé est de permettre l'utilisation du *sous - ensemble non sécurisé* de C # pour le bloc en question. Le sous-ensemble non sécurisé inclut des fonctionnalités telles que les pointeurs, l'allocation de pile, les tableaux de type C, etc.

Le code dangereux n'est pas vérifiable et c'est pourquoi son utilisation est déconseillée. La compilation de code non sécurisé nécessite de passer un commutateur au compilateur C #. En outre, le CLR exige que l'assembly en cours d'exécution soit totalement sécurisé.

Malgré ces limitations, le code non sécurisé a des utilisations valides pour rendre certaines opérations plus performantes (par exemple, indexation de tableaux) ou plus simples (par exemple, interopérer avec certaines bibliothèques non gérées).

Comme exemple très simple

```
// compile with /unsafe
class UnsafeTest
{
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p; // the '*' dereferences the pointer.
      //Since we passed in "the address of i", this becomes "i *= i"
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
      Console.WriteLine(i); // Output: 25
   }
}
```

En travaillant avec des pointeurs, nous pouvons changer les valeurs des emplacements de mémoire directement, plutôt que de devoir les adresser par leur nom. Notez que cela nécessite souvent l'utilisation du mot clé fixed pour éviter toute corruption de mémoire car le ramasse-miettes déplace les objets (sinon, vous risquez d'obtenir l' erreur CS0212). Comme une variable qui a été "corrigée" ne peut pas être écrite, nous devons souvent avoir un deuxième pointeur qui pointe vers le même emplacement que le premier.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;
```

```
    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++;      //move the pointer to the next memory space.
                      // NOTE that the pointer will move 4 bytes since "p" is an
                      // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;"
        }
    }
}
```

Sortie:

```
1
4
9
16
25
36
49
64
81
100
```

`unsafe` permet également l'utilisation de stackalloc qui allouera de la mémoire sur la pile comme _alloca dans la bibliothèque d'exécution C. Nous pouvons modifier l'exemple ci-dessus pour utiliser `stackalloc` comme suit:

```
unsafe void Main()
{
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
```

```
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

(La sortie est la même que ci-dessus)

implicite

Le mot clé `implicit` est utilisé pour surcharger un opérateur de conversion. Par exemple, vous pouvez déclarer une classe `Fraction` qui doit automatiquement être convertie en `double` lorsque cela est nécessaire et qui peut être automatiquement convertie à partir de l' `int` :

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

vrai faux

Les mots-clés `true` et `false` ont deux utilisations:

1. En tant que valeurs booléennes littérales

```
var myTrueBool = true;
var myFalseBool = false;
```

2. En tant qu'opérateurs pouvant être surchargés

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}
```

```
public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

La surcharge de l'opérateur false était utile avant C # 2.0, avant l'introduction des types `Nullable` . Un type qui surcharge l'opérateur `true` doit également surcharger l'opérateur `false` .

chaîne

`string` est un alias du type de données .NET, `System.String` , qui permet de stocker du texte (séquences de caractères).

Notation:

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Chaque caractère de la chaîne est codé en UTF-16, ce qui signifie que chaque caractère nécessite au moins 2 octets d'espace de stockage.

ushort

Type numérique utilisé pour stocker des entiers positifs à 16 bits. `ushort` est un alias de `System.UInt16` et occupe 2 octets de mémoire.

La plage valide est comprise entre `0` et `65535` .

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

sbyte

Type numérique utilisé pour stocker des entiers *signés à* 8 bits. `sbyte` est un alias pour `System.SByte` et occupe 1 octet de mémoire. Pour l'équivalent non signé, utilisez l' `byte` .

La plage valide est comprise entre `-127` et `127` (le reste est utilisé pour stocker le signe).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

var

Une variable locale implicitement typée qui est fortement typée comme si l'utilisateur avait déclaré

le type. Contrairement aux autres déclarations de variables, le compilateur détermine le type de variable qu'il représente en fonction de la valeur qui lui est attribuée.

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

Contrairement aux autres types de variables, les définitions de variable avec ce mot-clé doivent être initialisées lors de la déclaration. Cela est dû au mot-clé **var** représentant une variable implicitement typée.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

Le mot-clé **var** peut également être utilisé pour créer de nouveaux types de données à la volée. Ces nouveaux types de données sont connus sous le nom de *types anonymes* . Ils sont très utiles, car ils permettent à un utilisateur de définir un ensemble de propriétés sans avoir à déclarer explicitement un type d'objet en premier.

Type anonyme simple

```
var a = new { number = 1, text = "hi" };
```

Requête LINQ qui renvoie un type anonyme

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName  { get; set; }
}

public void GetDogsWithBreedNames()
{
    var db = new DogDataContext(ConnectString);
    var result = from d in db.Dogs
              join b in db.Breeds on d.BreedId equals b.BreedId
              select new
                     {
                         DogName = d.Name,
                         BreedName = b.BreedName
                     };

    DoStuff(result);
}
```

Vous pouvez utiliser le mot-clé var dans l'instruction foreach

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

déléguer

Les délégués sont des types qui représentent une référence à une méthode. Ils sont utilisés pour transmettre des méthodes comme arguments à d'autres méthodes.

Les délégués peuvent contenir des méthodes statiques, des méthodes d'instance, des méthodes anonymes ou des expressions lambda.

```
class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}
```

Lors de l'attribution d'une méthode à un délégué, il est important de noter que la méthode doit avoir le même type de retour et les mêmes paramètres. Cela diffère de la surcharge de méthode

«normale», où seuls les paramètres définissent la signature de la méthode.

Les événements sont construits au-dessus des délégués.

un événement

Un `event` permet au développeur d'implémenter un modèle de notification.

Exemple simple

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}

public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

Référence MSDN

partiel

Le mot-clé `partial` peut être utilisé lors de la définition de type de class, struct ou interface pour permettre de diviser la définition de type en plusieurs fichiers. Ceci est utile pour incorporer de nouvelles fonctionnalités dans le code généré automatiquement.

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
```

```
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

Remarque: une classe peut être divisée en un nombre quelconque de fichiers. Cependant, toute déclaration doit être sous le même espace de noms et le même assembly.

Les méthodes peuvent également être déclarées partielles en utilisant le mot-clé `partial` . Dans ce cas, un fichier contiendra uniquement la définition de la méthode et un autre fichier contiendra l'implémentation.

> Une méthode partielle a sa signature définie dans une partie d'un type partiel et son implémentation est définie dans une autre partie du type. Les méthodes partielles permettent aux concepteurs de classes de fournir des hooks de méthodes, similaires aux gestionnaires d'événements, que les développeurs peuvent décider d'implémenter ou non. Si le développeur ne fournit pas d'implémentation, le compilateur supprime la signature au moment de la compilation. Les conditions suivantes s'appliquent aux méthodes partielles:
>
> - Les signatures dans les deux parties du type partiel doivent correspondre.
> - La méthode doit retourner vide.
> - Aucun modificateur d'accès n'est autorisé. Les méthodes partielles sont implicitement privées.
>
> - MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
```

```
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

Remarque: Le type contenant la méthode partielle doit également être déclaré partiel.

Lire Mots clés en ligne: https://riptutorial.com/fr/csharp/topic/26/mots-cles

Chapitre 117: nom de l'opérateur

Introduction

L'opérateur `nameof` vous permet d'obtenir le nom d'une **variable** , d'un **type** ou d'un **membre** sous forme de chaîne sans le coder comme un littéral.

L'opération est évaluée au moment de la compilation, ce qui signifie que vous pouvez renommer un identifiant référencé à l'aide de la fonction de renommage d'un IDE et que la chaîne de nom sera mise à jour avec cet identifiant.

Syntaxe

- nameof (expression)

Examples

Utilisation de base: Impression d'un nom de variable

L'opérateur `nameof` vous permet d'obtenir le nom d'une variable, d'un type ou d'un membre sous forme de chaîne sans le coder comme un littéral. L'opération est évaluée au moment de la compilation, ce qui signifie que vous pouvez renommer, en utilisant la fonctionnalité de renommage de l'EDI, un identifiant référencé et que la chaîne de nom sera mise à jour avec lui.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Serait sortie

    myString

car le nom de la variable est "myString". Refactoriser le nom de la variable changerait la chaîne.

Si elle est appelée sur un type de référence, l'opérateur `nameof` renvoie le nom de la référence en cours, *pas* le nom ou le nom du type de l'objet sous-jacent. Par exemple:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Impression d'un nom de paramètre

Fragment

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Sortie de console

paramValue

Événement Raising PropertyChanged

Fragment

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

Sortie de console

Adresse

Gestion des événements PropertyChanged

Fragment

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Sortie de console

Titre changé en Tout est en feu et cassé

Statut changé en ShowStopper

Appliqué à un paramètre de type générique

Fragment

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}

...

var myClass = new SomeClass<int>();
myClass.PrintTypeName();
```

```
Console.WriteLine(nameof(SomeClass<int>));
```

Sortie de console

TItem

SomeClass

Appliqué aux identificateurs qualifiés

Fragment

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Sortie de console

MyNamespace

Ma classe

MyNestedClass

MyStaticProperty

Vérification des arguments et clauses de garde

Préférer

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Plus de

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

L'utilisation de la fonction `nameof` facilite la refactorisation des paramètres de méthode.

Liens d'action MVC fortement typés

Au lieu de l'habituellement tapé librement:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

Vous pouvez maintenant créer des liens d'action fortement typés:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Maintenant, si vous souhaitez modifier votre code et renommer la méthode `UserController.LogIn` en `UserController.SignIn` , vous n'avez pas à vous soucier de rechercher toutes les occurrences de chaînes. Le compilateur fera le travail.

Lire nom de l'opérateur en ligne: https://riptutorial.com/fr/csharp/topic/80/nom-de-l-operateur

Chapitre 118: NullReferenceException

Examples

NullReferenceException expliqué

Une `NullReferenceException` est `NullReferenceException` lorsque vous essayez d'accéder à un membre non statique (propriété, méthode, champ ou événement) d'un objet de référence, mais qu'il est nul.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

Pour déboguer une telle exception, c'est assez simple: sur la ligne où l'exception est lancée, il suffit de regarder avant chaque exception `.` 'ou' `[` ', ou à de rares occasions' `(` '.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

D'où vient mon exception? Non plus:

- `myGarage` est `null`
- `myGarage.CarCollection` est `null`
- `currentIndex` est `null`
- `myGarage.CarCollection[currentIndex.Value]` est `null`
- `theCarInTheStreet` est `null`

En mode débogage, il vous suffit de placer le curseur de la souris sur chacun de ces éléments et vous trouverez votre référence null. Ensuite, vous devez comprendre pourquoi il n'a pas de valeur. La correction dépend totalement du but de votre méthode.

Avez-vous oublié d'instancier / initialiser?

```
myGarage.CarCollection = new Car[10];
```

Êtes-vous censé faire quelque chose de différent si l'objet est nul?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

Ou peut-être que quelqu'un vous a donné un argument nul et n'était pas censé:

```
if (theCarInTheStreet == null)
```

```
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

Dans tous les cas, rappelez-vous qu'une méthode ne doit jamais lancer une exception NullReferenceException. Si c'est le cas, cela signifie que vous avez oublié de vérifier quelque chose.

Lire NullReferenceException en ligne:
https://riptutorial.com/fr/csharp/topic/2702/nullreferenceexception

Chapitre 119: O (n) Algorithme de rotation circulaire d'un tableau

Introduction

Dans mon cheminement vers l'étude de la programmation, il y a eu des problèmes simples mais intéressants à résoudre en tant qu'exercices. L'un de ces problèmes était de faire pivoter un tableau (ou une autre collection) d'une certaine valeur. Ici, je partagerai avec vous une formule simple pour le faire.

Examples

Exemple de méthode générique qui fait pivoter un tableau par un décalage donné

Je voudrais souligner que nous tournons à gauche lorsque la valeur de décalage est négative et que nous tournons à droite lorsque la valeur est positive.

```
public static void Main()
{
    int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int shiftCount = 1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = 15;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

    array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = -1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

    array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = -35;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
}

private static void Rotate<T>(ref T[] array, int shiftCount)
{
    T[] backupArray= new T[array.Length];

    for (int index = 0; index < array.Length; index++)
    {
```

```
            backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
        }

        array = backupArray;
    }
```

La chose qui est importante dans ce code est la formule avec laquelle nous trouvons la nouvelle valeur d'index après la rotation.

(index + array.Length + shiftCount% array.Length)% array.Length

Voici un peu plus d'informations à ce sujet:

(shiftCount% array.Length) -> nous normalisons la valeur de décalage pour qu'elle soit dans la longueur du tableau (puisque dans un tableau de longueur 10, le décalage de 1 ou 11 est la même chose, la même chose vaut pour -1 et -11) .

array.Length + (shiftCount% array.Length) -> ceci est dû aux rotations de gauche pour s'assurer que nous **n'entrons** pas dans un index négatif, mais que nous le **tournons** à la fin du tableau. Sans cela, pour un tableau de longueur 10 pour l'index 0 et une rotation -1, nous entrerions dans un nombre négatif (-1) et n'obtiendrions pas la valeur réelle de l'indice de rotation, qui est 9. (10 + (-1% 10) =) 9)

index + array.Length + (shiftCount% array.Length) - pas grand chose à dire car nous appliquons la rotation à l'index pour obtenir le nouvel index. (0 + 10 + (-1% 10) = 9)

index + array.Length + (shiftCount% array.Length)% array.Length -> la deuxième normalisation **vérifie** que la nouvelle valeur d'index ne **sort** pas du tableau, mais fait pivoter la valeur au début du tableau. C'est pour les rotations droites, puisque dans un tableau de longueur 10 sans index 9 et de rotation 1 on irait dans index 10, qui est en dehors du tableau, et que la valeur réelle de l'indice de rotation est 0. ((9 + 10 + (1% 10))% 10 = 0)

Lire O (n) Algorithme de rotation circulaire d'un tableau en ligne:
https://riptutorial.com/fr/csharp/topic/9770/o--n--algorithme-de-rotation-circulaire-d-un-tableau

Chapitre 120: ObservableCollection

Examples

Initialiser ObservableCollection

`ObservableCollection` est une collection de type `T` like `List<T>` qui signifie qu'il contient des objets de type `T`

De la documentation, nous lisons que:

> `ObservableCollection` représente une collection de données dynamique qui fournit des notifications lorsque des éléments sont ajoutés, supprimés ou lorsque la liste entière est actualisée.

La principale différence avec les autres collections est que `ObservableCollection` implémente les interfaces `INotifyCollectionChanged` et `INotifyCollectionChanged` et `INotifyPropertyChanged` immédiatement un événement de notification lorsqu'un nouvel objet est ajouté ou supprimé et lorsque la collecte est effacée.

Cela est particulièrement utile pour connecter l'interface utilisateur et le backend d'une application sans avoir à écrire de code supplémentaire car, lorsqu'un objet est ajouté ou supprimé d'une collection observable, l'interface utilisateur est automatiquement mise à jour.

La première étape pour l'utiliser est d'inclure

```
using System.Collections.ObjectModel
```

Vous pouvez créer une instance vide d'une collection, par exemple de type `string`

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

ou une instance qui est remplie de données

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

N'oubliez pas que dans toute collection IList, l'index commence à 0 (propriété IList.Item).

Lire ObservableCollection en ligne:
https://riptutorial.com/fr/csharp/topic/7351/observablecollection--t-

Chapitre 121: Opérateur d'égalité

Examples

Types d'égalité dans l'opérateur c # et d'égalité

En C #, il existe deux types d'égalité: l'égalité de référence et l'égalité de valeur. L'égalité des valeurs est la signification généralement comprise de l'égalité: cela signifie que deux objets contiennent les mêmes valeurs. Par exemple, deux entiers de la valeur 2 ont une valeur d'égalité. L'égalité de référence signifie qu'il n'y a pas deux objets à comparer. Au lieu de cela, il existe deux références d'objet, toutes deux faisant référence au même objet.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

Pour les types de valeur prédéfinis, l'opérateur d'égalité (==) renvoie true si les valeurs de ses opérandes sont égales, false sinon. Pour les types de référence autres que string, == renvoie true si ses deux opérandes font référence au même objet. Pour le type de chaîne, == compare les valeurs des chaînes.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

Lire Opérateur d'égalité en ligne: https://riptutorial.com/fr/csharp/topic/1491/operateur-d-egalite

Chapitre 122: Opérateur de coalescence nulle

Syntaxe

- var result = possibleNullObject ?? valeur par défaut;

Paramètres

Paramètre	Détails
`possibleNullObject`	La valeur à tester pour une valeur nulle. Si non nul, cette valeur est renvoyée. Doit être un type nullable.
`defaultValue`	La valeur renvoyée si `possibleNullObject` est null. Doit être le même type que `possibleNullObject` .

Remarques

L'opérateur de coalescence null lui-même est composé de deux points d'interrogation consécutifs: `??`

C'est un raccourci pour l'expression conditionnelle:

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

L'opérande de gauche (objet testé) doit être un type de valeur ou un type de référence nullable, ou une erreur de compilation se produira.

Le ?? L'opérateur fonctionne à la fois pour les types de référence et les types de valeur.

Examples

Utilisation de base

L'utilisation de l' `null-coalescing operator (??)` vous permet de spécifier une valeur par défaut pour un type nullable si l'opérande de gauche est `null` .

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Démo en direct sur .NET Fiddle

Ceci est logiquement équivalent à:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

ou en utilisant l' opérateur ternary (? :) :

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Null fall-through et chaining

L'opérande de gauche doit être nul, tandis que l'opérande de droite peut l'être ou non. Le résultat sera saisi en conséquence.

Non nullable

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Sortie:

Type: System.Int32
valeur: 3

Voir la démo

Nullable

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output` sera de type `int?` et égal à `b` ou `null` .

Coalescence Multiple

La coalescence peut également être effectuée en chaînes:

```
int? a = null;
int? b = null;
```

```
int c = 3;
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

Sortie:

Type: System.Int32
valeur: 3

Voir la démo

Chaînage conditionnel nul

L'opérateur de coalescence nulle peut être utilisé en parallèle avec l' opérateur de propagation nul pour fournir un accès plus sûr aux propriétés des objets.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

Sortie:

Type: System.String
valeur: valeur par défaut

Voir la démo

Opérateur de coalescence null avec appels de méthode

L'opérateur de coalescence null permet de s'assurer qu'une méthode pouvant renvoyer une valeur `null` revient à une valeur par défaut.

Sans l'opérateur de coalescence nulle:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

Avec l'opérateur de coalescence null:

```
string name = GetName() ?? "Unknown!";
```

Utiliser existant ou créer de nouvelles

Un scénario d'utilisation courant avec lequel cette fonctionnalité aide vraiment est lorsque vous recherchez un objet dans une collection et que vous devez en créer un s'il n'existe pas déjà.

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Initialisation des propriétés paresseuses avec un opérateur de coalescence nul

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

La première fois que la propriété `.FooBars` est accessible à la `_fooBars` variable d' évaluer comme `null` , tombant ainsi à travers à la déclaration d' affectation attribue et évalue à la valeur résultante.

Fil de sécurité

Ce n'est **pas un** moyen **sûr** d'utiliser des propriétés paresseuses. Pour la paresse thread-safe, utilisez la classe `Lazy<T>` intégrée au .NET Framework.

C # 6 Sugar Syntactic utilisant des corps d'expression

Notez que depuis C # 6, cette syntaxe peut être simplifiée en utilisant le corps de l'expression pour la propriété:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Les accès ultérieurs à la propriété donneront la valeur stockée dans la variable `_fooBars` .

Exemple dans le pattern MVVM

Ceci est souvent utilisé lors de l'implémentation de commandes dans le modèle MVVM. Au lieu d'initialiser les commandes avec empressement avec la construction d'un modèle de vue, les commandes sont initialisées paresseusement en utilisant ce modèle comme suit:

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Lire Opérateur de coalescence nulle en ligne: https://riptutorial.com/fr/csharp/topic/37/operateur-

de-coalescence-nulle

Chapitre 123: Opérateurs Null-Conditionnels

Syntaxe

- X? .Y; // null si X est nul sinon XY
- X? .Y? .Z; // null si X est nul ou Y est nul sinon XYZ
- X? [Index]; // null si X est nul sinon X [index]
- X? .ValueMethod (); // null si X est nul sinon le résultat de X.ValueMethod ();
- X? .VoidMethod (); // ne fait rien si X est nul sinon appelle X.VoidMethod ();

Remarques

Notez que lorsque vous utilisez l'opérateur de coalescence null sur un type de valeur `T` vous obtiendrez un `Nullable<T>` back.

Examples

Opérateur Null-Conditionnel

Le `?.` l'opérateur est le sucre syntaxique pour éviter les vérifications nulles verbeuses. Il est également connu sous le nom de l' opérateur de navigation sécurisé .

Classe utilisée dans l'exemple suivant:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

Si un objet est potentiellement nul (comme une fonction renvoyant un type de référence), l'objet doit d'abord être vérifié pour la valeur null afin d'éviter une éventuelle `NullReferenceException` . Sans l'opérateur null-conditionnel, cela ressemblerait à ceci:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

Le même exemple en utilisant l'opérateur null-conditionnel:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

Enchaînement de l'opérateur

L'opérateur conditionnel null peut être combiné sur les membres et sous-membres d'un objet.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

Combinaison avec l'opérateur de coalescence nulle

L'opérateur null-conditionnel peut être combiné avec l' opérateur null-coalescing pour fournir une valeur par défaut:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

L'indice Null-Conditionnel

De même que le `?.` opérateur, l'opérateur d'index null-conditionnel vérifie les valeurs NULL lors de l'indexation dans une collection pouvant être nulle.

```
string item = collection?[index];
```

est le sucre syntaxique pour

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

Éviter les NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

Cet effet peut être enchaîné:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
```

```
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

L'opérateur Null-conditionnel peut être utilisé avec la méthode d'extension

La méthode d'extension peut fonctionner sur des références nulles , mais vous pouvez utiliser `?.` pour annuler la vérification de toute façon.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normalement, la méthode sera déclenchée pour `null` références `null` et retournera -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

En utilisant `?.` la méthode ne sera pas déclenchée pour `null` références `null` , et le type est `int?` :

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

Ce comportement est en fait attendu de la façon dont le `?.` L'opérateur fonctionne: il évitera de faire des appels de méthode d'instance pour des instances NULL, afin d'éviter les `NullReferenceExceptions` . Cependant, la même logique s'applique à la méthode d'extension, malgré la différence sur la manière dont la méthode est déclarée.

Pour plus d'informations sur la raison pour laquelle la méthode d'extension est appelée dans le premier exemple, reportez-vous à la documentation relative aux méthodes d'extension - null .

Lire Opérateurs Null-Conditionnels en ligne: https://riptutorial.com/fr/csharp/topic/41/operateurs-null-conditionnels

Chapitre 124: Opérations sur les chaînes communes

Examples

Fractionnement d'une chaîne par caractère spécifique

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

Obtenir des sous-chaînes d'une chaîne donnée

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

`Substring` -chaîne renvoie la chaîne à partir d'un index donné ou entre deux index (tous deux inclus).

Déterminer si une chaîne commence par une séquence donnée

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Trouver une chaîne dans une chaîne

En utilisant `System.String.Contains` vous pouvez savoir si une chaîne particulière existe dans une chaîne. La méthode retourne un booléen, true si la chaîne existe, sinon false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Découpage des caractères indésirables au début et / ou à la fin des chaînes.

`String.Trim()`

```
string x = "   Hello World!    ";
string y = x.Trim(); // "Hello World!"
```

```
string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

`String.TrimStart()` et `String.TrimEnd()`

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

Formater une chaîne

Utilisez la méthode `String.Format()` pour remplacer un ou plusieurs éléments de la chaîne par la représentation sous forme de chaîne d'un objet spécifié:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Joindre un tableau de chaînes dans une nouvelle

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Remplissage d'une chaîne à une longueur fixe

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);        // paddedLeft = "  Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo+++"
string noPadded = s.PadLeft(2);          // noPadded = "Foo" (original string is never
shortened)
```

Construire une chaîne à partir de tableau

La méthode `String.Join` nous aidera à construire une chaîne De tableau / liste de caractères ou de chaîne. Cette méthode accepte deux paramètres. Le premier est le délimiteur ou le séparateur qui vous aidera à séparer chaque élément du tableau. Et le second paramètre est le tableau lui-même.

Chaîne du `char array` :

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Sortie : `a,b,c` si on change le `delimiter` comme `""` alors la sortie deviendra `abc` .

Chaîne de la `List of char` :

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Sortie : `a|b|c`

Chaîne de la `List of Strings` :

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

Sortie : `Ram is a boy`

Chaîne de `array of strings` :

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

Sortie : `Ram_is_a_boy`

Formatage avec ToString

Habituellement, nous `String.Format` méthode `String.Format` pour le formatage, le `.ToString` est généralement utilisé pour convertir d'autres types en chaîne. Nous pouvons spécifier le format avec la méthode ToString lorsque la conversion est en cours. Nous pouvons donc éviter un formatage supplémentaire. Laissez-moi expliquer comment cela fonctionne avec différents types;

Entier à la chaîne formatée:

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be   "Input value
is 10"
```

double à chaîne formatée:

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00");    // output 10
string customFormat = doubleValue.ToString("Input value is 0.0");  // Input value is 10.5
```

Mise en forme DateTime à l'aide de ToString

```
DateTime currentDate = DateTime.Now; //  {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

Obtenir x caractères du côté droit d'une chaîne

Visual Basic dispose de fonctions gauche, droite et moyenne qui renvoient des caractères à gauche, à droite et au milieu d'une chaîne. Ces méthodes n'existent pas en C #, mais peuvent être implémentées avec `Substring()` . Ils peuvent être implémentés en tant que méthodes d'extension telles que les suivantes:

```
public static class StringExtensions
{
   /// <summary>
   /// VB Left function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Left-most numchars characters</returns>
   public static string Left( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( 0, numchars );
   }

   /// <summary>
   /// VB Right function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Right-most numchars characters</returns>
   public static string Right( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( stringparam.Length - numchars );
   }

   /// <summary>
   /// VB Mid function - to end of string
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
   /// <returns>Balance of string beginning at startindex character</returns>
   public static string Mid( this string stringparam, int startindex )
   {
```

```
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
           startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1 );
     }

     /// <summary>
     /// VB Mid function - for number of characters
     /// </summary>
     /// <param name="stringparam"></param>
     /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
     /// <param name="numchars">number of characters to return</param>
     /// <returns>Balance of string beginning at startindex character</returns>
     public static string Mid( this string stringparam, int startindex, int numchars)
     {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
           startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1, numchars );

      }
   }
```

Cette méthode d'extension peut être utilisée comme suit:

```
string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);  // "World!"
string myLeftString = myLongString.Left(5);    // "Hello"
string myMidString1 = myLongString.Left(4);    // "lo World"
string myMidString2 = myLongString.Left(2,3);    // "ell"
```

Vérification de la chaîne vide à l'aide de String.IsNullOrEmpty () et String.IsNullOrWhiteSpace ()

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
```

```
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;

result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

Obtenir un caractère à un index spécifique et énumérer la chaîne

Vous pouvez utiliser la méthode `Substring` pour obtenir n'importe quel nombre de caractères d'une chaîne à un emplacement donné. Cependant, si vous ne voulez qu'un seul caractère, vous pouvez utiliser l'indexeur de chaînes pour obtenir un seul caractère à n'importe quel index, comme vous le faites avec un tableau:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Notez que le type de retour est `char` , contrairement à la méthode `Substring` qui renvoie un type de `string` .

Vous pouvez également utiliser l'indexeur pour parcourir les caractères de la chaîne:

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
/********* This will print each character on a new line:
h
e
l
l
o
**********/
```

Convertir un nombre décimal en format binaire, octal et hexadécimal

1. Pour convertir le nombre décimal en format binaire, utilisez la **base 2**

   ```
   Int32 Number = 15;
   Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
   ```

2. Pour convertir le nombre décimal en format octal, utilisez la **base 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
```

3. Pour convertir un nombre décimal au format hexadécimal, utilisez la **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

Fractionnement d'une chaîne par une autre chaîne

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Résultat:

["this", "is", "a", "complete", "phrase"]

Renverser correctement une chaîne

La plupart du temps, lorsque les gens doivent inverser une chaîne, ils le font plus ou moins comme ceci:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

Cependant, ce que ces gens ne réalisent pas, c'est que c'est vraiment faux.
Et je ne veux pas dire à cause de la vérification NULL manquante.

C'est en fait faux car un Glyph / GraphemeCluster peut être composé de plusieurs points de code (alias caractères).

Pour voir pourquoi, il faut d'abord être conscient du fait que le terme "personnage" signifie réellement.

Référence:

> Le caractère est un terme surchargé qui peut vouloir dire beaucoup de choses.
>
> Un point de code est l'unité atomique de l'information. Le texte est une séquence de points de code. Chaque point de code est un nombre qui a un sens par le standard Unicode.
>
> Un graphème est une séquence d'un ou plusieurs points de code affichés sous la forme d'une unité graphique unique qu'un lecteur reconnaît comme un élément unique du système d'écriture. Par exemple, a et ä sont tous deux des graphèmes, mais ils peuvent être composés de plusieurs points de code (par exemple, ä deux codes, un pour le caractère de base, puis un autre pour un code unique). point représentant ce grapheme). Certains points de code ne font jamais partie d'un graphème (par exemple,

les non-jointures de largeur nulle ou les remplacements directionnels).

Un glyphe est une image, généralement stockée dans une police (qui est une collection de glyphes), utilisée pour représenter des graphèmes ou des parties de ceux-ci. Les polices peuvent composer plusieurs glyphes en une seule représentation, par exemple, si le point ä ci-dessus est un point de code unique, une police peut choisir de le rendre sous la forme de deux glyphes séparés, superposés dans l'espace. Pour OTF, les tables GSUB et GPOS de la police contiennent des informations de substitution et de positionnement pour que cela fonctionne. Une police peut également contenir plusieurs glyphes alternatifs pour le même graphème.

Donc, en C #, un caractère est en fait un CodePoint.

Ce qui signifie que si vous inversez simplement une chaîne valide comme `Les Misé rables` , qui peut ressembler à ceci

```
string s = "Les Mise\u0301rables";
```

en tant que séquence de caractères, vous obtiendrez:

selbaàesiM seL

Comme vous pouvez le voir, l'accent est mis sur le caractère R au lieu du caractère e.
Bien que string.reverse.reverse fournisse la chaîne d'origine si vous inversez les deux fois le tableau de caractères, ce type d'inversion n'est certainement PAS l'inverse de la chaîne d'origine.

Vous devez inverser chaque graphe GraphemeCluster uniquement.
Donc, si c'est fait correctement, vous inversez une chaîne comme ceci:

```
    private static System.Collections.Generic.List<string> GraphemeClusters(string s)
    {
        System.Collections.Generic.List<string> ls = new
System.Collections.Generic.List<string>();

        System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
        while (enumerator.MoveNext())
        {
            ls.Add((string)enumerator.Current);
        }

        return ls;
    }


    // this
    private static string ReverseGraphemeClusters(string s)
    {
        if(string.IsNullOrEmpty(s) || s.Length == 1)
             return s;

        System.Collections.Generic.List<string> ls = GraphemeClusters(s);
        ls.Reverse();
```

```
        return string.Join("", ls.ToArray());
    }

    public static void TestMe()
    {
        string s = "Les Mise\u0301rables";
        // s = "noël";
        string r = ReverseGraphemeClusters(s);

        // This would be wrong:
        // char[] a = s.ToCharArray();
        // System.Array.Reverse(a);
        // string r = new string(a);

        System.Console.WriteLine(r);
    }
```

Et - oh joie - vous vous rendrez compte que si vous le faites correctement comme cela, cela fonctionnera également pour les langues asiatiques / asiatiques du sud / est-asiatiques (et français / suédois / norvégien, etc.) ...

Remplacement d'une chaîne dans une chaîne

À l'aide de la méthode `System.String.Replace` , vous pouvez remplacer une partie d'une chaîne par une autre chaîne.

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Toutes les occurrences de la chaîne de recherche sont remplacées.

Cette méthode peut également être utilisée pour supprimer une partie d'une chaîne à l'aide du champ `String.Empty` :

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Changer la casse des caractères dans une chaîne

La classe `System.String` prend en charge un certain nombre de méthodes pour convertir des caractères majuscules et minuscules dans une chaîne.

- `System.String.ToLowerInvariant` est utilisé pour renvoyer un objet String converti en minuscule.
- `System.String.ToUpperInvariant` est utilisé pour renvoyer un objet String converti en majuscule.

Remarque: La raison d'utiliser les versions *invariantes* de ces méthodes est d'empêcher la production de lettres inattendues spécifiques à la culture. Ceci est expliqué ici en détail .

Exemple:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Notez que vous *pouvez* choisir de spécifier une **culture** spécifique lors de la conversion en minuscules et en majuscules en utilisant les méthodes String.ToLower (CultureInfo) et String.ToUpper (CultureInfo) en conséquence.

Concaténer un tableau de chaînes en une seule chaîne

La méthode `System.String.Join` permet de concaténer tous les éléments d'un tableau de chaînes, en utilisant un séparateur spécifié entre chaque élément:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Concaténation de chaînes

La concaténation de chaînes peut être effectuée à l'aide de la méthode `System.String.Concat` ou (beaucoup plus facile) à l'aide de l'opérateur `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

En C # 6, cela peut être fait comme suit:

```
string concat = $"{first},{second}";
```

Lire Opérations sur les chaînes communes en ligne:
https://riptutorial.com/fr/csharp/topic/73/operations-sur-les-chaines-communes

Chapitre 125: Parallèle LINQ (PLINQ)

Syntaxe

- ParallelEnumerable.Aggregate (func)
- ParallelEnumerable.Aggregate (graine, func)
- ParallelEnumerable.Aggregate (seed, updateAccumulatorFunc, combineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.Aggregate (seedFactory, updateAccumulatorFunc, combineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.All (prédicat)
- ParallelEnumerable.Any ()
- ParallelEnumerable.Any (prédicat)
- ParallelEnumerable.AsEnumerable ()
- ParallelEnumerable.AsOrdered ()
- ParallelEnumerable.AsParallel ()
- ParallelEnumerable.AsSequential ()
- ParallelEnumerable.AsUnordered ()
- ParallelEnumerable.Average (sélecteur)
- ParallelEnumerable.Cast ()
- ParallelEnumerable.Concat (seconde)
- ParallelEnumerable.Contains (valeur)
- ParallelEnumerable.Contains (value, comparateur)
- ParallelEnumerable.Count ()
- ParallelEnumerable.Count (prédicat)
- ParallelEnumerable.DefaultIfEmpty ()
- ParallelEnumerable.DefaultIfEmpty (defaultValue)
- ParallelEnumerable.Distinct ()
- ParallelEnumerable.Distinct (comparateur)
- ParallelEnumerable.ElementAt (index)
- ParallelEnumerable.ElementAtOrDefault (index)
- ParallelEnumerable.Empty ()
- ParallelEnumerable.Except (second)
- ParallelEnumerable.Except (second, comparateur)
- ParallelEnumerable.First ()
- ParallelEnumerable.First (prédicat)
- ParallelEnumerable.FirstOrDefault ()
- ParallelEnumerable.FirstOrDefault (prédicat)
- ParallelEnumerable.ForAll (action)
- ParallelEnumerable.GroupBy (keySelector)
- ParallelEnumerable.GroupBy (keySelector, comparateur)
- ParallelEnumerable.GroupBy (keySelector, elementSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, comparateur)
- ParallelEnumerable.GroupBy (keySelector, resultSelector)

- ParallelEnumerable.GroupBy (keySelector, resultSelector, comparateur)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector, comparateur)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.GroupJoin (interne, outerKeySelector, innerKeySelector, resultSelector, comparateur)
- ParallelEnumerable.Intersect (second)
- ParallelEnumerable.Intersect (second, comparateur)
- ParallelEnumerable.Join (interne, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.Join (interne, outerKeySelector, innerKeySelector, resultSelector, comparateur)
- ParallelEnumerable.Last ()
- ParallelEnumerable.Last (prédicat)
- ParallelEnumerable.LastOrDefault ()
- ParallelEnumerable.LastOrDefault (prédicat)
- ParallelEnumerable.LongCount ()
- ParallelEnumerable.LongCount (prédicat)
- ParallelEnumerable.Max ()
- ParallelEnumerable.Max (sélecteur)
- ParallelEnumerable.Min ()
- ParallelEnumerable.Min (sélecteur)
- ParallelEnumerable.OfType ()
- ParallelEnumerable.OrderBy (keySelector)
- ParallelEnumerable.OrderBy (keySelector, comparateur)
- ParallelEnumerable.OrderByDescending (keySelector)
- ParallelEnumerable.OrderByDescending (keySelector, comparateur)
- ParallelEnumerable.Range (démarrer, compter)
- ParallelEnumerable.Repeat (élément, nombre)
- ParallelEnumerable.Reverse ()
- ParallelEnumerable.Select (sélecteur)
- ParallelEnumerable.SelectMany (sélecteur)
- ParallelEnumerable.SelectMany (collectionSelector, resultSelector)
- ParallelEnumerable.SequenceEqual (second)
- ParallelEnumerable.SequenceEqual (second, comparateur)
- ParallelEnumerable.Single ()
- ParallelEnumerable.Single (prédicat)
- ParallelEnumerable.SingleOrDefault ()
- ParallelEnumerable.SingleOrDefault (prédicat)
- ParallelEnumerable.Skip (count)
- ParallelEnumerable.SkipWhile (prédicat)
- ParallelEnumerable.Sum ()
- ParallelEnumerable.Sum (sélecteur)
- ParallelEnumerable.Take (count)
- ParallelEnumerable.TakeWhile (prédicat)
- ParallelEnumerable.ThenBy (keySelector)
- ParallelEnumerable.ThenBy (keySelector, comparateur)

- ParallelEnumerable.ThenByDescending (keySelector)
- ParallelEnumerable.ThenByDescending (keySelector, comparateur)
- ParallelEnumerable.ToArray ()
- ParallelEnumerable.ToDictionary (keySelector)
- ParallelEnumerable.ToDictionary (keySelector, comparateur)
- ParallelEnumerable.ToDictionary (elementSelector)
- ParallelEnumerable.ToDictionary (elementSelector, comparateur)
- ParallelEnumerable.ToList ()
- ParallelEnumerable.ToLookup (keySelector)
- ParallelEnumerable.ToLookup (keySelector, comparateur)
- ParallelEnumerable.ToLookup (keySelector, elementSelector)
- ParallelEnumerable.ToLookup (keySelector, elementSelector, comparateur)
- ParallelEnumerable.Union (second)
- ParallelEnumerable.Union (second, comparateur)
- ParallelEnumerable.Where (prédicat)
- ParallelEnumerable.WithCancellation (annulationToken)
- ParallelEnumerable.WithDegreeOfParallelism (degreeOfParallelism)
- ParallelEnumerable.WithExecutionMode (mode d'exécution)
- ParallelEnumerable.WithMergeOptions (mergeOptions)
- ParallelEnumerable.Zip (second, resultSelector)

Examples

Exemple simple

Cet exemple montre comment PLINQ peut être utilisé pour calculer les nombres pairs compris entre 1 et 10 000 en utilisant plusieurs threads. Notez que la liste ne sera pas commandée!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

WithDegreeOfParallelism

Le degré de parallélisme correspond au nombre maximal de tâches exécutées simultanément qui seront utilisées pour traiter la requête.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

AsOrdered

Cet exemple montre comment PLINQ peut être utilisé pour calculer les nombres pairs compris entre 1 et 10 000 en utilisant plusieurs threads. L'ordre sera maintenu dans la liste résultante, mais gardez à l'esprit `AsOrdered` peut nuire aux performances d'un grand nombre d'éléments, de sorte que le traitement non ordonné est préférable lorsque cela est possible.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

Comme unordorded

Les séquences ordonnées peuvent nuire aux performances lorsque vous manipulez un grand nombre d'éléments. Pour atténuer ce `AsUnordered` , il est possible d'appeler `AsUnordered` lorsque l'ordre des séquences n'est plus nécessaire.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
                          .OrderBy(x => x)
                          .Take(5000)
                          .AsUnordered()
                          .Where(x => x % 2 == 0) // This line won't be affected by ordering
                          .ToList();
```

Lire Parallèle LINQ (PLINQ) en ligne: https://riptutorial.com/fr/csharp/topic/3569/parallele-linq--plinq-

Chapitre 126: Plate-forme de compilation .NET (Roslyn)

Examples

Créer un espace de travail à partir d'un projet MSBuild

Obtenez d'abord le nuget `Microsoft.CodeAnalysis.CSharp.Workspaces` avant de continuer.

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

Pour charger du code existant dans l'espace de travail, compilez et signalez les erreurs. Le code sera ensuite stocké dans la mémoire. De là, le côté syntaxique et le côté sémantique seront disponibles pour travailler avec.

Arbre de syntaxe

Un **arbre de syntaxe** est une structure de données immuable représentant le programme sous la forme d'une arborescence de noms, de commandes et de marques (comme précédemment configuré dans l'éditeur).

Par exemple, supposons qu'une instance `Microsoft.CodeAnalysis.Compilation` nommée `compilation` ait été configurée. Il existe plusieurs façons de répertorier les noms de chaque variable déclarée dans le code chargé. Pour ce faire, prenez tous les éléments de la syntaxe dans chaque document (la méthode `DescendantNodes`) et utilisez Linq pour sélectionner les nœuds décrivant la déclaration de variable:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Chaque type de construction C # avec un type correspondant existera dans l'arbre de syntaxe. Pour trouver rapidement des types spécifiques, utilisez la fenêtre `Syntax Visualizer` de Visual Studio. Ceci interprétera le document ouvert actuel comme un arbre de syntaxe Roslyn.

Modèle sémantique

Un **modèle sémantique** offre un niveau plus élevé d'interprétation et de compréhension du code par rapport à un arbre de syntaxe. Lorsque les arbres de syntaxe peuvent indiquer les noms des variables, les modèles sémantiques indiquent également le type et toutes les références. Les arbres de syntaxe remarquent les appels de méthode, mais les modèles sémantiques donnent des références à l'emplacement précis de la méthode déclarée (après l'application d'une résolution de surcharge).

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379)"
            }
        }
    }
}
```

Cela génère une liste de variables locales à l'aide d'un arbre de syntaxe. Ensuite, il consulte le modèle sémantique pour obtenir le nom complet du type et trouver toutes les références de chaque variable.

Lire Plate-forme de compilation .NET (Roslyn) en ligne:

https://riptutorial.com/fr/csharp/topic/4886/plate-forme-de-compilation--net--roslyn-

Chapitre 127: Pointeurs

Remarques

Pointeurs et `unsafe`

En raison de leur nature, les pointeurs produisent du code invérifiable. Ainsi, l'utilisation de tout type de pointeur nécessite un contexte `unsafe` .

Le type `System.IntPtr` est un wrapper sûr autour d'un `void*` . Il est conçu comme une alternative plus pratique au `void*` lorsqu'un contexte dangereux n'est pas requis pour effectuer la tâche en cours.

Comportement non défini

Comme en C et C ++, l'utilisation incorrecte des pointeurs peut invoquer un comportement indéfini, avec des effets secondaires possibles, à savoir la corruption de la mémoire et l'exécution de code non intentionnel. En raison de la nature invérifiable de la plupart des opérations de pointeur, l'utilisation correcte des pointeurs est entièrement à la charge du programmeur.

Types prenant en charge les pointeurs

Contrairement à C et C ++, tous les types C # ne possèdent pas les types de pointeurs correspondants. Un type `T` peut avoir un type de pointeur correspondant si les deux critères suivants s'appliquent:

- `T` est un type de structure ou un type de pointeur.
- `T` contient uniquement les membres qui satisfont ces deux critères de manière récursive.

Examples

Pointeurs d'accès au tableau

Cet exemple montre comment les pointeurs peuvent être utilisés pour un accès de type C aux tableaux C #.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
```

```
            *(p + i) = i;
        }
    }
}
```

Le mot-clé `unsafe` est requis car l'accès au pointeur n'émettra aucune vérification des limites normalement émises lors de l'accès aux tableaux C # de manière régulière.

Le mot-clé `fixed` indique au compilateur C # d'émettre des instructions pour épingler l'objet d'une manière sûre. L'épinglage est nécessaire pour garantir que le garbage collector ne déplace pas le tableau en mémoire, car cela invaliderait les pointeurs pointant dans le tableau.

Arithmétique de pointeur

L'addition et la soustraction dans les pointeurs fonctionnent différemment des nombres entiers. Lorsqu'un pointeur est incrémenté ou décrémenté, l'adresse indiquée est augmentée ou diminuée de la taille du type de référence.

Par exemple, le type `int` (alias pour `System.Int32`) a une taille de 4. Si un `int` peut être stocké dans l'adresse 0, l' `int` ultérieur peut être stocké dans l'adresse 4, et ainsi de suite. Dans du code:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

De même, le type `long` (alias pour `System.Int64`) a une taille de 8. Si un `long` peut être stocké dans l'adresse 0, le `long` peut être stocké dans l'adresse 8, et ainsi de suite. Dans du code:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

Le type `void` est spécial et les pointeurs de `void` sont également spéciaux et ils sont utilisés comme pointeurs fourre-tout lorsque le type n'est pas connu ou n'a pas d'importance. En raison de leur nature agnostique, les pointeurs de `void` ne peuvent pas être incrémentés ou décrémentés:

```
var ptr = (void*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

L'astérisque fait partie du type

En C et C ++, l'astérisque dans la déclaration d'une variable de pointeur fait *partie de l'expression en* cours de déclaration. En C #, l'astérisque dans la déclaration fait *partie du type* .

En C, C ++ et C #, l'extrait de code suivant déclare un pointeur `int` :

```
int* a;
```

En C et C ++, l'extrait de code suivant déclare un pointeur `int` et une variable `int` . En C #, il déclare deux pointeurs `int` :

```
int* a, b;
```

En C et C ++, l'extrait de code suivant déclare deux pointeurs `int` . En C #, il est invalide:

```
int *a, *b;
```

vide*

C # hérite de C et C ++ l'utilisation de `void*` tant que pointeur agnostique et agnostique.

```
void* ptr;
```

Tout type de pointeur peut être assigné à `void*` utilisant une conversion implicite:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

L'inverse nécessite une conversion explicite:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Accès membre utilisant ->

C # hérite de C et C ++ l'utilisation du symbole `->` comme moyen d'accéder aux membres d'une instance via un pointeur typé.

Considérons la structure suivante:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

Voici un exemple d'utilisation de `->` pour accéder à ses membres:

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

Pointeurs génériques

Les critères qu'un type doit satisfaire pour prendre en charge les pointeurs (voir *Remarques*) ne peuvent pas être exprimés en termes de contraintes génériques. Par conséquent, toute tentative de déclaration d'un pointeur sur un type fourni via un paramètre de type générique échouera.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Lire Pointeurs en ligne: https://riptutorial.com/fr/csharp/topic/5524/pointeurs

Chapitre 128: Pointeurs & Code dangereux

Examples

Introduction au code non sécurisé

C # permet d'utiliser des variables de pointeur dans une fonction de bloc de code lorsqu'il est marqué par le modificateur `unsafe` . Le code non sécurisé ou le code non géré est un bloc de code qui utilise une variable de pointeur.

Un pointeur est une variable dont la valeur est l'adresse d'une autre variable, c'est-à-dire l'adresse directe de l'emplacement mémoire. similaire à toute variable ou constante, vous devez déclarer un pointeur avant de pouvoir l'utiliser pour stocker une adresse de variable.

La forme générale d'une déclaration de pointeur est la suivante:

```
type *var-name;
```

Voici les déclarations de pointeur valides:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

L'exemple suivant illustre l'utilisation de pointeurs en C #, en utilisant le modificateur unsafe:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

Lorsque le code ci-dessus est compilé et exécuté, le résultat est le suivant:

```
Data is: 20
Address is: 99215364
```

Au lieu de déclarer une méthode entière comme non sécurisée, vous pouvez également déclarer

une partie du code comme non sécurisée:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

Récupération de la valeur de données à l'aide d'un pointeur

Vous pouvez récupérer les données stockées à l'emplacement référencé par la variable de pointeur, à l'aide de la méthode ToString (). L'exemple suivant illustre ceci:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

Lorsque le code ci-dessus a été compilé et exécuté, il produit le résultat suivant:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Les pointeurs de passage comme paramètres des méthodes

Vous pouvez passer une variable de pointeur à une méthode en tant que paramètre. L'exemple suivant illustre ceci:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
```

```
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Accès aux éléments du tableau à l'aide d'un pointeur

En C #, un nom de tableau et un pointeur vers un type de données identique à celui des données du tableau ne sont pas du même type de variable. Par exemple, `int *p` et `int[] p` ne sont pas du même type. Vous pouvez incrémenter la variable de pointeur `p` car celle-ci n'est pas fixe en mémoire mais une adresse de tableau est fixe en mémoire et vous ne pouvez pas l'incrémenter.

Par conséquent, si vous avez besoin d'accéder à un tableau de données en utilisant une variable de pointeur, comme nous le faisons habituellement en C ou en C ++, vous devez corriger le pointeur à l'aide du mot clé fixe.

L'exemple suivant illustre ceci:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }
```

```
            Console.ReadKey();
        }
    }
}
```

Lorsque le code ci-dessus a été compilé et exécuté, il produit le résultat suivant:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

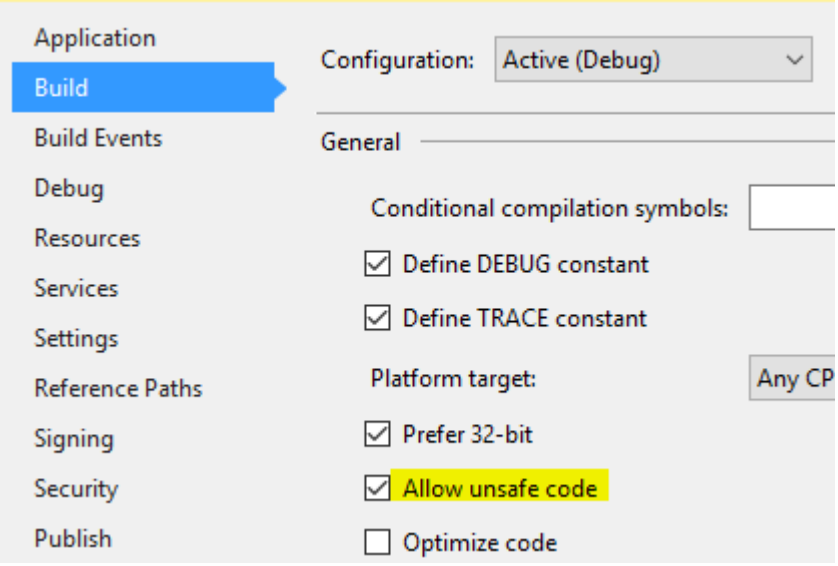
Compiler un code non sécurisé

Pour compiler du code non sécurisé, vous devez spécifier le commutateur de ligne de commande `/unsafe` avec le compilateur de ligne de commande.

Par exemple, pour compiler un programme nommé prog1.cs contenant du code non sécurisé, à partir de la ligne de commande, indiquez la commande suivante:

```
csc /unsafe prog1.cs
```

Si vous utilisez Visual Studio IDE, vous devez activer l'utilisation de code non sécurisé dans les propriétés du projet.



Pour faire ça:

- Ouvrez les propriétés du projet en double-cliquant sur le noeud des propriétés dans l'explorateur de solutions.
- Cliquez sur l'onglet Construire.
- Sélectionnez l'option "Autoriser le code non sécurisé"

Lire Pointeurs & Code dangereux en ligne: https://riptutorial.com/fr/csharp/topic/5514/pointeurs--amp--code-dangereux

Chapitre 129: Polymorphisme

Examples

Un autre exemple de polymorphisme

Le polymorphisme est l'un des piliers de la POO. Poly dérive d'un terme grec qui signifie «formes multiples».

Vous trouverez ci-dessous un exemple montrant le polymorphisme. La classe `Vehicle` prend plusieurs formes en tant que classe de base.

Les classes dérivées `Ducati` et `Lamborghini` héritent de `Vehicle` et remplacent la méthode `Display()` la classe de base pour afficher ses propres `NumberOfWheels` .

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
```

```
}
```

Voici l'extrait de code où Polymorphism est exposé. L'objet est créé pour le type de base `Vehicle` utilisant un `vehicle` variable sur la ligne 1. Il appelle la méthode de la classe de base `Display()` sur la ligne 2 et affiche la sortie comme indiqué.

```
static void Main(string[] args)
{
   Vehicle vehicle = new Vehicle();     //Line 1
   vehicle.Display();                   //Line 2
   vehicle = new Ducati();              //Line 3
   vehicle.Display();                   //Line 4
   vehicle = new Lamborghini();         //Line 5
   vehicle.Display();                   //Line 6
}
```

À la ligne 3, l'objet `vehicle` pointe vers la classe dérivée `Ducati` et appelle sa méthode `Display()` , qui affiche la sortie comme indiqué. Voici le comportement polymorphique, même si l'objet `vehicle` est de type de `Vehicle` , il appelle la méthode de la classe dérivée d' `Display()` comme le type `Ducati` remplace la classe de base d' `Display()` méthode, puisque le `vehicle` objet est pointé vers `Ducati` .

La même explication est applicable lorsqu'elle appelle la méthode `Display()` du type `Lamborghini` .

La sortie est indiquée ci-dessous

```
The number of wheels for the Vehicle is 0        // Line 2
The number of wheels for the Ducati is 2         // Line 4
The number of wheels for the Lamborghini is 4    // Line 6
```

Types de polymorphisme

Le polymorphisme signifie qu'une opération peut également être appliquée à des valeurs d'autres types.

Il existe plusieurs types de polymorphisme:

- **Polymorphisme ad hoc:**
  contient `function overloading` . L'objectif est qu'une méthode puisse être utilisée avec différents types sans avoir besoin d'être générique.
- **Polymorphisme paramétrique:**
  est l'utilisation de types génériques. Voir les génériques
- **Sous-typage:**
  la cible hérite d'une classe pour généraliser une fonctionnalité similaire

---

Polymorphisme ad hoc

La cible du `Ad hoc polymorphism` est de créer une méthode pouvant être appelée par différents

types de données sans avoir besoin de conversion de type dans l'appel de fonction ou les génériques. La méthode suivante (s) `sumInt(par1, par2)` peut être appelée avec différents types de données et possède pour chaque combinaison de types une implémentation propre:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Voici un exemple d'appel:

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 ));  //  3
    Console.WriteLine(sumInt("3","4"));  //  7
    Console.WriteLine(sumInt("5", 6 ));  // 11
    Console.WriteLine(sumInt( 7 ,"8"));  // 15
}
```

Sous-typage

Le sous-typage est l'utilisation de inherit d'une classe de base pour généraliser un comportement similaire:

```
public interface Car{
    void refuel();
```

```
}

public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

Les deux classes `NormalCar` et `ElectricCar` ont maintenant une méthode pour se ravitailler, mais leur propre implémentation. Voici un exemple:

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

La sortie sera la suivante:

> Faire le plein d'essence
> Charge de la batterie

Lire Polymorphisme en ligne: https://riptutorial.com/fr/csharp/topic/1589/polymorphisme

Chapitre 130: Pour commencer: Json avec C

Introduction

La rubrique suivante présentera un moyen de travailler avec Json en utilisant le langage C # et les concepts de sérialisation et de désérialisation.

Examples

Exemple Json Simple

```
{
    "id": 89,
    "name": "Aldous Huxley",
    "type": "Author",
    "books":[{
                "name": "Brave New World",
                "date": 1932
             },
             {
                "name": "Eyeless in Gaza",
                "date": 1936
             },
             {
                "name": "The Genius and the Goddess",
                "date": 1955
             }]
}
```

Si vous êtes nouveau dans Json, voici un tutoriel illustré .

Les premières choses: la bibliothèque pour travailler avec Json

Pour travailler avec Json en utilisant C #, il est nécessaire d'utiliser Newtonsoft (bibliothèque .net). Cette bibliothèque fournit des méthodes permettant au programmeur de sérialiser et de désérialiser des objets, etc. Il y a un tutoriel si vous voulez connaître les détails sur ses méthodes et ses utilisations.

Si vous utilisez Visual Studio, accédez à *Outils / Gestionnaire de packages Nuget / Gérer le package vers la solution* / et tapez «Newtonsoft» dans la barre de recherche et installez le package. Si vous ne possédez pas NuGet, ce tutoriel détaillé pourrait vous aider.

Implémentation C

Avant de lire du code, il est important de comprendre les principaux concepts qui aideront à programmer les applications utilisant json.

Sérialisation : Processus de conversion d'un objet en flux d'octets pouvant être envoyé via des applications. Le code suivant peut être sérialisé et converti dans json précédent.

Désérialisation : Processus de conversion d'un flux d'octets / json en objet. C'est exactement le processus inverse de la sérialisation. Le json précédent peut être désérialisé en un objet C # comme le montrent les exemples ci-dessous.

Pour cela, il est important de transformer la structure json en classes afin d'utiliser les processus déjà décrits. Si vous utilisez Visual Studio, vous pouvez transformer automatiquement un json en une classe simplement en sélectionnant *"Editer / Coller Spécial / Coller JSON en tant que classes"* et en collant la structure json.

```
using Newtonsoft.Json;

  class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;       //"id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type= type;
        this.books = books;
    }
}

 class Book
{
   [JsonProperty("name")]
   public string name;
   [JsonProperty("date")]
   public DateTime date;
}
```

La sérialisation

```
 static void Main(string[] args)
    {
        Book[] books = new Book[3];
        Author author = new Author(89,"Aldous Huxley","Author",books);
        string objectDeserialized = JsonConvert.SerializeObject(author);
        //Converting author into json
    }
```

La méthode ".SerializeObject" reçoit en paramètre un *objet type* , vous pouvez donc y mettre n'importe quoi.

Désérialisation

Vous pouvez recevoir un fichier json de n'importe où, d'un fichier ou même d'un serveur. Il n'est donc pas inclus dans le code suivant.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

La méthode ".DeserializeObject" désérialise " *jsonExample* " en un objet " *Author* ". C'est pourquoi il est important de définir les variables json dans la définition des classes, de sorte que la méthode y accède pour la remplir.

Fonction Serialization & De-Serialization Common Utilities

Cet exemple utilisait une fonction commune pour toute la sérialisation et la désérialisation des objets de type.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
public static class IGUtilities
{
        public static string Serialization(this T obj)
        {
        string data = JsonConvert.SerializeObject(obj);
        return data;
        }

        public static T Deserialization(this string JsonData)
        {
        T copy = JsonConvert.DeserializeObject(JsonData);
        return copy;
        }

         public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
}
}
```

Lire Pour commencer: Json avec C # en ligne: https://riptutorial.com/fr/csharp/topic/9910/pour-commencer--json-avec-c-sharp

Chapitre 131: Prise asynchrone

Introduction

En utilisant des sockets asynchrones, un serveur peut écouter les connexions entrantes et faire une autre logique dans le même temps, contrairement au socket synchrone lorsqu'il écoute le thread principal et que l'application ne répond plus.

Remarques

Socket et réseau

Comment accéder à un serveur en dehors de mon propre réseau? C'est une question courante et lorsqu'elle est posée, elle est principalement marquée comme sujet.

Du côté serveur

Sur le réseau de votre serveur, vous devez transférer votre routeur sur votre serveur.

Pour un exemple de PC sur lequel le serveur est exécuté:

IP locale = `192.168.1.115`

Le serveur écoute le port 1234.

Transférer les connexions entrantes sur le `Port 1234` routeur vers `192.168.1.115`

Côté client

La seule chose que vous devez changer est l'IP. Vous ne souhaitez pas vous connecter à votre adresse de bouclage mais à l'adresse IP publique du réseau sur lequel votre serveur est exécuté. Cette adresse IP, vous pouvez obtenir ici .

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

Alors maintenant, vous créez une requête sur ce noeud final: `10.10.10.10:1234` si vous avez fait un `10.10.10.10:1234` propriété de votre routeur, votre serveur et votre client se connecteront sans aucun problème.

Si vous voulez vous connecter à une adresse IP locale, vous n'aurez pas besoin de changer l'adresse de bouclage en `192.168.1.178` ou quelque chose comme ça.

Envoi de données:

Les données sont envoyées dans un tableau d'octets. Vous devez emballer vos données dans un tableau d'octets et les décompresser de l'autre côté.

Si vous êtes familier avec socket, vous pouvez également essayer de chiffrer votre tableau d'octets avant de l'envoyer. Cela empêchera quiconque de voler votre colis.

Examples

Exemple de socket asynchrone (client / serveur).

Exemple côté serveur

Créer un écouteur pour le serveur

Commencez par créer un serveur qui gérera les clients qui se connectent et les requêtes qui seront envoyées. Donc, créez une classe d'écoute qui gérera cela.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming
connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    }
 }
```

Nous devons d'abord initialiser le socket Listener sur lequel nous pouvons écouter toutes les connexions. Nous allons utiliser un socket TCP, c'est pourquoi nous utilisons SocketType.Stream. Aussi, nous spécifions à quel port le serveur doit écouter

Ensuite, nous commençons à écouter toutes les connexions entrantes.

Les méthodes d'arborescence que nous utilisons ici sont:

1. ListenerSocket.Bind ();

   Cette méthode lie le socket à un IPEndPoint . Cette classe contient les informations d'hôte et de port local ou distant nécessaires à une application pour se connecter à un service sur un hôte.

2. ListenerSocket.Listen (10);

   Le paramètre backlog spécifie le nombre de connexions entrantes pouvant être mises en file d'attente pour acceptation.

3. ListenerSocket.BeginAccept ();

   Le serveur commence à écouter les connexions entrantes et continue avec une autre logique. Lorsqu'il y a une connexion, le serveur revient à cette méthode et exécute la méthode AcceptCallBack

```
    public void StartListening()
    {
        try
        {
                MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
                ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
                ListenerSocket.Listen(10);
                ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch(Exception ex)
        {
            throw new Exception("listening error" + ex);
        }
    }
```

Ainsi, lorsqu'un client se connecte, nous pouvons les accepter par cette méthode:

Trois méthodes que nous utilisons ici sont:

1. ListenerSocket.EndAccept ()

   Nous avons commencé le rappel avec `Listener.BeginAccept()` maintenant nous devons terminer cet appel. `The EndAccept()` méthode `The EndAccept()` accepte un paramètre IAsyncResult, cela stockera l'état de la méthode asynchrone. À partir de cet état, nous pouvons extraire le socket d'où provient la connexion entrante.

2. `ClientController.AddClient()`

   Avec le socket obtenu à partir d' `EndAccept()` nous créons un client avec une méthode propre *(code ClientController sous exemple de serveur)* .

3. ListenerSocket.BeginAccept ()

   Nous devons recommencer à écouter lorsque le socket est terminé en gérant la nouvelle connexion. Transmettez la méthode qui interceptera ce rappel. Et aussi passer dans le socket Listener afin que nous puissions réutiliser ce socket pour les prochaines connexions.

```
    public void AcceptCallback(IAsyncResult ar)
    {
        try
        {
            Console.WriteLine($"Accept CallBack port:{Port} protocol type:
{ProtocolType.Tcp}");
            Socket acceptedSocket = ListenerSocket.EndAccept(ar);
            ClientController.AddClient(acceptedSocket);

            ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch (Exception ex)
        {
            throw new Exception("Base Accept error"+ ex);
        }
    }
```

Maintenant, nous avons un socket d'écoute, mais comment recevons-nous les données envoyées par le client, comme le montre le code suivant.

Créer un récepteur de serveur pour chaque client

Tout d'abord créer une classe de réception avec un constructeur qui prend en paramètre Socket:

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

Dans la méthode suivante, nous commençons par donner au tampon une taille de 4 octets (Int32) ou le paquet contient des parties {longueur, données réelles}. Donc, les 4 premiers octets, nous réservons pour la longueur des données le reste pour les données réelles.

Ensuite, nous utilisons la méthode BeginReceive () . Cette méthode est utilisée pour commencer à recevoir des clients connectés et lorsqu'elle recevra des données, elle exécutera la fonction `ReceiveCallback` .

```
    public void StartReceiving()
    {
        try
        {
            _buffer = new byte[4];
            _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
ReceiveCallback, null);
        }
        catch {}
    }

    private void ReceiveCallback(IAsyncResult AR)
    {
        try
        {
            // if bytes are less than 1 takes place when a client disconnect from the server.
            // So we run the Disconnect function on the current client
            if (_receiveSocket.EndReceive(AR) > 1)
            {
                // Convert the first 4 bytes (int 32) that we received and convert it to an
Int32 (this is the size for the coming data).
                _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
                // Next receive this data into the buffer with size that we did receive before
                _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
                // When we received everything its onto you to convert it into the data that
you've send.
                // For example string, int etc... in this example I only use the
implementation for sending and receiving a string.

                // Convert the bytes to string and output it in a message box
                string data = Encoding.Default.GetString(_buffer);
```

```
                MessageBox.Show(data);
                // Now we have to start all over again with waiting for a data to come from
the socket.
                StartReceiving();
            }
            else
            {
                Disconnect();
            }
        }
        catch
        {
            // if exeption is throw check if socket is connected because than you can
startreive again else Dissconect
            if (!_receiveSocket.Connected)
            {
                Disconnect();
            }
            else
            {
                StartReceiving();
            }
        }
    }

    private void Disconnect()
    {
        // Close connection
        _receiveSocket.Disconnect(true);
        // Next line only apply for the server side receive
        ClientController.RemoveClient(_clientId);
        // Next line only apply on the Client Side receive
        Here you want to run the method TryToConnect()
    }
```

Nous avons donc configuré un serveur capable de recevoir et d'écouter les connexions entrantes. Lorsqu'un client se connecte, il sera ajouté à une liste de clients et chaque client a sa propre classe de réception. Pour que le serveur écoute:

```
Listener listener = new Listener();
listener.StartListening();
```

Quelques classes que j'utilise dans cet exemple

```
    class Client
    {
        public Socket _socket { get; set; }
        public ReceivePacket Receive { get; set; }
        public int Id { get; set; }

        public Client(Socket socket, int id)
        {
            Receive = new ReceivePacket(socket, id);
            Receive.StartReceiving();
            _socket = socket;
            Id = id;
        }
    }
```

```
static class ClientController
{
     public static List<Client> Clients = new List<Client>();

     public static void AddClient(Socket socket)
     {
         Clients.Add(new Client(socket,Clients.Count));
     }

     public static void RemoveClient(int id)
     {
         Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
     }
 }
```

Exemple côté client

Connexion au serveur

Tout d'abord, nous voulons créer une classe qui se connecte au serveur avec le nom que nous lui donnons: Connecteur:

```
class Connector
{
    private Socket _connectingSocket;
}
```

La méthode suivante pour cette classe est TryToConnect ()

Cette méthode suscite quelques intérêts:

1. Créer le socket;
2. Ensuite je boucle jusqu'à ce que le socket soit connecté
3. Chaque boucle, il ne fait que tenir le fil pendant 1 seconde, nous ne voulons pas DOS le serveur XD
4. Avec Connect (), il essaiera de se connecter au serveur. Si cela échoue, une exception sera lancée, mais le serveur gardera le programme connecté au serveur. Vous pouvez utiliser une méthode Connect CallBack pour cela, mais je vais simplement appeler une méthode lorsque le socket est connecté.
5. Notez que le client essaie maintenant de se connecter à votre PC local sur le port 1234.

```
 public void TryToConnect()
 {
     _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

      while (!_connectingSocket.Connected)
      {
          Thread.Sleep(1000);
```

```
          try
          {
              _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
 1234));
          }
          catch { }
      }
      SetupForReceiveing();
     }
 }

 private void SetupForReceiveing()
 {
    // View Client Class bottom of Client Example
     Client.SetClient(_connectingSocket);
     Client.StartReceiving();
 }
```

Envoi d'un message au serveur

Nous avons donc maintenant une application presque terminée ou Socket. La seule chose que nous n'avons pas jet est une classe pour envoyer un message au serveur.

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what hapends here:
                 1. Create a list of bytes
                 2. Add the length of the string to the list.
                    So if this message arrives at the server we can easily read the length of
the coming message.
                 3. Add the message(string) bytes
            */

            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}*/
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
```

Enfin, créez deux boutons l'un pour vous connecter et l'autre pour envoyer un message:

```
private void ConnectClick(object sender, EventArgs e)
{
    Connector tpp = new Connector();
    tpp.TryToConnect();
}

private void SendClick(object sender, EventArgs e)
{
    Client.SendString("Test data from client");
}
```

La classe de client que j'ai utilisée dans cet exemple

```
public static void SetClient(Socket socket)
{
    Id = 1;
    Socket = socket;
    Receive = new ReceivePacket(socket, Id);
    SendPacket = new SendPacket(socket);
}
```

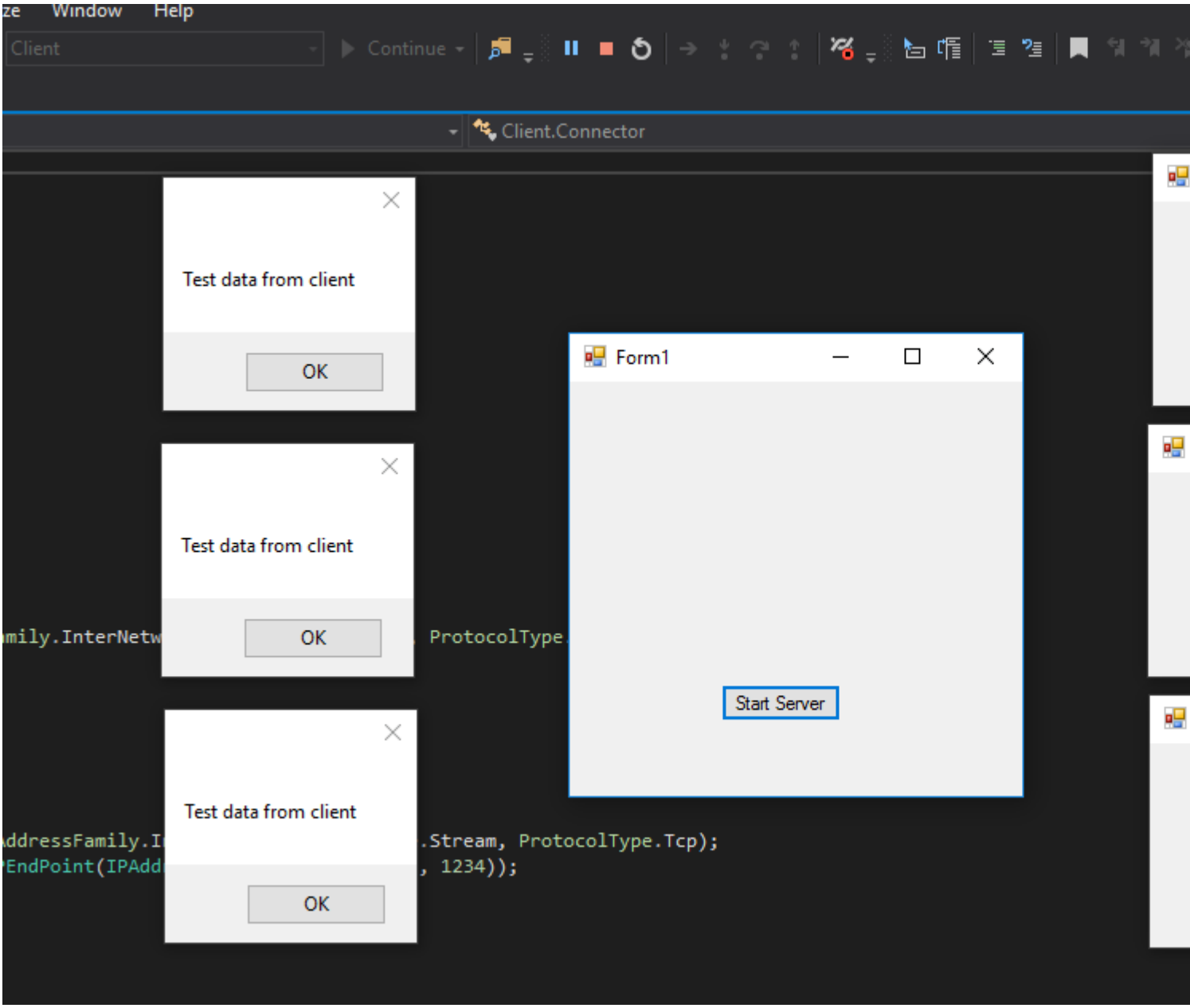
Remarquer

La classe de réception du serveur est identique à la classe de réception du client.

Conclusion

Vous avez maintenant un serveur et un client. Vous pouvez utiliser cet exemple de base. Par exemple, faites en sorte que le serveur puisse également recevoir des fichiers ou d'autres éléments. Ou envoyez un message au client. Dans le serveur, vous avez une liste de clients, donc lorsque vous recevez quelque chose que vous connaissez avec le client, il vient.

Résultat final:



Lire Prise asynchrone en ligne: https://riptutorial.com/fr/csharp/topic/9638/prise-asynchrone

Chapitre 132: Programmation fonctionnelle

Examples

Func et Action

Func fournit un support pour les fonctions anonymes paramétrées. Les principaux types sont les entrées et le dernier type est toujours la valeur de retour.

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

Les objets d' **action** sont comme des méthodes vides, ils n'ont donc qu'un type d'entrée. Aucun résultat n'est placé sur la pile d'évaluation.

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

Immutabilité

L'imputabilité est courante dans la programmation fonctionnelle et rare dans la programmation orientée objet.

Créez, par exemple, un type d'adresse avec un état mutable:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
```

```
}
```

Tout morceau de code pourrait altérer toute propriété de l'objet ci-dessus.

Créez maintenant le type d'adresse immuable:

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City  = city;
    }
}
```

Gardez à l’esprit qu’avoir des collections en lecture seule ne respecte pas l’immutabilité. Par exemple,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

n'est pas immuable, car l'utilisateur de l'objet peut modifier la collection (ajouter ou supprimer des éléments). Pour le rendre immuable, il faut soit utiliser une interface comme IEnumerable, qui n'expose pas de méthodes à ajouter, soit en faire une ReadOnlyCollection.

```
public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }
}

List<Students> list = new List<Student>();
// add students
Classroom c = new Classroom(list.AsReadOnly());
```

Avec l'objet immuable, nous avons les avantages suivants:

- Il sera dans un état connu (autre code ne peut pas le changer).
- C'est thread-safe.

- Le constructeur offre un lieu unique pour la validation.
- Savoir que l'objet ne peut pas être modifié rend le code plus facile à comprendre.

Éviter les références nulles

Les développeurs C # ont beaucoup d'exceptions de référence à traiter. Les développeurs F # ne le font pas car ils ont le type Option. Un type Option <> (certains préfèrent peut-être <> comme nom) fournit un type de retour Some et a None. Il est explicite qu'une méthode est sur le point de retourner un enregistrement nul.

Par exemple, vous ne pouvez pas lire ce qui suit et savoir si vous devrez gérer une valeur nulle.

```
var user = _repository.GetUser(id);
```

Si vous connaissez le possible null, vous pouvez introduire un code standard pour le gérer.

```
var username = user != null ? user.Name : string.Empty;
```

Que se passe-t-il si une option <> est retournée à la place?

```
Option<User> maybeUser = _repository.GetUser(id);
```

Le code rend maintenant explicite le fait que nous pouvons avoir un enregistrement None et le code passe-partout pour vérifier certains ou aucun est requis:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

La méthode suivante montre comment retourner une option <>

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Voici une implémentation minimale d'Option <>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
```

```
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

Pour démontrer ce qui précède, AvoidNull.csx peut être exécuté avec le C # REPL.

Comme indiqué, il s'agit d'une implémentation minimale. Une recherche de paquets "Maybe" NuGet permettra de créer un certain nombre de bonnes bibliothèques.

Fonctions d'ordre supérieur

Une fonction d'ordre supérieur est une fonction qui prend une autre fonction en tant qu'argument ou retourne une fonction (ou les deux).

Cela se fait généralement avec lambdas, par exemple lors du passage d'un prédicat à une clause LINQ Where:

```
var results = data.Where(p => p.Items == 0);
```

La clause Where () pourrait recevoir de nombreux prédicats différents, ce qui lui confère une flexibilité considérable.

Le passage d'une méthode à une autre méthode est également visible lors de la mise en œuvre du modèle de conception de stratégie. Par exemple, différentes méthodes de tri peuvent être choisies et passées dans une méthode de tri sur un objet en fonction des exigences lors de l'exécution.

Collections immuables

Le package `System.Collections.Immutable` NuGet fournit des classes de collection immuables.

Créer et ajouter des éléments

```
var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

Création à l'aide du constructeur

Certaines collections immuables ont une classe interne `Builder` qui peut être utilisée pour créer à moindre coût de grandes instances immuables:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

Créer à partir d'un IEnumerable existant

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

Liste de tous les types de collection immuables:

- `System.Collections.Immutable.ImmutableArray<T>`
- `System.Collections.Immutable.ImmutableDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableHashSet<T>`
- `System.Collections.Immutable.ImmutableList<T>`
- `System.Collections.Immutable.ImmutableQueue<T>`
- `System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableSortedSet<T>`
- `System.Collections.Immutable.ImmutableStack<T>`

Lire Programmation fonctionnelle en ligne:
https://riptutorial.com/fr/csharp/topic/2564/programmation-fonctionnelle

Chapitre 133: Programmation orientée objet en C

Introduction

Ce sujet essaie de nous dire comment nous pouvons écrire des programmes basés sur l'approche OOP. Mais nous n'essayons pas d'enseigner le paradigme de la programmation orientée objet. Nous traiterons des sujets suivants: Classes, Propriétés, Héritage, Polymorphisme, Interfaces, etc.

Examples

Des classes:

Le squelette de la classe déclarante est:

<>: Obligatoire

[]:Optionnel

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited
class][,][[Interface Name 1],[Interface Name 2],...]
{
    //Your code
}
```

Ne vous inquiétez pas si vous ne comprenez pas toute la syntaxe, nous allons nous familiariser avec tout cela. Pour le premier exemple, considérez la classe suivante:

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

Dans cette classe, nous créons la variable `i` avec `int` type et avec la méthode par défaut Access Modifiers et `getMyValue()` avec des modificateurs d'accès public.

Lire Programmation orientée objet en C # en ligne:
https://riptutorial.com/fr/csharp/topic/9856/programmation-orientee-objet-en-c-sharp

Chapitre 134: Propriétés

Remarques

Les propriétés combinent le stockage des données de classe des champs avec l'accessibilité des méthodes. Parfois, il peut être difficile de décider d'utiliser une propriété, une propriété faisant référence à un champ ou une méthode faisant référence à un champ. En règle générale:

- Les propriétés doivent être utilisées sans champ interne si elles n'obtiennent et / ou ne définissent que des valeurs; sans autre logique. Dans de tels cas, l'ajout d'un champ interne ajouterait du code sans avantage.
- Les propriétés doivent être utilisées avec des champs internes lorsque vous devez manipuler ou valider les données. Un exemple peut consister à supprimer les espaces de début et de fin des chaînes ou à s'assurer qu'une date n'est pas passée.

En ce qui concerne les méthodes et les propriétés, où vous pouvez récupérer (`get`) et mettre à jour (`set`) une valeur, une propriété est le meilleur choix. En outre, .Net fournit de nombreuses fonctionnalités qui utilisent la structure d'une classe; par exemple, en ajoutant une grille à un formulaire, .Net affichera par défaut toutes les propriétés de la classe sur ce formulaire; pour tirer le meilleur parti de ces conventions, prévoyez d'utiliser des propriétés lorsque ce comportement est généralement souhaitable, ainsi que des méthodes dans lesquelles vous préféreriez que les types ne soient pas ajoutés automatiquement.

Examples

Diverses propriétés en contexte

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned
through the public property.
    public DateTime DOB
    {
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
```

```
        }
    }

    //this is not a property but a method; though it could be rewritten as a property if
desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}
```

Public Get

Les getters sont utilisés pour exposer les valeurs des classes.

```
string name;
public string Name
{
    get { return this.name; }
}
```

Ensemble public

Les paramètres sont utilisés pour affecter des valeurs aux propriétés.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

Accès aux propriétés

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
```

```
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = "   Hans Trimmer  ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only
accessible within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as
there's validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
```

```
    }
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB.  The date
must not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}
```

Valeurs par défaut pour les propriétés

Vous pouvez définir une valeur par défaut en utilisant les initialiseurs (C # 6)

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

S'il est en lecture seule, vous pouvez renvoyer des valeurs comme ceci:

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
  }
```

Propriétés implémentées automatiquement

Les propriétés implémentées automatiquement ont été introduites dans C # 3.
Une propriété auto-implémentée est déclarée avec un getter et un setter vides (accesseurs):

```
public bool IsValid { get; set; }
```

Lorsqu'une propriété implémentée automatiquement est écrite dans votre code, le compilateur crée un champ anonyme privé auquel il est uniquement possible d'accéder via les accesseurs de la propriété.

La déclaration de propriété implémentée ci-dessus équivaut à écrire ce long code:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

Les propriétés implémentées automatiquement ne peuvent avoir aucune logique dans leurs accesseurs, par exemple:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

Une propriété implémentée automatiquement *peut* cependant avoir différents modificateurs d'accès pour ses accesseurs:

```
public bool IsValid { get; private set; }
```

C # 6 permet aux propriétés implémentées automatiquement de ne pas avoir de setter du tout (ce qui le rend immuable, puisque sa valeur ne peut être définie qu'à l'intérieur du constructeur ou codée en dur):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

Pour plus d'informations sur l'initialisation des propriétés implémentées automatiquement, consultez la documentation relative aux initialiseurs de propriétés automatiques .

Propriétés en lecture seule

Déclaration

Un malentendu courant, en particulier pour les débutants, est que la propriété en lecture seule est celle marquée avec le mot-clé `readonly` . Ce n'est pas correct et en fait, il *y a une erreur de compilation* :

```
public readonly string SomeProp { get; set; }
```

Une propriété est en lecture seule lorsqu'elle ne contient qu'un getter.

```
public string SomeProp { get; }
```

Utilisation de propriétés en lecture seule pour créer des classes immuables

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

Lire Propriétés en ligne: https://riptutorial.com/fr/csharp/topic/49/proprietes

Chapitre 135: Récursivité

Remarques

Notez que l'utilisation de la récursivité peut avoir un impact important sur votre code, car chaque appel de fonction récursif sera ajouté à la pile. S'il y a trop d'appels, cela pourrait entraîner une exception **StackOverflow** . La plupart des "fonctions récursives naturelles" peuvent être écrites comme une construction en boucle `for` , `while` ou `foreach` , et bien que ne semblant pas si **chic** ou **intelligent,** elles seront plus efficaces.

Pensez toujours à deux fois et utilisez la récursivité avec soin - sachez pourquoi vous l'utilisez:

- la récursivité doit être utilisée lorsque vous savez que le nombre d'appels récursifs n'est pas *excessif*
    - *des* moyens *excessifs* , cela dépend de la quantité de mémoire disponible
- la récursivité est utilisée car il s'agit d'une version de code plus claire et plus propre, plus lisible qu'une fonction itérative ou basée sur une boucle. C'est souvent le cas parce que cela donne un code plus propre et plus compact (autrement dit moins de lignes de code).
    - mais soyez conscient, cela peut être moins efficace! Par exemple, dans la récursion de Fibonacci, pour calculer le *nième* numéro de la séquence, le temps de calcul augmentera de manière exponentielle!

Si vous voulez plus de théorie, veuillez lire:

- https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

Examples

Décrire récursivement une structure d'objet

La récursivité se produit lorsqu'une méthode s'appelle elle-même. De préférence, il le fera jusqu'à ce qu'une condition spécifique soit remplie, puis il quittera la méthode normalement, pour revenir au point d'où la méthode a été appelée. Si ce n'est pas le cas, une exception de dépassement de capacité de la pile peut se produire en raison d'un trop grand nombre d'appels récursifs.

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
    public string Name { get; set; }
    public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
```

```
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}
```

Récursivité en anglais

La récursion peut être définie comme suit:

> Une méthode qui s'appelle jusqu'à ce qu'une condition spécifique soit remplie.

Un exemple simple et excellent de récursivité est une méthode qui obtiendra la factorielle d'un nombre donné:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

Dans cette méthode, nous pouvons voir que la méthode prendra un argument, `number` .

Pas à pas:

Prenons l'exemple, exécutant `Factorial(4)`

1. Est le `number (4) == 1` ?
2. Non? retour `4 * Factorial(number-1)` (3)
3. Comme la méthode est appelée à nouveau, elle répète maintenant la première étape en utilisant `Factorial(3)` comme nouvel argument.
4. Cela continue jusqu'à ce que `Factorial(1)` soit exécuté et que le `number (1) == 1` renvoie 1.
5. Globalement, le calcul "se construit" `4 * 3 * 2 * 1` et retourne finalement 24.

La clé pour comprendre la récursivité est que la méthode appelle une *nouvelle instance* d'elle-même. Après retour, l'exécution de l'instance appelante se poursuit.

Utilisation de la récursivité pour obtenir l'arborescence de répertoires

L'une des utilisations de la récursivité consiste à naviguer dans une structure de données hiérarchique, telle qu'une arborescence de répertoires de systèmes de fichiers, sans connaître le nombre de niveaux de l'arborescence ni le nombre d'objets sur chaque niveau. Dans cet exemple, vous verrez comment utiliser la récursivité dans une arborescence de répertoires pour rechercher tous les sous-répertoires d'un répertoire spécifié et imprimer l'arborescence complète dans la console.

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
            if (!Directory.Exists(rootDirectoryPath))
            {
                throw new DirectoryNotFoundException(
                    $"Directory '{rootDirectoryPath}' not found.");
            }

            var rootDirectory = new DirectoryInfo(rootDirectoryPath);
            PrintDirectoryTree(rootDirectory, RootLevel);
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
```

```
        }
    }

    private static void PrintDirectoryTree(
        DirectoryInfo directory, int currentLevel)
    {
        var indentation = string.Empty;
        for (var i = RootLevel; i < currentLevel; i++)
        {
            indentation += Tab;
        }

        Console.WriteLine($"{indentation}-{directory.Name}");
        var nextLevel = currentLevel + 1;
        try
        {
            foreach (var subDirectory in directory.GetDirectories())
            {
                PrintDirectoryTree(subDirectory, nextLevel);
            }
        }
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine($"{indentation}-{e.Message}");
        }
    }
}
```

Ce code est un peu plus compliqué que le strict minimum pour effectuer cette tâche, car il inclut la vérification des exceptions pour gérer les problèmes liés à l'obtention des répertoires. Vous trouverez ci-dessous une ventilation du code en segments plus petits avec des explications sur chacun.

`Main` :

La méthode principale prend une entrée d'un utilisateur sous forme de chaîne, qui doit être utilisée comme chemin d'accès au répertoire racine. Il appelle ensuite la méthode `PrintDirectoryTree` avec cette chaîne comme paramètre.

`PrintDirectoryTree(string)` :

C'est la première des deux méthodes qui gèrent l'impression de l'arborescence de répertoires. Cette méthode prend en paramètre une chaîne représentant le chemin d'accès au répertoire racine. Il vérifie si le chemin est un répertoire réel et, sinon, lève une exception `DirectoryNotFoundException` qui est ensuite traitée dans le bloc catch. Si le chemin est un répertoire réel, un `DirectoryInfo` objet `rootDirectory` est créé à partir du chemin d' accès, et le second `PrintDirectoryTree` méthode est appelée avec le `rootDirectory` objet et `RootLevel` , qui est un nombre entier constant avec une valeur de zéro.

`PrintDirectoryTree(DirectoryInfo, int)` :

Cette seconde méthode traite le plus gros du travail. Il prend un `DirectoryInfo` et un entier comme paramètres. `DirectoryInfo` est le répertoire en cours et l'entier est la profondeur du répertoire par rapport à la racine. Pour faciliter la lecture, la sortie est en retrait pour chaque niveau de

profondeur du répertoire en cours, de sorte que la sortie ressemble à ceci:

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

Une fois le répertoire en cours imprimé, ses sous-répertoires sont récupérés, et cette méthode est ensuite appelée sur chacun d'eux avec une valeur de niveau de profondeur supérieure au courant. Cette partie est la récursivité: la méthode elle-même. Le programme s'exécutera de cette manière jusqu'à ce qu'il ait visité tous les répertoires de l'arborescence. Lorsqu'il atteint un répertoire sans sous-répertoires, la méthode est automatiquement renvoyée.

Cette méthode intercepte également une `UnauthorizedAccessException` , qui est lancée si l'un des sous-répertoires du répertoire en cours est protégé par le système. Le message d'erreur est imprimé au niveau d'indentation actuel pour la cohérence.

La méthode ci-dessous propose une approche plus simple de ce problème:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Cela n'inclut pas la vérification d'erreur spécifique ou le formatage de sortie de la première approche, mais cela fait effectivement la même chose. Comme il n'utilise que des chaînes contrairement à `DirectoryInfo` , il ne peut pas donner accès à d'autres propriétés de répertoire telles que les autorisations.

Séquence de Fibonacci

Vous pouvez calculer un nombre dans la séquence de Fibonacci en utilisant la récursivité.

Suivant la théorie mathématique de F (n) = F (n-2) + F (n-1), pour tout i> 0,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
```

```
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

Calcul factoriel

La factorielle d'un nombre (noté avec!, Comme par exemple 9!) Est la multiplication de ce nombre par la factorielle de un inférieur. Donc, par exemple, 9! = 9 x 8! = 9 x 8 x 7! = 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1.

Donc, dans le code qui devient, en utilisant la récursivité:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

Calcul PowerOf

Le calcul de la puissance d'un nombre donné peut également se faire de manière récursive. Étant donné un nombre de base `n` et un exposant `e` , nous devons nous assurer de diviser le problème en plusieurs parties en diminuant l’exposant `e` .

Exemple théorique:

- $2^2 = 2x2$
- $2^3 = 2x2x2$ ou, $2^3 = 2^2 \times 2$
  C'est là que réside le secret de notre algorithme récursif (voir le code ci-dessous). Il s’agit de prendre le problème et de le séparer en plusieurs parties plus petites et plus simples pour résoudre les problèmes.
- **Remarques**
  - lorsque le nombre de base est 0, il faut savoir que 0 est $0^3 = 0 \times 0 \times 0$
  - quand l'exposant est à 0, il faut être conscient de toujours retourner 1, car c'est une règle mathématique.

Exemple de code:

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,
aka: 2³ = 2 * 2² etc..
}
```

Tests dans xUnit pour vérifier la logique:
Bien que cela ne soit pas nécessaire, il est toujours bon d'écrire des tests pour vérifier votre logique. J'inclus ceux qui sont écrits ici dans le framework xUnit .

```
    [Theory]
    [MemberData(nameof(PowerOfTestData))]
    public void PowerOfTest(int @base, int exponent, int expected) {
        Assert.Equal(expected, CalcPowerOf(@base, exponent));
    }

    public static IEnumerable<object[]> PowerOfTestData() {
        yield return new object[] { 0, 0, 0 };
        yield return new object[] { 0, 1, 0 };
        yield return new object[] { 2, 0, 1 };
        yield return new object[] { 2, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 2, 25 };
        yield return new object[] { 5, 3, 125 };
        yield return new object[] { 5, 4, 625 };
}
```

Lire Récursivité en ligne: https://riptutorial.com/fr/csharp/topic/2470/recursivite

Chapitre 136: Réflexion

Introduction

Reflection est un mécanisme de langage C # permettant d'accéder aux propriétés des objets dynamiques lors de l'exécution. En règle générale, la réflexion est utilisée pour récupérer les informations sur le type d'objet dynamique et les valeurs d'attribut d'objet. Dans l'application REST, par exemple, la réflexion peut être utilisée pour parcourir un objet de réponse sérialisé.

Remarque: Selon les directives MS, le code de performance critique doit éviter la réflexion. Voir https://msdn.microsoft.com/en-us/library/ff647790.aspx

Remarques

Reflection permet au code d'accéder aux informations sur les assemblys, les modules et les types au moment de l'exécution (exécution du programme). Cela peut ensuite être utilisé pour créer, modifier ou accéder de manière dynamique aux types. Les types incluent les propriétés, les méthodes, les champs et les attributs.

Lectures complémentaires:

Réflexion (C #)

Réflexion dans le cadre .Net

Examples

Obtenir un System.Type

Pour une instance d'un type:

```
var theString = "hello";
var theType = theString.GetType();
```

Du type lui-même:

```
var theType = typeof(string);
```

Obtenir les membres d'un type

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
```

```
  public static void Main()
  {
    var members = typeof(object)
                    .GetMembers(BindingFlags.Public |
                                BindingFlags.Static |
                                BindingFlags.Instance);

    foreach (var member in members)
    {
      bool inherited = member.DeclaringType.Equals( typeof(object).Name );
      Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                        $"it has {(inherited ? "":"not")} been inherited.");
    }
  }
}
```

Sortie (*voir note sur l'ordre de sortie plus bas*):

```
GetType is a Method, it has not been inherited.
GetHashCode is a Method, it has not been inherited.
ToString is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
ReferenceEquals is a Method, it has not been inherited.
.ctor is a Constructor, it has not been inherited.
```

Nous pouvons également utiliser `GetMembers()` sans passer par `BindingFlags` . Cela retournera *tous* les membres publics de ce type spécifique.

Une chose à noter que `GetMembers` ne `GetMembers` pas les membres dans un ordre particulier, alors ne comptez jamais sur l'ordre que `GetMembers` vous renvoie.

Voir la démo

Obtenez une méthode et invoquez-la

Obtenir la méthode d'instance et l'invoquer

```
using System;

public class Program
{
    public static void Main()
    {
        var theString = "hello";
        var method = theString
                    .GetType()
                    .GetMethod("Substring",
                               new[] {typeof(int), typeof(int)}); //The types of the method
arguments
         var result = method.Invoke(theString, new object[] {0, 4});
         Console.WriteLine(result);
    }
}
```

Sortie:

enfer

Voir la démo

Obtenez la méthode statique et invoquez-la

D'un autre côté, si la méthode est statique, vous n'avez pas besoin d'une instance pour l'appeler.

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2});//Pass null as the first argument (no need
for an instance)
Console.WriteLine(result); //You'll get e^2
```

Sortie:

7.38905609893065

Voir la démo

Obtenir et définir les propriétés

Utilisation de base:

```
PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.setValue(myInstance, newValue);
```

La définition des propriétés implémentées automatiquement en lecture seule peut être effectuée via son champ de sauvegarde (dans le nom du champ de sauvegarde de .NET Framework, "k__BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

Attributs personnalisés

Rechercher des propriétés avec un attribut personnalisé - `MyAttribute`

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Rechercher tous les attributs personnalisés sur une propriété donnée

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Énumérez toutes les classes avec un attribut personnalisé - `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
    }
}
```

Valeur de lecture d'un attribut personnalisé à l'exécution

```
public static class AttributeExtensions
{

        /// <summary>
        /// Returns the value of a member attribute for any member in a class.
        ///     (a member is a Field, Property, Method, etc...)
        /// <remarks>
        /// If there is more than one member of the same name in the class, it will return the
first one (this applies to overloaded methods)
        /// </remarks>
        /// <example>
        /// Read System.ComponentModel Description Attribute from method 'MyMethodName' in
class 'MyClass':
        ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName",
(DescriptionAttribute d) => d.Description);
        /// </example>
        /// <param name="type">The class that contains the member as a type</param>
        /// <param name="MemberName">Name of the member in the class</param>
        /// <param name="valueSelector">Attribute type and property to get (will return first
instance if there are multiple attributes of the same type)</param>
        /// <param name="inherit">true to search this member's inheritance chain to find the
attributes; otherwise, false. This parameter is ignored for properties and events</param>
        /// </summary>
        public static TValue GetAttribute<TAttribute, TValue>(this Type type, string
MemberName, Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute :
Attribute
        {
            var att =
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),
inherit).FirstOrDefault() as TAttribute;
            if (att != null)
            {
                return valueSelector(att);
            }
            return default(TValue);
        }
    }
```

Usage

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
```

```
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Faire le tour de toutes les propriétés d'une classe

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

Détermination des arguments génériques des instances de types génériques

Si vous avez une instance d'un type générique mais que, pour une raison quelconque, vous ne connaissez pas le type spécifique, vous souhaiterez peut-être déterminer les arguments génériques utilisés pour créer cette instance.

Disons que quelqu'un a créé une instance de `List<T>` comme ça et la passe à une méthode:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

où `ShowGenericArguments` a cette signature:

```
public void ShowGenericArguments(object o)
```

au moment de la compilation, vous n'avez aucune idée des arguments génériques utilisés pour créer `o` . Reflection fournit de nombreuses méthodes pour inspecter les types génériques. Au début, nous pouvons déterminer si le type de `o` est un type générique du tout:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` renvoie `true` si le type est un type générique et `false` si ce n'est pas le cas.

Mais ce n'est pas tout ce que nous voulons savoir. `List<>` est lui-même un type générique. Mais nous voulons seulement examiner les instances de types *génériques construits* spécifiques. Un type générique construit est par exemple un `List<int>` qui possède un *argument de* type spécifique pour tous ses *paramètres* génériques.

La classe `Type` fournit deux propriétés supplémentaires, `IsConstructedGenericType` et

`IsGenericTypeDefinition` , pour distinguer ces types génériques construits des définitions de type génériques:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

Pour énumérer les arguments génériques d'une instance, nous pouvons utiliser la méthode `GetGenericArguments()` qui renvoie un tableau `Type` contenant les arguments de type générique:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

L'appel du dessus (`ShowGenericArguments(myList)`) aboutit à cette sortie:

```
Int32
```

Obtenez une méthode générique et invoquez-la

Disons que vous avez une classe avec des méthodes génériques. Et vous devez appeler ses fonctions avec réflexion.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Disons que nous voulons appeler la méthode GenericMethod avec la chaîne de type.

```
Sample sample = new Sample();//or you can get an instance via reflection

MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null);//Since there are no arguments, we are passing null
```

Pour la méthode statique, vous n'avez pas besoin d'instance. Par conséquent, le premier argument sera également nul.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Créer une instance d'un type générique et appeler sa méthode

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

Classes d'instanciation qui implémentent une interface (par exemple, activation de plug-in)

Si vous souhaitez que votre application prenne en charge un système de plug-in, par exemple pour charger des plug-ins à partir d'assemblages situés dans le dossier `plugins` :

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

Cette classe serait située dans une DLL séparée

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Le chargeur de plug-in de votre application trouvera les fichiers dll, récupérera tous les types dans les assemblys qui implémentent `IPlugin` et en créera des instances.

```
    public IEnumerable<IPlugin> InstantiatePlugins(string directory)
    {
        var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name =>
new FileInfo(name).FullName).ToArray();
        //load the assemblies into the current AppDomain, so we can instantiate the types
later
        foreach (var fileName in pluginAssemblyNames)
            AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
        var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
        var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
        var pluginTypes = typesInAssembly.Where(type => typeof
```

```
(IPlugin).IsAssignableFrom(type));
        return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
    }
```

Création d'une instance d'un type

Le moyen le plus simple est d'utiliser la classe `Activator` .

Cependant, même si les performances d' `Activator` ont été améliorées depuis .NET 3.5, l'utilisation d' `Activator.CreateInstance()` est parfois une mauvaise option en raison de performances (relativement) faibles: Test 1 , Test 2 , Test 3 ...

Avec la classe `Activator`

```
Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an
'int' compatible argument.
Console.WriteLine(result); //Output: 123
```

Vous pouvez passer un tableau d'objets à `Activator.CreateInstance` si vous avez plusieurs paramètres.

```
// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });

Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

Pour un type générique

La méthode `MakeGenericType` transforme un type générique ouvert (comme `List<>`) en un type concret (comme `List<string>`) en lui appliquant des arguments de type.

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

La syntaxe `List<>` n'est pas autorisée en dehors d'une expression de `typeof` .

Sans classe d' `Activator`

Utiliser un `new` mot-clé (fera pour les constructeurs sans paramètre)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Utiliser la méthode Invoke

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

Utiliser des arbres d'expression

Les arbres d'expression représentent du code dans une structure de données arborescente, où chaque nœud est une expression. Comme MSDN explique:

> L'expression est une séquence d'un ou plusieurs opérandes et de zéro ou plusieurs opérateurs pouvant être évalués en une seule valeur, objet, méthode ou espace de noms. Les expressions peuvent consister en une valeur littérale, une invocation de méthode, un opérateur et ses opérandes ou un nom simple. Les noms simples peuvent être le nom d'une variable, d'un membre de type, d'un paramètre de méthode, d'un espace de noms ou d'un type.

```
public class GenericFactory<TKey, TType>
    {
       private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; //
dictionary, that holds constructor functions.
       private object _locker = new object(); // object for locking dictionary, to guarantee
thread safety

        public GenericFactory()
        {
            _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
        }

        /// <summary>
        /// Find and register suitable constructor for type
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key">Key for this constructor</param>
        /// <param name="parameters">Parameters</param>
        public void Register(TKey key, params Type[] parameters)
        {
            ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] {
}); // Get the instance of ctor.
            if (ci == null)
                throw new InvalidOperationException(string.Format("Constructor for type '{0}'
```

```
was not found.", typeof(TType)));

            Func<object[], TType> ctor;

            lock (_locker)
            {
                if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor
already been registered
                {
                    var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
parameter Expression
                    var ctorParams = ci.GetParameters(); // get parameter info from
constructor

                    var argExpressions = new Expression[ctorParams.Length]; // array that will
contains parameter expessions
                    for (var i = 0; i < parameters.Length; i++)
                    {

                        var indexedAcccess = Expression.ArrayIndex(pExp,
Expression.Constant(i));

                        if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
parameter is a value type
                        {
                            var localVariable = Expression.Variable(parameters[i],
"localVariable"); // if so - we should create local variable that will store paraameter value

                            var block = Expression.Block(new[] { localVariable },
                                    Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
                                        Expression.Assign(localVariable,
Expression.Default(parameters[i])),
                                        Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
                                    ),
                                    localVariable
                                );

                            argExpressions[i] = block;

                        }
                        else
                            argExpressions[i] = Expression.Convert(indexedAcccess,
parameters[i]);
                    }
                    var newExpr = Expression.New(ci, argExpressions); // create expression
that represents call to specified ctor with the specified arguments.

                    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp
}).Compile() as Func<object[], TType>); // compile expression to create delegate, and add
fucntion to dictionary
                }
            }
        }

        /// <summary>
        /// Returns instance of registered type by key.
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key"></param>
```

```
        /// <param name="args"></param>
        /// <returns></returns>
        public TType Create(TKey key, params object[] args)
        {
            Func<object[], TType> foo;
            if (_registeredTypes.TryGetValue(key, out foo))
            {
                return (TType)foo(args);
            }

            throw new ArgumentException("No type registered for this key.");
        }
    }
```

Peut être utilisé comme ceci:

```
 public class TestClass
 {
        public TestClass(string parameter)
        {
            Console.Write(parameter);
        }
 }


public void TestMethod()
{
       var factory = new GenericFactory<string, TestClass>();
       factory.Register("key", typeof(string));
       TestClass newInstance = factory.Create("key", "testParameter");
}
```

Utilisation de FormatterServices.GetUninitializedObject

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

En cas d'utilisation de constructeurs `FormatterServices.GetUninitializedObject` et d'initialisateurs de champs, ils ne seront pas appelés. Il est destiné à être utilisé dans les sérialiseurs et les moteurs à distance

Obtenir un type par nom avec un espace de noms

Pour ce faire, vous avez besoin d'une référence à l'assembly qui contient le type. Si vous avez un autre type disponible que vous savez être dans le même assemblage que celui que vous voulez, vous pouvez le faire:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- où `typeName` est le nom du type que vous recherchez (y compris l'espace de noms) et `KnownType` est le type que vous connaissez dans le même assembly.

Moins efficace mais plus général est comme suit:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Notez la coche pour exclure les assemblys de noms de systèmes d'analyse afin d'accélérer la recherche. Si votre type peut effectivement être un type CLR, vous devrez supprimer ces deux lignes.

Si vous avez le nom de type entièrement assemblé, y compris l'assemblage, vous pouvez simplement l'obtenir avec

```
Type.GetType(fullyQualifiedName);
```

Obtenir un délégué fortement typé sur une méthode ou une propriété via la réflexion

Lorsque les performances `MethodInfo.Invoke` problème, l'appel d'une méthode via la réflexion (c'est-à-dire via la méthode `MethodInfo.Invoke`) n'est pas idéal. Cependant, il est relativement simple d'obtenir un délégué plus typé et plus performant à l'aide de la fonction `Delegate.CreateDelegate` . La pénalité de performance liée à l'utilisation de la réflexion ne survient que pendant le processus de création de délégué. Une fois le délégué créé, il y a peu ou pas de performance pour l'invoquer:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int,
int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

Cette technique peut également être étendue aux propriétés. Si nous avons une classe nommée `MyClass` avec une propriété `int` nommée `MyIntProperty` , le code pour obtenir un getter fortement typé serait (l'exemple suivant suppose que 'target' est une instance valide de `MyClass`):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

... et la même chose peut être faite pour le passeur:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Lire Réflexion en ligne: https://riptutorial.com/fr/csharp/topic/28/reflexion

Chapitre 137: Rendement

Introduction

Lorsque vous utilisez le mot-clé yield dans une instruction, vous indiquez que la méthode, l'opérateur ou le accesseur dans lequel il apparaît est un itérateur. L'utilisation de yield pour définir un itérateur supprime la nécessité d'une classe supplémentaire explicite (la classe qui contient l'état d'une énumération) lorsque vous implémentez les modèles IEnumerable et IEnumerator pour un type de collection personnalisé.

Syntaxe

- rendement de retour [TYPE]
- rupture de rendement

Remarques

Indiquer le mot-clé de `yield` dans une méthode avec le type de retour `IEnumerable` , `IEnumerable<T>` , `IEnumerator` ou `IEnumerator<T>` indique au compilateur de générer une implémentation du type de retour (`IEnumerable` ou `IEnumerator`) qui exécute la boucle méthode jusqu'à chaque "rendement" pour obtenir chaque résultat.

Le mot-clé `yield` est utile lorsque vous souhaitez renvoyer l'élément "suivant" d'une séquence théoriquement illimitée. Il serait donc impossible de calculer l'intégralité de la séquence ou de calculer la séquence complète des valeurs avant de renvoyer une pause indésirable pour l'utilisateur. .

`yield break` peut également être utilisée pour terminer la séquence à tout moment.

Comme le mot clé `yield` nécessite un type d'interface d'itérateur en tant que type de retour, tel que `IEnumerable<T>` , vous ne pouvez pas l'utiliser dans une méthode asynchrone, car cela retourne un objet `Task<IEnumerable<T>>` .

Lectures complémentaires

- https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

Examples

Utilisation simple

Le mot-clé `yield` est utilisé pour définir une fonction qui renvoie un `IEnumerable` ou un `IEnumerator` (ainsi que leurs variantes génériques dérivées) dont les valeurs sont générées paresseusement au fur et à mesure qu'un utilisateur effectue une itération sur la collection renvoyée. En savoir plus sur le but dans la section remarques .

L'exemple suivant contient une instruction return return qui se trouve dans une boucle `for` .

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Ensuite, vous pouvez l'appeler:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Sortie de console

```
4
5
6
...
14
```

Démo en direct sur .NET Fiddle

Chaque itération du corps de l'instruction `foreach` crée un appel à la fonction `Count` iterator. Chaque appel à la fonction itérateur passe à l'exécution suivante de l'instruction `yield return` , qui se produit lors de la prochaine itération de la boucle `for` .

Utilisation plus pertinente

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name
FROM Users"))
    {
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

Bien sûr, il existe d'autres moyens d'obtenir un `IEnumerable<User>` partir d'une base de données SQL - cela montre simplement que vous pouvez utiliser `yield` pour transformer en `IEnumerable<T>` tout ce qui a une sémantique "séquence d'éléments". .

Résiliation anticipée

Vous pouvez étendre les fonctionnalités des méthodes de `yield` existantes en transmettant une ou plusieurs valeurs ou éléments pouvant définir une condition de terminaison dans la fonction en appelant une `yield break` pour empêcher l'exécution de la boucle interne.

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

La méthode ci-dessus effectuerait une itération à partir d'une position de `start` donnée jusqu'à ce qu'une des valeurs `earlyTerminationSet` dans le `earlyTerminationSet` soit rencontrée.

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

Sortie:

1
2
3
4
5
6

Démo en direct sur .NET Fiddle

Vérification correcte des arguments

Une méthode itérateur n'est pas exécutée tant que la valeur de retour n'est pas énumérée. Il est donc avantageux d'affirmer des conditions préalables en dehors de l'itérateur.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));

    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

Code côté appelant (utilisation):

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

Sortie:

1
2
3
4
5
6
7
8
9
dix

Démo en direct sur .NET Fiddle

Lorsqu'une méthode utilise un `yield` pour générer un énumérable, le compilateur crée une machine d'état qui, une fois itérée, exécute du code jusqu'à un `yield` . Il retourne ensuite l'élément produit et enregistre son état.

Cela signifie que vous ne découvrirez pas d'arguments non valides (`null` passant, etc.) lorsque

vous appelez la méthode pour la première fois (car cela crée la machine d'état), uniquement lorsque vous essayez d'accéder au premier élément méthode est exécuté par la machine d'état). En l'encapsulant dans une méthode normale qui vérifie d'abord les arguments, vous pouvez les vérifier lorsque la méthode est appelée. Ceci est un exemple d'échec rapide.

Lorsque vous utilisez C # 7+, la fonction `CountCore` peut être facilement cachée dans la fonction `Count` tant que *fonction locale* . Voir exemple ici .

Renvoie un autre Enumerable dans une méthode renvoyant Enumerable

```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

Évaluation paresseuse

Ce n'est que lorsque l'instruction `foreach` passe à l'élément suivant que le bloc itérateur est évalué jusqu'à la déclaration de `yield` suivante.

Prenons l'exemple suivant:

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}
```

Cela va sortir:

Début de l'itération
Itérateur interne: 0
Intérieur de la salle: 0
Itérateur interne: 1
Inside foreach: 1
Itérateur intérieur: 2
Intérieur de la salle: 2

Voir la démo

En conséquence:

- "Lancement de l'itération" est imprimé en premier, même si la méthode de l'itérateur a été appelée avant l'impression par la ligne `Integers().Take(3);` ne démarre pas réellement l'itération (aucun appel à `IEnumerator.MoveNext()` n'a été effectué)
- Les lignes imprimant sur la console alternent entre celle qui se trouve dans la méthode de l'itérateur et celle de la méthode `foreach` , plutôt que celles de la méthode de l'itérateur évaluant d'abord
- Ce programme se termine en raison de la méthode `.Take()` , même si la méthode iterator a un `while true` dont elle ne sort jamais.

Essayez ... enfin

Si une méthode itérative a un rendement dans un `try...finally` , le `IEnumerator` retourné exécutera l'instruction `finally` lorsque `Dispose` sera appelée, tant que le point d'évaluation actuel se trouve dans le bloc `try` .

Vu la fonction:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

En appelant:

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
```

```
}
```

Ensuite, il imprime:

1

Voir la démo

En appelant:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Ensuite, il imprime:

1
2
Enfin exécuté

Voir la démo

Utiliser les rendements pour créer un IEnumerator lors de l'implémentation de IEnumerable

L'interface `IEnumerable<T>` possède une méthode unique, `GetEnumerator()` , qui renvoie un `IEnumerator<T>` .

Alors que le mot-clé `yield` peut être utilisé pour créer directement un `IEnumerable<T>` , il peut *également* être utilisé exactement de la même manière pour créer un `IEnumerator<T>` . La seule chose qui change est le type de retour de la méthode.

Cela peut être utile si nous voulons créer notre propre classe qui implémente `IEnumerable<T>` :

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }

    // This method returns an IEnumerator<T>, rather than an IEnumerable<T>
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
```

```
    {
        foreach(var item in _wrapped)
        {
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

(Notez que cet exemple particulier est simplement illustratif et pourrait être implémenté plus proprement avec une seule méthode itérateur renvoyant un `IEnumerable<T>` .)

Évaluation avide

Le mot-clé de `yield` permet une évaluation paresseuse de la collection. Le chargement forcé de toute la collection en mémoire est appelé **évaluation rapide** .

Le code suivant montre ceci:

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
...
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

L'appel de `ToList` , `ToDictionary` ou `ToArray` force l'évaluation immédiate de l'énumération, en récupérant tous les éléments dans une collection.

Exemple d'évaluation paresseuse: numéros de Fibonacci

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numberics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
```

```
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

Comment ça marche sous le capot (je recommande de décompiler le fichier .exe résultant dans l'outil IL Disaambler):

1. Le compilateur C # génère une classe implémentant `IEnumerable<BigInteger>` et `IEnumerator<BigInteger>` (`<Fibonacci>d__0` dans ildasm).
2. Cette classe implémente une machine à états. L'état se compose de la position actuelle dans la méthode et des valeurs des variables locales.
3. Le code le plus intéressant se trouve dans la méthode `bool IEnumerator.MoveNext()` . Fondamentalement, ce que `MoveNext()` fait:
    - Restaure l'état actuel. Les variables comme `prev` et `current` deviennent des champs dans notre classe (`<current>5__2` et `<prev>5__1` in ildasm). Dans notre méthode, nous avons deux positions (`<>1__state`): la première à l'accolade ouvrante, la deuxième à la `yield return` .
    - Exécute le code jusqu'au prochain `yield return` `yield break` ou `yield break` / `}` .
    - Pour les `yield return` valeur résultante est enregistrée, ainsi la propriété `Current` peut la renvoyer. `true` est retourné. À ce stade, l'état actuel est à nouveau enregistré pour la prochaine invocation `MoveNext` .
    - Pour la méthode `yield break` / `}` méthode renvoie simplement `false` ce qui signifie que l'itération est effectuée.

Notez également que ce nombre 10001 est long de 468 octets. La machine d'état enregistre uniquement `current` variables `current` et `prev` tant que champs. Alors que si vous souhaitez enregistrer tous les nombres dans la séquence du premier au dixième, la taille de la mémoire consommée sera supérieure à 4 mégaoctets. L'évaluation paresseuse, si elle est utilisée correctement, peut réduire l'encombrement de la mémoire dans certains cas.

La différence entre break et break

Utiliser la `yield break` par opposition à la `break` pourrait ne pas être aussi évidente qu'on pourrait le penser. Il y a beaucoup de mauvais exemples sur Internet où l'utilisation des deux est interchangeable et ne démontre pas vraiment la différence.

La partie déroutante est que les deux mots-clés (ou phrases-clés) n'ont de sens que dans les boucles (`foreach` , `while` ...) Alors, quand choisir l'un sur l'autre?

Il est important de réaliser qu'une fois que vous utilisez le mot-clé `yield` dans une méthode, vous transformez la méthode en itérateur . Le seul but de cette méthode est alors d'itérer sur une collection finie ou infinie et de produire (produire) ses éléments. Une fois l'objectif atteint, il n'y a aucune raison de continuer l'exécution de la méthode. Parfois, cela se produit naturellement avec le dernier crochet de fermeture de la méthode `}` . Mais parfois, vous voulez mettre fin à la méthode prématurément. Dans une méthode normale (non itérative), vous utiliseriez le mot-clé `return` . Mais vous ne pouvez pas utiliser `return` dans un itérateur, vous devez utiliser `yield break` . En d' autres termes, la `yield break` pour un itérateur est le même que le `return` d'une méthode normalisée. Alors que l'instruction `break` fait que terminer la boucle la plus proche.

Voyons quelques exemples:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
            break;
        }
```

```
        }
        // Execution continues
        yield return 10;
    }
```

Lire Rendement en ligne: https://riptutorial.com/fr/csharp/topic/61/rendement

Chapitre 138: Rendre un thread variable sûr

Examples

Contrôler l'accès à une variable dans une boucle Parallel.For

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

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500.  sum is: {0}", sum );
    }
}
```

Il ne suffit pas de faire `sum = sum + i` sans le verrou car l'opération read-modify-write n'est pas atomique. Un thread remplacera toute modification externe apportée à la `sum` qui se produit après avoir lu la valeur actuelle de `sum` , mais avant de stocker la valeur modifiée de `sum + i` en `sum` .

Lire Rendre un thread variable sûr en ligne: https://riptutorial.com/fr/csharp/topic/4140/rendre-un-thread-variable-sur

Chapitre 139: Requêtes LINQ

Introduction

LINQ est un acronyme qui signifie **L** anguage **IN** ITéS **Q** uery. C'est un concept qui intègre un langage de requête en offrant un modèle cohérent pour travailler avec des données sur différents types de sources de données et de formats. 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 ensembles de données ADO.NET, des collections .NET et tout autre format pour lequel un fournisseur LINQ est disponible.

Syntaxe

- Syntaxe de requête:
    - de <variable de plage> dans <collection>
    - [de <variable de plage> dans <collection>, ...]
    - <filtrer, joindre, grouper, agréger des opérateurs, ...> <expression lambda>
    - <opérateur select ou groupBy> <formuler le résultat>
- Syntaxe de la méthode:
    - Enumerable.Aggregate (func)
    - Enumerable.Aggregate (graine, func)
    - Enumerable.Aggregate (graine, func, resultSelector)
    - Enumerable.All (prédicat)
    - Enumerable.Any ()
    - Enumerable.Any (prédicat)
    - Enumerable.AsEnumerable ()
    - Enumerable.Average ()
    - Enumerable.Average (sélecteur)
    - Enumerable.Cast <Résultat> ()
    - Enumerable.Concat (second)
    - Enumerable.Contains (value)
    - Enumerable.Contains (valeur, comparateur)
    - Enumerable.Count ()
    - Enumerable.Count (prédicat)
    - Enumerable.DefaultIfEmpty ()
    - Enumerable.DefaultIfEmpty (defaultValue)
    - Enumerable.Distinct ()
    - Enumerable.Distinct (comparateur)
    - Enumerable.ElementAt (index)
    - Enumerable.ElementAtOrDefault (index)
    - Enumerable.Empty ()
    - Enumerable.Except (second)

- Enumerable.Except (second, comparateur)
- Enumerable.First ()
- Enumerable.First (prédicat)
- Enumerable.FirstOrDefault ()
- Enumerable.FirstOrDefault (prédicat)
- Enumerable.GroupBy (keySelector)
- Enumerable.GroupBy (keySelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector)
- Enumerable.GroupBy (keySelector, comparateur)
- Enumerable.GroupBy (keySelector, resultSelector, comparateur)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector, comparateur)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector, comparateur)
- Enumerable.Intersect (second)
- Enumerable.Intersect (second, comparateur)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, comparateur)
- Enumerable.Last ()
- Enumerable.Last (prédicat)
- Enumerable.LastOrDefault ()
- Enumerable.LastOrDefault (prédicat)
- Enumerable.LongCount ()
- Enumerable.LongCount (prédicat)
- Enumerable.Max ()
- Enumerable.Max (sélecteur)
- Enumerable.Min ()
- Enumerable.Min (sélecteur)
- Enumerable.OfType <TResult> ()
- Enumerable.OrderBy (keySelector)
- Enumerable.OrderBy (keySelector, comparateur)
- Enumerable.OrderByDescending (keySelector)
- Enumerable.OrderByDescending (keySelector, comparateur)
- Enumerable.Range (démarrer, compter)
- Enumerable.Repeat (element, count)
- Enumerable.Reverse ()
- Enumerable.Select (sélecteur)
- Enumerable.SelectMany (sélecteur)
- Enumerable.SelectMany (collectionSelector, resultSelector)
- Enumerable.SequenceEqual (second)
- Enumerable.SequenceEqual (second, comparateur)
- Enumerable.Single ()
- Enumerable.Single (prédicat)
- Enumerable.SingleOrDefault ()
- Enumerable.SingleOrDefault (prédicat)
- Enumerable.Skip (count)

- Enumerable.SkipWhile (prédicat)
- Enumerable.Sum ()
- Enumerable.Sum (sélecteur)
- Enumerable.Take (count)
- Enumerable.TakeWhile (prédicat)
- orderEnumerable.ThenBy (keySelector)
- orderEnumerable.ThenBy (keySelector, comparateur)
- orderEnumerable.ThenByDescending (keySelector)
- OrdreEnumerable.ThenByDescending (keySelector, comparateur)
- Enumerable.ToArray ()
- Enumerable.ToDictionary (keySelector)
- Enumerable.ToDictionary (keySelector, elementSelector)
- Enumerable.ToDictionary (keySelector, comparateur)
- Enumerable.ToDictionary (keySelector, elementSelector, comparateur)
- Enumerable.ToList ()
- Enumerable.ToLookup (keySelector)
- Enumerable.ToLookup (keySelector, elementSelector)
- Enumerable.ToLookup (keySelector, comparateur)
- Enumerable.ToLookup (keySelector, elementSelector, comparateur)
- Enumerable.Union (second)
- Enumerable.Union (second, comparateur)
- Enumerable.Where (prédicat)
- Enumerable.Zip (second, resultSelector)

Remarques

Pour utiliser les requêtes LINQ, vous devez importer `System.Linq` .

La syntaxe de la méthode est plus puissante et flexible, mais la syntaxe de la requête peut être plus simple et plus familière. Toutes les requêtes écrites dans la syntaxe Query sont traduites dans la syntaxe fonctionnelle par le compilateur. Les performances sont donc les mêmes.

Les objets de requête ne sont pas évalués tant qu'ils ne sont pas utilisés, ils peuvent donc être modifiés ou ajoutés sans pénalité de performance.

Examples

Où

Retourne un sous-ensemble d'éléments dont le prédicat spécifié est vrai pour eux.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Syntaxe de la méthode

```
// Select all trees with name of length 3
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Syntaxe de requête

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

Sélectionner - Éléments de transformation

Select vous permet d'appliquer une transformation à chaque élément de toute structure de données implémentant IEnumerable.

Obtenir le premier caractère de chaque chaîne dans la liste suivante:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Utiliser la syntaxe régulière (lambda)

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

Sortie:

O
B
B
E
H
M

Démo en direct sur .NET Fiddle

Utiliser la syntaxe de requête LINQ

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

Méthodes de chaînage

De nombreuses fonctions LINQ opèrent toutes deux sur un `IEnumerable<TSource>` et renvoient également un `IEnumerable<TResult>` . Les paramètres de type `TSource` et `TResult` peuvent ou non se référer au même type, selon la méthode en question et les fonctions qui lui sont transmises.

Quelques exemples de ceci sont

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

Alors que certains chaînages de méthodes peuvent nécessiter l'utilisation d'un ensemble complet avant de passer à un autre, LINQ tire parti de l' exécution différée en utilisant **MSDN de** rendement qui crée un Enumerable et un Enumerator en arrière-plan. Le processus de chaînage dans LINQ consiste essentiellement à créer un enumerable (itérateur) pour l'ensemble original - qui est différé - jusqu'à ce qu'il soit matérialisé en énumérant l'énumérable .

Cela permet à ces fonctions d'être couramment enchaînées sur **wiki** , où une fonction peut agir directement sur le résultat d'une autre. Ce style de code peut être utilisé pour effectuer de nombreuses opérations basées sur des séquences dans une seule instruction.

Par exemple, il est possible de combiner `Select` , `Where` et `OrderBy` pour transformer, filtrer et trier une séquence en une seule instruction.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
        .Select(n => n * 2)    // Multiply each number by 2
        .Where(n => n != 6)    // Keep all the results, except for 6
        .OrderBy(n => n);      // Sort in ascending order
```

Sortie:

2
4
8

Démo en direct sur .NET Fiddle

Toutes les fonctions qui étendent et retournent le type générique `IEnumerable<T>` peuvent être utilisées comme clauses chaînées dans une seule instruction. Ce style de programmation fluide est puissant et doit être pris en compte lors de la création de vos propres méthodes d'extension .

Portée et répétition

Les méthodes statiques `Range` et `Repeat` sur `Enumerable` peuvent être utilisées pour générer des

séquences simples.

Gamme

`Enumerable.Range()` génère une séquence d'entiers à partir d'une valeur de départ et d'un nombre.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Démo en direct sur .NET Fiddle

Répéter

`Enumerable.Repeat()` génère une séquence d'éléments répétés à partir d'un élément et du nombre de répétitions requis.

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

Démo en direct sur .NET Fiddle

Passer et prendre

La méthode Skip renvoie une collection à l'exclusion d'un certain nombre d'éléments du début de la collection source. Le nombre d'éléments exclus est le nombre donné en argument. Si la collection contient moins d'éléments que ceux spécifiés dans l'argument, une collection vide est renvoyée.

La méthode Take renvoie une collection contenant un certain nombre d'éléments du début de la collection source. Le nombre d'éléments inclus est le nombre donné en argument. Si la collection contient moins d'éléments que spécifié dans l'argument, la collection renvoyée contiendra les mêmes éléments que la collection source.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo        = values.Skip(2);         // { 3, 2, 1 }
var takeThree      = values.Take(3);         // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero       = values.Take(0);         // An IEnumerable<int> with 0 items
```

Démo en direct sur .NET Fiddle

Skip and Take sont couramment utilisés ensemble pour paginer les résultats, par exemple:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

> **Avertissement:** LINQ to Entities ne prend en charge que les requêtes de saut sur les commandes . Si vous essayez d'utiliser Skip sans passer commande, vous recevrez une **exception NotSupportedException** avec le message "La méthode 'Skip' est uniquement prise en charge pour les entrées triées dans LINQ to Entities. La méthode 'OrderBy' doit être appelée avant la méthode 'Skip'."

Tout d'abord, FirstOrDefault, Last, LastOrDefault, Single et SingleOrDefault

Les six méthodes renvoient une seule valeur du type de séquence et peuvent être appelées avec ou sans prédicat.

Selon le nombre d'éléments correspondant au `predicate` ou, si aucun `predicate` n'est fourni, le nombre d'éléments de la séquence source, ils se comportent comme suit:

Premier()

- Renvoie le premier élément d'une séquence ou le premier élément correspondant au `predicate` fourni.
- Si la séquence ne contient aucun élément, une `InvalidOperationException` est lancée avec le message: "La séquence ne contient aucun élément".
- Si la séquence ne contient aucun élément correspondant au `predicate` fourni, une `InvalidOperationException` est `InvalidOperationException` avec le message "La séquence ne contient aucun élément correspondant".

Exemple

```
// Returns "a":
new[] { "a" }.First();

// Returns "a":
new[] { "a", "b" }.First();

// Returns "b":
new[] { "a", "b" }.First(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].First();
```

Démo en direct sur .NET Fiddle

FirstOrDefault ()

- Renvoie le premier élément d'une séquence ou le premier élément correspondant au

`predicate` fourni.
- Si la séquence ne contient aucun élément ou aucun élément correspondant au `predicate` fourni, retourne la valeur par défaut du type de séquence en utilisant la `default(T)` par `default(T)` .

Exemple

```
// Returns "a":
new[] { "a" }.FirstOrDefault();

// Returns "a":
new[] { "a", "b" }.FirstOrDefault();

// Returns "b":
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].FirstOrDefault();
```

Démo en direct sur .NET Fiddle

Dernier()

- Renvoie le dernier élément d'une séquence ou le dernier élément correspondant au `predicate` fourni.
- Si la séquence ne contient aucun élément, une `InvalidOperationException` est lancée avec le message "La séquence ne contient aucun élément".
- Si la séquence ne contient aucun élément correspondant au `predicate` fourni, une `InvalidOperationException` est `InvalidOperationException` avec le message "La séquence ne contient aucun élément correspondant".

Exemple

```
// Returns "a":
new[] { "a" }.Last();

// Returns "b":
new[] { "a", "b" }.Last();

// Returns "a":
new[] { "a", "b" }.Last(x => x.Equals("a"));

// Returns "be":
new[] { "ba", "be" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.Last(x => x.Contains("b"));
```

```
// Throws InvalidOperationException:
new string[0].Last();
```

LastOrDefault ()

- Renvoie le dernier élément d'une séquence ou le dernier élément correspondant au `predicate` fourni.
- Si la séquence ne contient aucun élément ou aucun élément correspondant au `predicate` fourni, retourne la valeur par défaut du type de séquence en utilisant la `default(T)` par `default(T)` .

Exemple

```
// Returns "a":
new[] { "a" }.LastOrDefault();

// Returns "b":
new[] { "a", "b" }.LastOrDefault();

// Returns "a":
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));

 // Returns "be":
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].LastOrDefault();
```

Unique()

- Si la séquence contient exactement un élément ou exactement un élément correspondant au `predicate` fourni, cet élément est renvoyé.
- Si la séquence ne contient aucun élément ou aucun élément correspondant au `predicate` fourni, une `InvalidOperationException` est lancée avec le message "La séquence ne contient aucun élément".
- Si la séquence contient plusieurs éléments ou plusieurs éléments correspondant au `predicate` fourni, une `InvalidOperationException` est lancée avec le message "La séquence contient plusieurs éléments".
- **Remarque:** pour évaluer si la séquence contient exactement un élément, deux éléments au plus doivent être énumérés.

Exemple

```
// Returns "a":
new[] { "a" }.Single();
```

```
// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();
```

SingleOrDefault ()

- Si la séquence contient exactement un élément ou exactement un élément correspondant au `predicate` fourni, cet élément est renvoyé.
- Si la séquence ne contient aucun élément ou aucun élément correspondant au `predicate` fourni, la `default(T)` est renvoyée.
- Si la séquence contient plusieurs éléments ou plusieurs éléments correspondant au `predicate` fourni, une `InvalidOperationException` est lancée avec le message "La séquence contient plusieurs éléments".
- Si la séquence ne contient aucun élément correspondant au `predicate` fourni, retourne la valeur par défaut du type de séquence en utilisant la `default(T)` par `default(T)` .
- **Remarque:** pour évaluer si la séquence contient exactement un élément, deux éléments au plus doivent être énumérés.

Exemple

```
// Returns "a":
new[] { "a" }.SingleOrDefault();

// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();

// Returns null:
new string[0].SingleOrDefault();
```

Recommandations

- Bien que vous puissiez utiliser `FirstOrDefault` , `LastOrDefault` ou `SingleOrDefault` pour vérifier si une séquence contient des éléments, `Any` ou `Count` sont plus fiables. En effet, une valeur de retour `default(T)` de l'une de ces trois méthodes ne prouve pas que la séquence est vide, car la valeur du premier / dernier / seul élément de la séquence pourrait également être `default(T)`

- Décidez des méthodes qui correspondent le mieux au but de votre code. Par exemple, utilisez `Single` uniquement si vous devez vous assurer qu'un seul élément de la collection correspond à votre prédicat - sinon utilisez `First` ; comme `Single` lance une exception si la séquence a plus d'un élément correspondant. Ceci s'applique bien entendu aux partenaires "* OrDefault".

- En ce qui concerne l'efficacité: Bien qu'il soit souvent approprié de s'assurer qu'il n'ya qu'un seul élément (`Single`) ou qu'un seul élément ou un seul `SingleOrDefault` (`SingleOrDefault`) renvoyé par une requête, ces deux méthodes requièrent plus, et souvent la totalité de la collection. à examiner pour s'assurer qu'il n'y a pas de seconde correspondance à la requête. Ceci est différent du comportement, par exemple, de la méthode `First` , qui peut être satisfaite après avoir trouvé la première correspondance.

Sauf

La méthode Except renvoie l'ensemble des éléments contenus dans la première collection mais ne sont pas contenus dans la seconde. `IEqualityComparer` par défaut est utilisé pour comparer les éléments dans les deux ensembles. Il y a une surcharge qui accepte un `IEqualityComparer` comme argument.

Exemple:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Sortie:

1
4

Démo en direct sur .NET Fiddle

Dans ce cas, `.Except(second)` exclut les éléments contenus dans le tableau `second` , à savoir 2 et 3 (0 et 5 ne sont pas contenus dans le `first` tableau et sont ignorés).

Notez que `Except` implique `Distinct` (c.-à-d. Qu'il supprime les éléments répétés). Par exemple:

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

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

Sortie:

1
4

Démo en direct sur .NET Fiddle

Dans ce cas, les éléments 1 et 4 ne sont renvoyés qu'une seule fois.

Implémenter `IEquatable` ou fournir la fonction `IEqualityComparer` permettra d'utiliser une méthode différente pour comparer les éléments. Notez que la méthode `GetHashCode` doit également être remplacée afin de renvoyer un code de hachage identique pour un `object` identique selon l'implémentation `IEquatable` .

Exemple avec IEquatable:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
```

```
    }
}
```

Sortie:

    Hanoukka

Démo en direct sur .NET Fiddle

SelectMany: Aplatir une séquence de séquences

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

Utilisez `SelectMany()` si vous en avez, ou vous créez une séquence de séquences, mais vous voulez que le résultat soit une longue séquence.

Dans la syntaxe de requête LINQ:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

Si vous avez une collection de collections et que vous souhaitez pouvoir travailler sur les données de la collection parent et enfant en même temps, cela est également possible avec `SelectMany` .

Définissons des classes simples

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Supposons que nous avons la collection suivante.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
```

```
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        }
    },
    new BlogPost()
    {
        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    }
};
```

Nous souhaitons maintenant sélectionner les commentaires `Content` avec l' `Id` de `BlogPost` associé à ce commentaire. Pour ce faire, nous pouvons utiliser la surcharge `SelectMany` appropriée.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId =
post.Id, CommentContent = comment.Content });
```

Nos `commentsWithIds` ressemble à ceci

```
{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}
```

SelectMany

La méthode linq SelectMany «aplatit» un `IEnumerable<IEnumerable<T>>` en un `IEnumerable<T>` . Tous

les éléments T des instances `IEnumerable` contenus dans la source `IEnumerable` seront combinés en un seul `IEnumerable` .

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

Si vous utilisez une fonction de sélecteur qui transforme les éléments d'entrée en séquences, le résultat sera les éléments de ces séquences renvoyés un par un.

Notez que, contrairement à `Select()` , le nombre d'éléments dans la sortie ne doit pas nécessairement être identique à celui de l'entrée.

Plus exemple du monde réel

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"}
}},
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} }}
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

Sortie:

Bob
Jack
Jim
John

Démo en direct sur .NET Fiddle

Tout

`All` est utilisé pour vérifier si tous les éléments d'une collection correspondent à une condition ou non.
voir aussi: .Tout

1. paramètre vide

Tout : ne peut pas être utilisé avec un paramètre vide.

2. Expression Lambda en tant que paramètre

All : Renvoie `true` si tous les éléments de la collection satisfont l'expression lambda et `false` sinon:

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.All(i => i < 10); // true
bool result = numbers.All(i => i >= 3); // false
```

3. Collection vide

All : Renvoie `true` si la collection est vide et qu'une expression lambda est fournie:

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

Remarque: `All` arrêteront l'itération de la collection dès qu'elle trouvera un élément **ne** correspondant **pas à** la condition. Cela signifie que la collection ne sera pas nécessairement entièrement énumérée; il sera seulement énuméré assez loin pour trouver le premier élément **ne correspondant pas à** la condition.

Collection de requêtes par type / éléments de distribution à taper

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Utiliser `OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumerable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumerable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumerable<IFoo> with all four items
```

En utilisant `Where`

```
var foos = collection.Where(item => item is Foo); // result: IEnumerable<IFoo> with item0 and
item1
var bars = collection.Where(item => item is Bar); // result: IEnumerable<IFoo> with item2 and
```

```
item3
```

En utilisant la `Cast`

```
var bars = collection.Cast<Bar>();                // throws InvalidCastException on the 1st
item
var foos = collection.Cast<Foo>();                // throws InvalidCastException on the 3rd
item
var foosAndBars = collection.Cast<IFoo>();        // OK
```

syndicat

Fusionne deux collections pour créer une collection distincte à l'aide du comparateur d'égalité par défaut

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

Démo en direct sur .NET Fiddle

JOINT

Les jointures permettent de combiner différentes listes ou tables contenant des données via une clé commune.

Comme dans SQL, les types de jointures suivants sont pris en charge dans LINQ:
Jointures internes **, gauche, droite, croisées** et **complètes** .

Les deux listes suivantes sont utilisées dans les exemples ci-dessous:

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

(Jointure interne

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                        f => f,
                        s => s,
                        (f, s) => new { f, s });

// Result: {"a","a"}
//         {"c","c"}
```

Jointure externe gauche

```
var leftOuterJoin = from f in first
                    join s in second on f equals s into temp
                    from t in temp.DefaultIfEmpty()
                    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                    from s in second.Where(x => x == f).DefaultIfEmpty()
                    select new { First = f, Second = s};

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}


// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                      f => f,
                                      s => s,
                                      (f, s) => new { First = f, Second = s })
                                   .SelectMany(temp => temp.Second.DefaultIfEmpty(),
                                      (f, s) => new { First = f.First, Second = s });
```

Right Outer Join

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};

// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

Cross Join

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

Full Outer Join

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
//         {null,"d"}
```

Exemple pratique

Les exemples ci-dessus ont une structure de données simple afin que vous puissiez vous concentrer sur la compréhension technique des différentes jointures LINQ, mais dans le monde réel, vous aurez des tables avec des colonnes à joindre.

Dans l'exemple suivant, il n'y a qu'une classe `Region` utilisée. En réalité, vous rejoindrez plusieurs tables différentes contenant la même clé (dans cet exemple, la `first` et la `second` sont jointes via l' `ID` clé commune).

Exemple: considérez la structure de données suivante:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Maintenant, préparez les données (c.-à-d. Remplissez les données):

```
// Left data
var first = new List<Region>()
                { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
                {
                   new Region(1, "Eastern"),  new Region(2, "Western"),
                   new Region(3, "Northern"), new Region(4, "Southern")
                };
```

Vous pouvez voir que dans cet exemple, la `first` ne contient aucune description de région et que vous souhaitez donc les rejoindre depuis la `second` . Alors la jointure interne ressemblerait à:

```
// do the inner join
var result = from f in first
             join s in second on f.ID equals s.ID
             select new { f.ID, s.RegionDescription };


 // Result: {1,"Eastern"}
 //         {3, Northern}
```

```
//         {4,"Southern"}
```

Ce résultat a créé des objets anonymes à la volée, ce qui est bien, mais nous avons déjà créé une classe appropriée - nous pouvons donc le spécifier: au lieu de `select new { f.ID, s.RegionDescription };` on peut dire `select new Region(f.ID, s.RegionDescription);` , qui renverra les mêmes données mais créera des objets de type `Region` - qui maintiendront la compatibilité avec les autres objets.

Démo en direct sur le violon .NET

Distinct

Renvoie les valeurs uniques d'un `IEnumerable` . L'unicité est déterminée à l'aide du comparateur d'égalité par défaut.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

Pour comparer un type de données personnalisé, nous devons implémenter l' `IEquatable<T>` et fournir des méthodes `GetHashCode` et `Equals` pour le type. Ou le comparateur d'égalité peut être remplacé:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

GroupBy un ou plusieurs champs

Supposons que nous avons un modèle de film:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Propriété Group by Category:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
}
```

Grouper par catégorie et année:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

Utiliser Range avec différentes méthodes Linq

Vous pouvez utiliser la classe Enumerable aux côtés des requêtes Linq pour convertir les boucles en lignes Linq one.

Sélectionnez un exemple

Opposé à faire ceci:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

Tu peux le faire:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

Où exemple

Dans cet exemple, 100 numéros seront générés et même ceux qui seront extraits

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Ordre des requêtes - OrderBy () ThenBy () OrderByDescending () ThenByDescending ()

```
string[] names= { "mark", "steve", "adam" };
```

Ascendant:

Syntaxe de requête

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Syntaxe de la méthode

```
var sortedNames = names.OrderBy(name => name);
```

namedNames contient les noms dans l'ordre suivant: "adam", "mark", "steve"

Descendant:

Syntaxe de requête

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

Syntaxe de la méthode

```
var sortedNames = names.OrderByDescending(name => name);
```

namedNames contient les noms dans l'ordre suivant: "steve", "mark", "adam"

Commander par plusieurs champs

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil" , LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 15}
};
```

Syntaxe de requête

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

Syntaxe de la méthode

```
 sortedPeople = people.OrderBy(person => person.LastName)
                      .ThenBy(person => person.FirstName)
                      .ThenByDescending(person => person.Age);
```

Résultat

```
1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins  28
4. Steve Collins 30
```

Les bases

LINQ est largement bénéfique pour interroger des collections (ou des tableaux).

Par exemple, compte tenu des exemples de données suivants:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true  },
    new Student { Name = "Bob",   Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true  },
    new Student { Name = "Greg",  Grade = 90, HasSnack = false },
    new Student { Name = "Joe",   Grade = 59, HasSnack = false }
}
```

Nous pouvons "interroger" ces données en utilisant la syntaxe LINQ. Par exemple, pour récupérer tous les élèves qui ont une collation aujourd'hui:

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

Ou, pour récupérer les élèves ayant une note de 90 ou plus et ne renvoyer que leurs noms, pas l'objet complet de l' `Student` :

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

La fonctionnalité LINQ est composée de deux syntaxes qui remplissent les mêmes fonctions, ont des performances presque identiques, mais sont écrites de manière très différente. La syntaxe dans l'exemple ci-dessus est appelée **syntaxe de requête** . L'exemple suivant illustre toutefois la **syntaxe de la méthode** . Les mêmes données seront renvoyées comme dans l'exemple ci-dessus, mais la façon dont la requête est écrite est différente.

```
var topStudentNames = classroom.Students
                                .Where(s => s.Grade >= 90)
                                .Select(s => s.Name);
```

Par groupe

GroupBy est un moyen simple de trier une collection d'éléments `IEnumerable<T>` en groupes distincts.

Exemple simple

Dans ce premier exemple, nous nous retrouvons avec deux groupes, des articles pairs et impairs.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items

foreach(var group in grouped)
{
    foreach (int item in group)
    {
```

```
        Console.Write(item); // 135792468  (first odd then even)
    }
}
```

Exemple plus complexe

Prenons comme exemple le regroupement d'une liste de personnes par âge. Tout d'abord, nous allons créer un objet Person qui possède deux propriétés, Name et Age.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Ensuite, nous créons notre liste de personnes avec des noms et des âges différents.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Ensuite, nous créons une requête LINQ pour regrouper notre liste de personnes par âge.

```
var query = people.GroupBy(x => x.Age);
```

En faisant cela, nous pouvons voir l'âge pour chaque groupe et avoir une liste de chaque personne du groupe.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

Cela se traduit par la sortie suivante:

```
20
Mouse
30
Neo
Trinity
40
Morpheus
Dozer
Smith
```

Vous pouvez jouer avec la démo en direct sur .NET Fiddle

Tout

`Any` est utilisé pour vérifier si **un** élément d'une collection correspond à une condition ou non.
voir aussi: .All , Any et FirstOrDefault: meilleures pratiques

1. paramètre vide

Any : Renvoie `true` si la collection contient des éléments et `false` si la collection est vide:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

2. Expression Lambda en tant que paramètre

Any : Renvoie `true` si la collection comporte un ou plusieurs éléments répondant à la condition de l'expression lambda:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

3. Collection vide

Any : renvoie `false` si la collection est vide et qu'une expression lambda est fournie:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Remarque: `Any` arrête l'itération de la collection dès qu'il détecte un élément correspondant à la condition. Cela signifie que la collection ne sera pas nécessairement entièrement énumérée; il sera seulement énuméré assez loin pour trouver le premier article correspondant à la condition.

Démo en direct sur .NET Fiddle

ToDictionary

La `ToDictionary()` LINQ peut être utilisée pour générer une collection `Dictionary<TKey, TElement>` basée sur une source `IEnumerable<T>` donnée.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

Dans cet exemple, le seul argument transmis à `ToDictionary` est de type `Func<TSource, TKey>` , qui renvoie la clé pour chaque élément.

C'est un moyen concis d'effectuer l'opération suivante:

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

Vous pouvez également transmettre un second paramètre à la méthode `ToDictionary` , qui est de type `Func<TSource, TElement>` et renvoie la `Value` à ajouter pour chaque entrée.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

Il est également possible de spécifier l' `IComparer` utilisé pour comparer les valeurs clés. Cela peut être utile lorsque la clé est une chaîne et que vous souhaitez qu'elle corresponde à la casse.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsenstiveName = users.ToDictionary(x => x.Name,
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsenstiveName["john"];
var user2 = usersByCaseInsenstiveName["JOHN"];
user1 == user2; // Returns true
```

Remarque: la méthode `ToDictionary` nécessite que toutes les clés soient uniques, il ne doit y avoir aucune clé en double. Si tel est le cas, une exception est levée: `ArgumentException: An item with the same key has already been added.` Si vous avez un scénario où vous savez que vous aurez plusieurs éléments avec la même clé, il vaut mieux utiliser `ToLookup` place.

Agrégat

`Aggregate` Applique une fonction d'accumulateur sur une séquence.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- Au premier pas `prevSum = 1`
- Au second `prevSum = prevSum(at the first step) + 2`
- A la iième étape `prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);
// joinedString = "Hello World !"
```

Une deuxième surcharge de l' `Aggregate` reçoit également un paramètre d' `seed` qui correspond à la

valeur initiale de l'accumulateur. Cela peut être utilisé pour calculer plusieurs conditions sur une collection sans l'itérer plus d'une fois.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Pour la collection d' `items` nous voulons calculer

1. Le total `.Count`
2. La quantité de nombres pairs
3. Ramassez chaque article

En utilisant `Aggregate` cela peut se faire comme ceci:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },
                (accumelative,item) =>
                new {
                    Total = accumelative.Total + 1,
                    Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),
                    FourthItems = (accumelative.Total + 1)%4 == 0 ?
                        new List<int>(accumelative.FourthItems) { item } :
                        accumelative.FourthItems
                });
// Result:
// Total = 12
// Even = 6
// FourthItems = [4, 8, 12]
```

Notez que si vous utilisez un type anonyme comme graine, vous devez instancier un nouvel objet à chaque élément car les propriétés sont en lecture seule. En *utilisant une classe personnalisée peut simplement affecter les informations et aucune* `new` *est nécessaire (seulement en donnant la première* `seed` *paramètre*

Définir une variable dans une requête Linq (mot clé let)

Afin de définir une variable dans une expression linq, vous pouvez utiliser le mot **-** clé **let** . Ceci est généralement fait pour stocker les résultats des sous-requêtes intermédiaires, par exemple:

```
 int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

 var aboveAverages = from number in numbers
                     let average = numbers.Average()
                     let nSquared = Math.Pow(number,2)
                     where nSquared > average
                     select number;

 Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

 foreach (int n in aboveAverages)
 {
   Console.WriteLine("Query result includes number {0} with square of {1}.", n,
Math.Pow(n,2));
 }
```

Sortie:

La moyenne des nombres est de 4,5.
Le résultat de la requête inclut le numéro 3 avec un carré de 9.
Le résultat de la requête inclut le numéro 4 avec un carré de 16.
Le résultat de la requête inclut le numéro 5 avec un carré de 25.
Le résultat de la requête inclut le numéro 6 avec un carré de 36.
Le résultat de la requête inclut le numéro 7 avec un carré de 49.
Le résultat de la requête inclut le numéro 8 avec un carré de 64.
Le résultat de la requête inclut le numéro 9 avec un carré de 81.

Voir la démo

SkipWhile

`SkipWhile()` est utilisé pour exclure des éléments jusqu'à la première non-correspondance (cela peut être contre-intuitif pour la plupart)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

DefaultIfEmpty

DefaultIfEmpty est utilisé pour renvoyer un élément par défaut si la séquence ne contient aucun élément. Cet élément peut être la valeur par défaut du type ou une instance définie par l'utilisateur de ce type. Exemple:

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
     .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
        .DefaultIfEmpty().First(); // returns null;
```

Utilisation dans les jointures à gauche :

Avec `DefaultIfEmpty` la jointure Linq traditionnelle peut renvoyer un objet par défaut si aucune correspondance n'a été trouvée. Agissant ainsi comme une jointure de gauche de SQL. Exemple:

```
var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
```

```
                      select new
                      {
                          Number = l,
                          Character = result
                      };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

ouput:

Num = 99         Char = c
Num = 100        Char = d
Num = 5          Char = ?
Num = 20         Char = ?
Num = 102        Char = ?
Num = 105        Char = i
```

Dans le cas où un `DefaultIfEmpty` est utilisé (sans spécifier de valeur par défaut) et qu'il en résultera aucun élément correspondant sur la bonne séquence, il faut s'assurer que l'objet n'est pas `null` avant d'accéder à ses propriétés. Sinon, cela entraînera une `NullReferenceException` . Exemple:

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
    {
        new { Value = 1 },
        new { Value = 2 },
        new { Value = 3 },
        new { Value = 4 },
    };

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
                      {
                          Left = l,
                          // 5 will not have a matching object in the right so result
                          // will be equal to null.
                          // To avoid an error use:
                          //    -  C# 6.0 or above - ?.
                          //    -  Under           - result == null ? 0 : result.Value
                          Right = result?.Value
                      }).ToList();
```

SéquenceEqual

`SequenceEqual` est utilisé pour comparer deux séquences `IEnumerable<T>` entre elles.

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

```
bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Count et LongCount

`Count` renvoie le nombre d'éléments dans un `IEnumerable<T>` . `Count` expose également un paramètre de prédicat facultatif qui vous permet de filtrer les éléments à compter.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount` fonctionne de la même manière que `Count` mais a un type de retour `long` et est utilisé pour compter les séquences `IEnumerable<T>` plus longues que `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater
than 100
```

Créer progressivement une requête

Comme LINQ utilise une **exécution différée** , nous pouvons avoir un objet de requête qui ne contient pas réellement les valeurs, mais qui renverra les valeurs lors de l'évaluation. Nous pouvons donc construire dynamiquement la requête en fonction de notre flux de contrôle et l'évaluer une fois que nous avons terminé:

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int
count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
```

Nous pouvons appliquer des filtres conditionnellement:

```
    if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
        query = query.Where(v => search.Years.Contains(v.Year));

    if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Makes.Contains(v.Make));
    }

    if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
```

```
        query = query.Where(v => search.Models.Contains(v.Model));
    }

    if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
        decimal minCylinders = 0;
        decimal maxCylinders = 0;
        switch (search.Cylinders) {
            case "2-4":
                maxCylinders = 4;
                break;
            case "5-6":
                minCylinders = 5;
                maxCylinders = 6;
                break;
            case "8":
                minCylinders = 8;
                maxCylinders = 8;
                break;
            case "10+":
                minCylinders = 10;
                break;
        }
        if (minCylinders > 0) {
            query = query.Where(v => v.Cylinders >= minCylinders);
        }
        if (maxCylinders > 0) {
            query = query.Where(v => v.Cylinders <= maxCylinders);
        }
    }
```

Nous pouvons ajouter un ordre de tri à la requête en fonction d'une condition:

```
    switch (search.SortingColumn.ToLower()) {
        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
```

Notre requête peut être définie pour commencer à partir d'un point donné:

```
    query = query.Skip(start - 1);
```

et défini pour renvoyer un nombre spécifique d'enregistrements:

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
}
```

Une fois que nous avons l'objet de requête, nous pouvons évaluer les résultats avec une boucle `foreach` ou une des méthodes LINQ qui renvoie un ensemble de valeurs, telles que `ToList` ou `ToArray` :

```
SearchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

Zip *: français

La méthode d'extension `Zip` agit sur deux collections. Il associe chaque élément des deux séries en fonction de leur position. Avec une instance `Func` , nous utilisons `Zip` pour gérer les éléments des deux collections C # par paires. Si la taille de la série diffère, les éléments supplémentaires de la plus grande série seront ignorés.

Pour prendre un exemple du livre "C # in a Nutshell",

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

Sortie:

3 = trois
5 = cinq
7 = sept

Voir la démo

GroupJoin avec variable de plage externe

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };
```

ElementAt et ElementAtOrDefault

`ElementAt` retournera l'élément à l'index `n` . Si `n` n'est pas dans la plage de l'énumérable, lève une `ArgumentOutOfRangeException` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` renverra l'élément à l'index `n` . Si `n` n'est pas dans la plage de l'énumérable, retourne une `default(T)` par `default(T)` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

`ElementAt` et `ElementAtOrDefault` sont tous deux optimisés lorsque la source est un `IList<T>` et que l'indexation normale sera utilisée dans ces cas.

Notez que pour `ElementAt` , si l'index fourni est supérieur à la taille de `IList<T>` , la liste doit (mais n'est techniquement pas garantie) lancer une `ArgumentOutOfRangeException` .

Linq Quantifiers

Les opérations quantificateur renvoient une valeur booléenne si certains ou tous les éléments d'une séquence satisfont à une condition. Dans cet article, nous verrons quelques scénarios LINQ to Objects courants dans lesquels nous pouvons utiliser ces opérateurs. Il y a 3 opérations de quantificateurs qui peuvent être utilisées dans LINQ:

`All` - utilisé pour déterminer si tous les éléments d'une séquence satisfont à une condition. Par exemple:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

`Any` - utilisé pour déterminer si des éléments d'une séquence satisfont à une condition. Par exemple:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` - utilisé pour déterminer si une séquence contient un élément spécifié. Par exemple:

```
//for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
```

```
string[] query={"Tom","grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
stringValue.Contains("h");
```

Joindre plusieurs séquences

Considérez les entités `Customer` , `Purchase` et `PurchaseItem` comme suit:

```
public class Customer
{
   public string Id { get; set } // A unique Id that identifies customer
   public string Name  {get; set; }
}

public class Purchase
{
   public string Id { get; set }
   public string CustomerId {get; set; }
   public string Description { get; set; }
}

public class PurchaseItem
{
   public string Id { get; set }
   public string PurchaseId {get; set; }
   public string Detail { get; set; }
}
```

Envisagez de suivre les exemples de données pour les entités ci-dessus:

```
var customers = new List<Customer>()
 {
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },

    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
 };

 var purchases = new List<Purchase>()
 {
     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase2"
```

```
    },

    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase1"
    },

    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
 };

 var purchaseItems = new List<PurchaseItem>()
 {
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
 };
```

Maintenant, considérez ci-dessous la requête linq:

```
var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId     // second join
             select new
             {
                c.Name, p.Description, pi.Detail
             };
```

Pour afficher le résultat de la requête ci-dessus:

```
foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

La sortie de la requête serait:

Client1, Client1-Achat1, Achat1-AchatItem1

Client1, Client1-Achat2, Achat2-AchatItem1

Client1, Client1-Achat2, Achat2-AchatItem2

Client2, Client2-Achat2, Achat3-AchatItem1

Démo en direct sur .NET Fiddle

Se joindre à plusieurs clés

```
  PropertyInfo[] stringProps = typeof (string).GetProperties();//string properties
  PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties();//stringbuilder
properties

    var query =
        from s in stringProps
        join b in builderProps
            on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
        select new
        {
            s.Name,
            s.PropertyType,
            StringToken = s.MetadataToken,
            StringBuilderToken = b.MetadataToken
        };
```

Notez que les types anonymes dans la `join` ci-dessus doivent contenir les mêmes propriétés, car les objets sont considérés égaux uniquement si toutes leurs propriétés sont égales. Sinon, la requête ne sera pas compilée.

Sélectionnez avec Func sélecteur - Permet d'obtenir le classement des éléments

Sur les surcharges des méthodes d'extension `Select` , l' `index` de l'élément actuel de la collection `select` transmis. Ce sont quelques utilisations.

Récupère le "numéro de ligne" des articles

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

Obtenez le rang d'un article *dans* son groupe

```
var rankInGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
```

```
                            .SelectMany(group => group.OrderBy(item => item.Property2)
                                                      .ThenByDescending(item => item.Property3)
                                                      .Select((item, index) => new
                                                      {
                                                          Item = item,
                                                          RankInGroup = index
                                                      })).ToList();
```

Obtenez le classement des groupes (également connu sous Oracle sous le nom dense_rank)

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .Select((group, index) => new
                            {
                                Items = group,
                                Rank = index
                            })
                            .SelectMany(v => v.Items, (s, i) => new
                            {
                                Item = i,
                                DenseRank = s.Rank
                            }).ToList();
```

Pour tester cela, vous pouvez utiliser:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

Et des données:

```
List<SomeObject> collection = new List<SomeObject>
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}
};
```

TakeWhile

`TakeWhile` renvoie des éléments d'une séquence tant que la condition est vraie

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

Somme

La méthode d'extension `Enumerable.Sum` calcule la somme des valeurs numériques.

Si les éléments de la collection sont eux-mêmes des nombres, vous pouvez calculer directement la somme.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

Si le type des éléments est un type complexe, vous pouvez utiliser une expression lambda pour spécifier la valeur à calculer:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

La méthode d'extension de somme peut calculer avec les types suivants:

- Int32
- Int64
- Unique
- Double
- Décimal

Si votre collection contient des types nullables, vous pouvez utiliser l'opérateur null-coalescing pour définir une valeur par défaut pour les éléments null:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

Pour rechercher

> ToLookup renvoie une structure de données permettant l'indexation. C'est une méthode d'extension. Il produit une instance ILookup qui peut être indexée ou énumérée à l'aide d'une boucle foreach. Les entrées sont combinées en groupes à chaque touche. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",",lookup[3]));
//output: one,two
```

Un autre exemple:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",",lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(",",lookup[1]));
//output: 1,3,5,7
```

Construisez vos propres opérateurs Linq pour IEnumerable

L'un des avantages de Linq est qu'il est si facile à étendre. Il vous suffit de créer une méthode d'extension dont l'argument est `IEnumerable<T>` .

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

Cet exemple divise les éléments dans un `IEnumerable<T>` en listes d'une taille fixe, la dernière liste contenant le reste des éléments. Notez que l'objet auquel la méthode d'extension est appliquée est passé (argument `source`) comme argument initial à l'aide du mot `this` clé `this` . Ensuite, le mot-clé `yield` est utilisé pour générer l'élément suivant dans la sortie `IEnumerable<T>` avant de poursuivre l'exécution à partir de ce point (voir le mot-clé yield).

Cet exemple serait utilisé dans votre code comme ceci:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

Sur la première boucle, la sous-liste serait `{2, 3, 4}` et la seconde `{5, 6}` .

Les méthodes LinQ personnalisées peuvent également être combinées avec les méthodes LinQ standard. par exemple:

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)          // generate a list
                       .Where(x => x%2 == 0) // filter the list or do something other
                       .Batch(3)              // call our extension method
                       .ToList()              // call other standard methods
```

Cette requête renvoie des nombres pairs regroupés en lots de taille 3: `{0, 2, 4}, {6, 8, 10}, {12}`

N'oubliez pas que vous avez besoin de `using MyNamespace;` ligne pour pouvoir accéder à la méthode d'extension.

Utilisation de SelectMany au lieu de boucles imbriquées

2 listes données

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

si vous voulez sortir toutes les permutations, vous pouvez utiliser des boucles imbriquées comme

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

En utilisant SelectMany, vous pouvez effectuer la même opération que

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Any and First (OrDefault) - meilleures pratiques

Je n'expliquerai pas ce que font `Any` et `FirstOrDefault` car il y a déjà deux bons exemples à leur sujet. Voir Any et First, FirstOrDefault, Last, LastOrDefault, Single et SingleOrDefault pour plus d'informations.

Un modèle que je vois souvent dans le code qui **devrait être évité** est

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

Cela pourrait être écrit plus efficacement comme ça

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
```

```
{
    //Do stuff
}
```

En utilisant le deuxième exemple, la collection est recherchée une seule fois et donne le même résultat que le premier. La même idée peut être appliquée à `Single` .

GroupBy Sum et Count

Prenons un exemple de classe:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Considérons maintenant une liste de transactions:

```
var transactions = new List<Transaction>
{
   new Transaction { Category = "Saving Account", Amount = 56, Date =
DateTime.Today.AddDays(1) },
   new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-
10) },
   new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1)
},
   new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
   new Transaction { Category = "Current Account", Amount = 100, Date =
DateTime.Today.AddDays(5) },
};
```

Si vous voulez calculer la somme sage de la somme et le nombre, vous pouvez utiliser GroupBy comme suit:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
                           .Select(t => new
                           {
                               Category = t.Key,
                               Count = t.Count(),
                               Amount = t.Sum(ta => ta.Amount),
                           }).ToList();

Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
            row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

Sinon, vous pouvez le faire en une seule étape:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
        var transactionArray = t as Transaction[] ?? t.ToArray();
```

```
        return new
        {
            Category = key,
            Count = transactionArray.Length,
            Amount = transactionArray.Sum(ta => ta.Amount),
        };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

La sortie pour les deux requêtes ci-dessus serait identique:

> Catégorie: Compte d'épargne, Montant: 66, Nombre: 2
>
> Catégorie: Carte de crédit, montant: 71, nombre: 2
>
> Catégorie: Compte courant, montant: 100, compte: 1

Démo en direct dans .NET Fiddle

Sens inverse

- Inverse l'ordre des éléments dans une séquence.
- S'il n'y a pas d'éléments lève une `ArgumentNullException: source is null.`

Exemple:

```
// Create an array.
int[] array = { 1, 2, 3, 4 };                          //Output:
// Call reverse extension method on the array.         //4
var reverse = array.Reverse();                         //3
// Write contents of array to screen.                  //2
foreach (int value in reverse)                         //1
    Console.WriteLine(value);
```

Exemple de code en direct

Rappelez-vous que `Reverse()` peut fonctionner différemment en fonction de l'ordre des chaînes de vos déclarations LINQ.

```
        //Create List of chars
        List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

        //Reversing the list then taking the two first elements
        IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

        //Taking 2 elements and then reversing only thos two
        IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

        //reverseFirst output: 6, 5
        //reverseLast output:  2, 1
```

Exemple de code en direct

Reverse () fonctionne en tamponnant tout, puis marche en arrière, ce qui n'est pas très efficace, mais OrderBy ne l'est pas non plus.

Dans LINQ-to-Objects, il existe des opérations de mise en mémoire tampon (Reverse, OrderBy, GroupBy, etc.) et des opérations sans mise en mémoire tampon (Where, Take, Skip, etc.).

Exemple: extension inversée non tampon

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Exemple de code en direct

Cette méthode peut rencontrer des problèmes si vous modifiez la liste pendant une itération.

Énumérer les Enumerables

L'interface IEnumerable <T> est l'interface de base pour tous les énumérateurs génériques et constitue une partie essentielle de la compréhension de LINQ. À la base, il représente la séquence.

Cette interface sous-jacente est héritée par toutes les collections génériques, telles que Collection <T> , Array , List <T> , Dictionary <TKey, TValue> Class et HashSet <T> .

En plus de représenter la séquence, toute classe qui hérite de IEnumerable <T> doit fournir un IEnumerator <T>. L'énumérateur expose l'itérateur à l'énumérable, et ces deux interfaces et idées interconnectées sont à l'origine du dicton "énumérer les énumérables".

"Enumerating the enumerable" est une phrase importante. L'énumérateur est simplement une structure pour savoir comment itérer, il ne contient aucun objet matérialisé. Par exemple, lors du tri, un énumérable peut contenir les critères du champ à trier, mais l'utilisation de `.OrderBy()` en lui-même renverra un IEnumerable <T> qui ne sait *que* trier. L'utilisation d'un appel qui matérialisera les objets, comme dans l'itération de l'ensemble, est appelée énumération (par exemple `.ToList()`). Le processus d'énumération utilisera la définition énumérable de la *manière* de parcourir les séries et de renvoyer les objets pertinents (dans l'ordre, filtrés, projetés, etc.).

Ce n'est qu'une fois que l'énumérable a été énuméré que cela provoque la matérialisation des objets, c'est-à-dire lorsque des métriques telles que la complexité temporelle (durée liée à la taille des séries) et la complexité spatiale (quantité Être mesuré.

Créer votre propre classe qui hérite de IEnumerable <T> peut être un peu compliqué en fonction de la série sous-jacente qui doit être énumérée. En général, il est préférable d'utiliser l'une des collections génériques existantes. Cela dit, il est également possible d'hériter de l'interface IEnumerable <T> sans avoir un tableau défini comme structure sous-jacente.

Par exemple, utiliser la série Fibonacci comme séquence sous-jacente. Notez que l'appel à `Where`

construit simplement un `IEnumerable` , et ce n'est qu'un appel à énumérer que énumérable est fait que l'une des valeurs est matérialisée.

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Sortie

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

La force du second ensemble (le fibMod612) est que même si nous avons appelé pour ordonner l'ensemble de nos nombres de Fibonacci, une seule valeur ayant été prise avec `.First()` la

complexité du temps était O (n) comme une seule valeur. besoin d'être comparé lors de l'exécution de l'algorithme de commande. Ceci est dû au fait que notre enquêteur n'a demandé qu'une valeur, de sorte que l'intégralité de l'énumérateur n'a pas à être matérialisée. Si nous avions utilisé `.Take(5)` au lieu de `.First()` l'énumérateur aurait demandé 5 valeurs et au plus 5 valeurs devraient être matérialisées. Par rapport à la nécessité de commander un ensemble complet *et de* prendre *ensuite* les 5 premières valeurs, le principe d'économiser beaucoup de temps d'exécution et d'espace.

Commandé par

> Ordonne une collection par une valeur spécifiée.

Lorsque la valeur est un **entier** , **double** ou **float,** elle commence par la *valeur minimale* , ce qui signifie que vous obtenez d'abord les valeurs négatives, puis zéro et les mots après les valeurs positives (voir l'exemple 1).

Lorsque vous commandez un **ombles** la méthode compare les *valeurs ascii* des caractères pour trier la collection (voir l' exemple 2).

Lorsque vous triez des **chaînes,** la méthode OrderBy les compare en examinant CultureInfo, mais en commençant normalement par la *première lettre* de l'alphabet (a, b, c ...).

Ce type d'ordre est appelé ascendant, si vous voulez qu'il en soit autrement (voir OrderByDescending).

Exemple 1:

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

Exemple 2:

```
 char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
 IEnumerable<char> ascending = letters.OrderBy(x => x);
 // returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', '{' }
```

Exemple:

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
```

```
};
var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob
```

OrderByDescending

Ordonne une collection par une valeur spécifiée.

Lorsque la valeur est un **entier** , **double** ou **float,** elle commence par la *valeur maximale* , ce qui signifie que vous obtenez d'abord les valeurs positives, et zéro les valeurs négatives (voir l'exemple 1).

Lorsque vous commandez un **ombles** la méthode compare les *valeurs ascii* des caractères pour trier la collection (voir l' exemple 2).

Lorsque vous triez des **chaînes,** la méthode OrderBy les compare en examinant CultureInfo, mais en commençant normalement par la *dernière lettre* de l'alphabet (z, y, x, ...).

Ce type d'ordre est appelé décroissant, si vous voulez l'inverse, vous devez l'escalader (voir OrderBy).

Exemple 1:

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

Exemple 2:

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }
```

Exemple 3:

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

Concat

Fusionne deux collections (sans supprimer les doublons)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

Contient

MSDN:

> Détermine si une séquence contient un élément spécifié à l'aide d'un `IEqualityComparer<T>` spécifié

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true
only for 4,5
```

En utilisant un objet défini par l'utilisateur:

```
public class Person
{
   public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki"},
    new Person { Name = "Gilad"},
    new Person { Name = "Phil"},
    new Person { Name = "John"}
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

Utilisation de la `Enumerable.Contains(value, comparer)` :

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
```

```
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

Une utilisation judicieuse de `Contains` serait de remplacer plusieurs clauses `if` par un appel `Contains`.

Donc, au lieu de faire ceci:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Faites ceci:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

Lire Requêtes LINQ en ligne: https://riptutorial.com/fr/csharp/topic/68/requetes-linq

Chapitre 140: Résolution de surcharge

Remarques

Le processus de résolution de surcharge est décrit dans la spécification C # , section 7.5.3. Les sections 7.5.2 (inférence de type) et 7.6.5 (expressions d'invocation) sont également pertinentes.

La manière dont la résolution de la surcharge fonctionne sera probablement modifiée dans C # 7. Les notes de conception indiquent que Microsoft déploiera un nouveau système pour déterminer quelle méthode est la meilleure (dans les scénarios complexes).

Examples

Exemple de surcharge de base

Ce code contient une méthode surchargée nommée **Hello** :

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Lorsque la méthode **principale** est appelée, elle sera imprimée

```
int
double
```

Au moment de la compilation, lorsque le compilateur trouve l'appel à la méthode `Hello(0)` , il trouve toutes les méthodes portant le nom `Hello` . Dans ce cas, il en trouve deux. Il essaie ensuite de déterminer laquelle des méthodes est la *meilleure* . L'algorithme permettant de déterminer quelle méthode est la meilleure est complexe, mais cela revient généralement à "effectuer le moins de conversions implicites possible".

Ainsi, dans le cas de `Hello(0)` , aucune conversion n'est nécessaire pour la méthode `Hello(int)` mais une conversion numérique implicite est nécessaire pour la méthode `Hello(double)` . Ainsi, la

première méthode est choisie par le compilateur.

Dans le cas de `Hello(0.0)` , il n'y a aucun moyen de convertir implicitement `0.0` en `int` , donc la méthode `Hello(int)` n'est même pas prise en compte pour la résolution de la surcharge. Seule la méthode reste et elle est donc choisie par le compilateur.

"params" n'est pas développé, sauf si nécessaire.

Le programme suivant:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

imprimera:

```
5
two
3
```

La `Method(objectArray)` expression d'appel `Method(objectArray)` peut être interprétée de deux manières: un seul argument d' `Object` qui se trouve être un tableau (le programme afficherait donc `1` car ce serait le nombre d'arguments, ou un tableau d'arguments, donné dans le forme normale, comme si la méthode `Method` n'a pas le mot - clé `params` . Dans ces situations, la normale, sous forme non expansé a toujours la priorité. Ainsi, les résultats du programme `5` .

Dans la deuxième expression, `Method(objectArray, objectArray)` , la forme développée de la première méthode et la seconde méthode traditionnelle sont applicables. Dans ce cas également, les formulaires non développés sont prioritaires, de sorte que le programme imprime `two` .

Dans la troisième expression, `Method(objectArray, objectArray, objectArray)` , la seule option consiste à utiliser la forme développée de la première méthode et le programme imprime donc `3` .

Passer null comme l'un des arguments

Si tu as

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

et pour une raison quelconque, vous devez appeler la première surcharge de `F1` mais avec `x = null` , puis faire simplement

```
F1(null);
```

ne compilera pas car l'appel est ambigu. Pour contrer cela, vous pouvez faire

```
F1(null as MyType1);
```

Lire Résolution de surcharge en ligne: https://riptutorial.com/fr/csharp/topic/77/resolution-de-surcharge

Chapitre 141: Runtime Compile

Examples

RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript` est un nouveau moteur de script C #.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

Vous pouvez compiler et exécuter des instructions, des variables, des méthodes, des classes ou des segments de code.

CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider` peut être utilisé pour compiler des classes C #.

```
var code = @"
    public class Abc {
       public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

Lire Runtime Compile en ligne: https://riptutorial.com/fr/csharp/topic/3139/runtime-compile

Chapitre 142: Séquences d'échappement de chaîne

Syntaxe

- \ '- guillemet simple (0x0027)
- \ "- guillemet double (0x0022)
- \\ - barre oblique inverse (0x005C)
- \ 0 - null (0x0000)
- \ a - alerte (0x0007)
- \ b - retour arrière (0x0008)
- \ f - flux de formulaire (0x000C)
- \ n - nouvelle ligne (0x000A)
- \ r - retour chariot (0x000D)
- \ t - tabulation horizontale (0x0009)
- \ v - onglet vertical (0x000B)
- \ u0000 - \ uFFFF - Caractère Unicode
- \ x0 - \ xFFFF - Caractère Unicode (code de longueur variable)
- \ U00000000 - \ U0010FFFF - Caractère Unicode (pour générer des substituts)

Remarques

Les séquences d'échappement de chaînes sont transformées en caractères correspondants au **moment de la compilation** . Les chaînes ordinaires qui contiennent des barres obliques **ne** sont **pas** transformées.

Par exemple, les chaînes `notEscaped` et `notEscaped2` ci-dessous ne sont pas transformées en un caractère de nouvelle ligne, mais resteront deux caractères différents (`'\'` et `'n'`).

```
string escaped = "\n";
string notEscaped = "\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

Examples

Séquences d'échappement de caractères Unicode

```
string sqrt = "\u221A";      // √
string emoji = "\U0001F601"; // 
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

Échapper à des symboles spéciaux dans les littéraux de caractère

Apostrophes

```
char apostrophe = '\'';
```

Barre oblique inverse

```
char oneBackslash = '\\';
```

Échapper aux symboles spéciaux dans les littéraux de chaîne

Barre oblique inverse

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

Le deuxième exemple utilise une chaîne littérale textuelle , qui ne traite pas la barre oblique inverse comme un caractère d'échappement.

Citations

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @"""Hello World!"", said the quick brown fox.";
```

Les deux variables contiendront le même texte.

> "Hello World!", Dit le rapide renard brun.

Nouvelles lignes

Les littéraux de chaîne verbatim peuvent contenir des nouvelles lignes:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Les deux variables contiendront le même texte.

Les séquences d'échappement non reconnues génèrent des erreurs de compilation

Les exemples suivants ne compileront pas:

```
string s = "\c";
char c = '\c';
```

Au lieu de cela, ils produiront la `Unrecognized escape sequence` Erreur `Unrecognized escape sequence` au moment de la compilation.

Utilisation de séquences d'échappement dans les identificateurs

Les séquences d' échappement ne sont pas limités à `string` et `char` littéraux.

Supposons que vous deviez remplacer une méthode tierce:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

et supposons que le caractère `Œ` ne soit pas disponible dans le codage de caractères que vous utilisez pour vos fichiers source C #. Vous avez de la chance, il est permis d'utiliser des échappements du type `\u####` ou `\U########` dans les ***identifiants*** du code. Il est donc légal d'écrire:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

et le compilateur C # saura que `Œ` et `\u0152` ont le même caractère.

(Cependant, il peut être judicieux de passer à l'UTF-8 ou à un encodage similaire capable de traiter tous les caractères.)

Lire Séquences d'échappement de chaîne en ligne:
https://riptutorial.com/fr/csharp/topic/39/sequences-d-echappement-de-chaine

Chapitre 143: Sérialisation Binaire

Remarques

Le moteur de sérialisation binaire fait partie du framework .NET, mais les exemples donnés ici sont spécifiques à C #. Par rapport aux autres moteurs de sérialisation intégrés au framework .NET, le sérialiseur binaire est rapide et efficace et nécessite généralement très peu de code supplémentaire pour le faire fonctionner. Cependant, il est également moins tolérant aux modifications de code; En d'autres termes, si vous sérialisez un objet et modifiez légèrement la définition de l'objet, il ne sera probablement pas désérialisé correctement.

Examples

Rendre un objet sérialisable

Ajoutez l'attribut `[Serializable]` pour marquer un objet entier pour la sérialisation binaire:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

Tous les membres seront sérialisés à moins que nous ne les désactivions explicitement en utilisant l'attribut `[NonSerialized]` . Dans notre exemple, `X` , `Y` , `Z` et `Name` sont tous sérialisés.

Tous les membres doivent être présents lors de la désérialisation à moins qu'ils ne soient marqués avec `[NonSerialized]` ou `[OptionalField]` . Dans notre exemple, `X` , `Y` et `Z` sont tous requis et la désérialisation échouera s'ils ne sont pas présents dans le flux. `DontSerializeThis` sera toujours défini sur la `default(decimal)` (qui est 0). Si `Name` est présent dans le flux, alors il sera mis à cette valeur, sinon il sera défini sur `default(string)` (qui est nul). Le but de `[OptionalField]` est de fournir un peu de tolérance de version.

Contrôle du comportement de sérialisation avec des attributs

Si vous utilisez l'attribut `[NonSerialized]` , alors ce membre aura toujours sa valeur par défaut après la désérialisation (ex. 0 pour un `int` , null pour une `string` , false pour un `bool` , etc.), quelle que soit l'initialisation de l'objet (constructeurs, déclarations, etc.). Pour compenser, les attributs `[OnDeserializing]` (appelé juste avant la désérialisation) et `[OnDeserialized]` (appelé juste après la

désérialisation) avec leurs homologues, `[OnSerializing]` et `[OnSerialized]` sont fournis.

Supposons que nous voulions ajouter un "Rating" à notre vecteur et que nous voulons nous assurer que la valeur commence toujours à 1. La manière dont il est écrit ci-dessous sera 0 après avoir été désérialisé:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

Pour résoudre ce problème, nous pouvons simplement ajouter la méthode suivante à l'intérieur de la classe pour la définir à 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Ou, si nous voulons le définir à une valeur calculée, nous pouvons attendre la fin de la désérialisation, puis la définir:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

De même, nous pouvons contrôler la façon dont les choses sont écrites en utilisant `[OnSerializing]` et `[OnSerialized]` .

Ajouter plus de contrôle en implémentant ISerializable

Cela permettrait de mieux contrôler la sérialisation, de sauvegarder et de charger les types

Implémenter une interface ISerializable et créer un constructeur vide pour compiler

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

Pour la sérialisation des données, vous pouvez spécifier le nom souhaité et le type souhaité

```
info.AddValue("_name", _name, typeof(string));
```

Lorsque les données sont désérialisées, vous pourrez lire le type souhaité

```
_name = (string)info.GetValue("_name", typeof(string));
```

Les substituts de sérialisation (implémentation d'ISerializationSurrogate)

Implémente un sélecteur de substitution de sérialisation qui permet à un objet d'effectuer la sérialisation et la désérialisation d'un autre

Cela permet aussi de sérialiser ou désérialiser correctement une classe qui n'est pas elle-même sérialisable

Implémenter l'interface ISerializationSurrogate

```
public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
```

```
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}
```

Ensuite, vous devez informer votre IFormatter des substituts en définissant et en initialisant un SurrogateSelector et en l'attribuant à votre IFormatter.

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

Même si la classe n'est pas marquée sérialisable.

```
//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

La solution complète

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext
context)
        {
```

```
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext
context, ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Classeur de sérialisation

Le classeur vous permet d'examiner les types chargés dans votre domaine d'application

Créer une classe héritée de SerializationBinder

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Maintenant, nous pouvons vérifier quels types sont en cours de chargement et sur cette base pour décider ce que nous voulons vraiment recevoir

Pour utiliser un classeur, vous devez l'ajouter au BinaryFormatter.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

La solution complète

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
        {
```

```
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var binaryFormatter = new BinaryFormatter();
            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var binaryFormatter = new BinaryFormatter
            {
                Binder = new MyBinder()
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Quelques pièges dans la compatibilité ascendante

Ce petit exemple montre comment vous pouvez perdre la rétrocompatibilité dans vos programmes si vous ne vous en souciez pas à l'avance. Et des moyens de mieux contrôler le processus de sérialisation

Au début, nous allons écrire un exemple de la première version du programme:

Version 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
```

```
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

Et maintenant, supposons que dans la deuxième version du programme ajouté une nouvelle classe. Et nous devons le stocker dans un tableau.

Maintenant, le code ressemblera à ceci:

Version 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

Et code pour sérialiser et désérialiser

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
```

```
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

Et alors, que se passerait-il lorsque vous sérialisez les données dans le programme de v2 et que vous essayez de les désérialiser dans le programme de v1?

Vous obtenez une exception:

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem
in the Formatter.Source=mscorlib
StackTrace:
   at System.Runtime.Serialization.ObjectManager.DoFixups()
   at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler
handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
   at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
   at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

Pourquoi?

ObjectManager a une logique différente pour résoudre les dépendances pour les tableaux et pour les types de référence et de valeur. Nous avons ajouté un tableau de nouveau le type de référence qui est absent dans notre assemblage.

Lorsque ObjectManager tente de résoudre des dépendances, il génère le graphique. Lorsqu'il voit le tableau, il ne peut pas le réparer immédiatement, de sorte qu'il crée une référence factice, puis corrige le tableau ultérieurement.

Et puisque ce type n'est pas dans l'assemblage et que les dépendances ne peuvent pas être corrigées. Pour une raison quelconque, il ne supprime pas le tableau de la liste des éléments pour les correctifs et à la fin, il génère une exception «IncorrectNumberOfFixups».

Ce sont des «pièges» dans le processus de sérialisation. Pour une raison quelconque, il ne fonctionne pas correctement uniquement pour les tableaux de nouveaux types de référence.

```
A Note:
Similar code will work correctly if you do not use arrays with new classes
```

Et le premier moyen de résoudre ce problème et de maintenir la compatibilité?

- Utiliser une collection de nouvelles structures plutôt que des classes ou utiliser un dictionnaire (classes possibles), car un dictionnaire est une collection de keyvaluepair (sa structure)
- Utilisez ISerializable, si vous ne pouvez pas changer l'ancien code

Lire Sérialisation Binaire en ligne: https://riptutorial.com/fr/csharp/topic/4120/serialisation-binaire

Chapitre 144: String.Format

Introduction

Les méthodes `Format` sont un ensemble de surcharges dans la classe `System.String` utilisées pour créer des chaînes qui combinent des objets dans des représentations de chaîne spécifiques. Ces informations peuvent être appliquées à `String.Format` , à diverses méthodes `WriteLine` et à d'autres méthodes du framework .NET.

Syntaxe

- string.Format (format de chaîne, params object [] args)
- string.Format (fournisseur IFormatProvider, format de chaîne, paramètres params [] args)
- $ "string {text} blablabla" // Depuis C # 6

Paramètres

Paramètre	Détails
format	Une chaîne de format composite , qui définit la manière dont les *arguments* doivent être combinés en une chaîne.
args	Une séquence d'objets à combiner en une chaîne. Comme il utilise un argument `params` , vous pouvez utiliser une liste d'arguments séparés par des virgules ou un tableau d'objets réel.
fournisseur	Une collection de façons de formater des objets en chaînes. Les valeurs typiques incluent CultureInfo.InvariantCulture et CultureInfo.CurrentCulture .

Remarques

Remarques:

- `String.Format()` gère `null` arguments `null` sans lancer d'exception.
- Il existe des surcharges qui remplacent le paramètre `args` par un, deux ou trois paramètres d'objet.

Examples

Endroits où String.Format est "incorporé" dans la structure

Il y a plusieurs endroits où vous pouvez utiliser `String.Format` *indirectement* : Le secret est de rechercher la surcharge avec le `string format, params object[] args` signature `string format,`

`params object[] args` , par exemple:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Peut être remplacé par une version plus courte:

```
Console.WriteLine("{0} - {1}", name, value);
```

Il existe d'autres méthodes qui utilisent également `String.Format` par exemple:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

Utilisation du format numérique personnalisé

`NumberFormatInfo` peut être utilisé pour formater des nombres entiers et flottants.

```
// invariantResult is "1,234,567.89"
var invarianResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);

// NumberFormatInfo is one of classes that implement IFormatProvider
var customProvider = new NumberFormatInfo
{
    NumberDecimalSeparator = "_NS_", // will be used instead of ','
    NumberGroupSeparator = "_GS_", // will be used instead of '.'
};

// customResult is "1_GS_234_GS_567_NS_89"
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

Créer un fournisseur de format personnalisé

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
        {
            return null;
        }

        if (format == "Reverse")
        {
            return String.Join("", arg.ToString().Reverse());
        }

        return arg.ToString();
    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}
```

Usage:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Sortie:

```
-> dlroW olleH <-
```

Aligner à gauche / à droite, pad avec des espaces

La deuxième valeur dans les accolades indique la longueur de la chaîne de remplacement. En ajustant la deuxième valeur pour qu'elle soit positive ou négative, l'alignement de la chaîne peut être modifié.

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Sortie:

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

Formats numériques

```
// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

Sortie:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

Mise en forme de devise

Le spécificateur de format "c" (ou devise) convertit un nombre en une chaîne représentant un montant en devise.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

Précision

La valeur par défaut est 2. Utilisez c1, c2, c3, etc. pour contrôler la précision.

```
string.Format("{0:C1}", 112.236677) //$112.2
string.Format("{0:C3}", 112.236677) //$112.237
string.Format("{0:C4}", 112.236677) //$112.2367
string.Format("{0:C9}", 112.236677) //$112.236677000
```

Symbole de la monnaie

1. Passez l'instance de `CultureInfo` pour utiliser le symbole de culture personnalisé.

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); //$112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); //112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); //□ 112.24
```

2. Utilisez n'importe quelle chaîne comme symbole de devise. Utilisez `NumberFormatInfo` pour personnaliser le symbole monétaire.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

Position du symbole monétaire

Utilisez CurrencyPositivePattern pour les valeurs positives et CurrencyNegativePattern pour les valeurs négatives.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); //$112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); //112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); //$ 112.24
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

L'utilisation d'un motif négatif est identique à un motif positif. Beaucoup plus de cas d'utilisation s'il vous plaît se référer au lien original.

Séparateur décimal personnalisé

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
```

```
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

Depuis C # 6.0

6,0

Depuis C # 6.0, il est possible d'utiliser l'interpolation de chaînes à la place de `String.Format` .

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Bonjour John Doe!

Plus d'exemples pour cela sous la rubrique Fonctions C # 6.0: Interpolation de chaînes .

Echappement des accolades dans une expression String.Format ()

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "{I am in brackets!} I am outside of bracket"
```

Formatage de date

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.Write(String.Format("{0:dd}",date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"),"{0:dddd}",date);
```

6,0

```
Console.Write($"{date:ddd}");
```

sortie:

```
06
Лхагва
06
```

Spécificateur	Sens	Échantillon	Résultat
ré	Rendez-vous amoureux	`{0:d}`	7/6/2016
dd	Jour, sans rembourrage	`{0:dd}`	06

Spécificateur	Sens	Échantillon	Résultat
ddd	Nom de jour court	`{0:ddd}`	Mer
dddd	Nom de la journée complète	`{0:dddd}`	Mercredi
ré	Longue date	`{0:D}`	Mercredi 6 juillet 2016
F	Date complète et heure, courte	`{0:f}`	Mercredi 6 juillet 2016 18h30
ff	Secondes fractions, 2 chiffres	`{0:ff}`	20
fff	Secondes fractions, 3 chiffres	`{0:fff}`	201
ffff	Deuxième fraction, 4 chiffres	`{0:ffff}`	2016
F	Date et heure complètes, longues	`{0:F}`	Mercredi 6 juillet 2016 18h30
g	Date et heure par défaut	`{0:g}`	7/6/2016 18h30
gg	Ère	`{0:gg}`	UN D
hh	Heure (2 chiffres, 12H)	`{0:hh}`	06
HH	Heure (2 chiffres, 24H)	`{0:HH}`	18
M	Mois et jour	`{0:M}`	6 juillet
mm	Minutes, zéro-rembourré	`{0:mm}`	30
MM	Mois, zéro-rembourré	`{0:MM}`	07
MMM	Nom du mois à 3 lettres	`{0:MMM}`	Juil
MMMM	Nom complet du mois	`{0:MMMM}`	juillet
ss	Secondes	`{0:ss}`	14
r	RFC1123 date	`{0:r}`	Mer. 06 juil. 2016 18:30:14 GMT
s	Chaîne de date triable	`{0:s}`	2016-07-06T18: 30: 14
t	Court instant	`{0:t}`	18H30
T	Longtemps	`{0:T}`	18h30
tt	MATIN APRÈS-MIDI	`{0:tt}`	PM
tu	Heure locale triable universelle	`{0:u}`	2016-07-06 18: 30: 14Z

Spécificateur	Sens	Échantillon	Résultat
U	Universal GMT	`{0:U}`	Mercredi 6 juillet 2016 09:30:14
Y	Mois et année	`{0:Y}`	Juillet 2016
yy	Année à 2 chiffres	`{0:yy}`	16
aaaa	Année à 4 chiffres	`{0:yyyy}`	2016
zz	Décalage du fuseau horaire à 2 chiffres	`{0:zz}`	+09
zzz	décalage du fuseau horaire complet	`{0:zzz}`	+09: 00

ToString ()

La méthode ToString () est présente sur tous les types d'objets de référence. Cela est dû au fait que tous les types de référence sont dérivés de Object qui a la méthode ToString (). La méthode ToString () sur la classe de base de l'objet renvoie le nom du type. Le fragment ci-dessous imprimera "Utilisateur" sur la console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Cependant, la classe User peut également remplacer ToString () afin de modifier la chaîne renvoyée. Le fragment de code ci-dessous affiche "Id: 5, Name: User1" sur la console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Relation avec ToString ()

Bien que la méthode `String.Format()` soit certainement utile pour formater des données sous forme de chaînes, cela peut souvent être un peu exagéré, en particulier lorsque vous travaillez avec un seul objet, comme indiqué ci-dessous:

```
String.Format("{0:C}", money);  // yields "$42.00"
```

Une approche plus simple pourrait être d'utiliser simplement la méthode `ToString()` disponible sur tous les objets de C #. Il prend en charge toutes les mêmes chaînes de formatage standard et personnalisées , mais ne nécessite pas le mappage de paramètres nécessaire, car il n'y aura qu'un seul argument:

```
money.ToString("C");  // yields "$42.00"
```

Restrictions relatives aux mises en garde et au formatage

Bien que cette approche soit peut-être plus simple dans certains scénarios, l'approche `ToString()` est limitée en ce qui concerne l'ajout de padding à gauche ou à droite comme vous le feriez dans la méthode `String.Format()` :

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

Pour accomplir ce même comportement avec la méthode `ToString()` , vous devez utiliser une autre méthode comme `PadLeft()` ou `PadRight()` respectivement:

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

Lire String.Format en ligne: https://riptutorial.com/fr/csharp/topic/79/string-format

Chapitre 145: StringBuilder

Examples

Qu'est-ce qu'un StringBuilder et quand l'utiliser

Un `StringBuilder` représente une série de caractères qui, contrairement à une chaîne normale, sont mutables. Souvent, il est nécessaire de modifier les chaînes que nous avons déjà créées, mais l'objet de chaîne standard n'est pas mutable. Cela signifie que chaque fois qu'une chaîne est modifiée, un nouvel objet chaîne doit être créé, copié et réaffecté.

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

Dans l'exemple ci-dessus, `myString` n'a initialement que la valeur `"Apples"` . Cependant, lorsque nous concaténons `` sont mes fruits préférés '', ce que la classe de chaînes a besoin de faire en interne implique:

- Créer un nouveau tableau de caractères égal à la longueur de `myString` et la nouvelle chaîne que nous ajoutons.
- Copier tous les caractères de `myString` au début de notre nouveau tableau et copier la nouvelle chaîne à la fin du tableau.
- Créez un nouvel objet chaîne en mémoire et réaffectez-le à `myString` .

Pour une seule concaténation, ceci est relativement trivial. Cependant, si nécessaire pour effectuer de nombreuses opérations d'ajout, par exemple en boucle?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

En raison de la copie répétée et de la création d'objets, les performances de notre programme s'en trouveront considérablement dégradées. Nous pouvons éviter cela en utilisant un `StringBuilder` .

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Maintenant, lorsque la même boucle est exécutée, les performances et la vitesse d'exécution du programme seront nettement plus rapides que l'utilisation d'une chaîne normale. Pour rendre le `StringBuilder` dans une chaîne normale, nous pouvons simplement appeler la méthode `ToString()` de `StringBuilder` .

Cependant, ce n'est pas la seule optimisation de `StringBuilder` . Afin d'optimiser davantage les fonctions, nous pouvons tirer parti d'autres propriétés qui contribuent à améliorer les

performances.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

Si nous savons à l'avance combien de temps notre `StringBuilder` doit être, nous pouvons spécifier sa taille à l'avance, ce qui l'empêchera de redimensionner le tableau de caractères interne.

```
sb.Append('k', 2000);
```

Bien que l'utilisation de `StringBuilder` pour l'ajout soit beaucoup plus rapide qu'une chaîne, elle peut s'exécuter encore plus rapidement si vous n'avez besoin d'ajouter qu'un seul caractère plusieurs fois.

Une fois que vous avez terminé la construction de votre chaîne, vous pouvez utiliser la méthode `ToString()` sur `StringBuilder` pour la convertir en `string` base. Cela est souvent nécessaire car la classe `StringBuilder` n'hérite pas de la `string` .

Par exemple, voici comment vous pouvez utiliser un `StringBuilder` pour créer une `string` :

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

En conclusion, `StringBuilder` devrait être utilisé à la place de string lorsque de nombreuses modifications à une chaîne doivent être effectuées en tenant compte des performances.

Utilisez StringBuilder pour créer une chaîne à partir d'un grand nombre d'enregistrements

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
       customerNamesCsv
           .Append(record.LastName)
           .Append(',')
           .Append(record.FirstName)
           .Append(Environment.NewLine);
    }
```

```
    return customerNamesCsv.ToString();
}
```

Lire StringBuilder en ligne: https://riptutorial.com/fr/csharp/topic/4675/stringbuilder

Chapitre 146: Structs

Remarques

Contrairement aux classes, une `struct` est un type de valeur et est créée *par défaut* sur la pile locale et non sur le segment de mémoire géré. Cela signifie qu'une fois que la pile spécifique est hors de portée, la `struct` est `struct` . Les types de référence `struct` les `struct` sont également balayés, une fois que le GC détermine qu'ils ne sont plus référencés par la `struct` .

`struct` ne peuvent pas hériter et ne peuvent pas être des bases d'héritage, elles sont implicitement scellées et ne peuvent pas non plus inclure de membres `protected` . Cependant, une `struct` peut implémenter une interface, comme le font les classes.

Examples

Déclarer une structure

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct` champs d'instance de `struct` peuvent être définis via un constructeur paramétré ou individuellement après la construction d'une `struct` .
- Les membres privés ne peuvent être initialisés que par le constructeur.
- `struct` définit un type scellé qui hérite implicitement de System.ValueType.
- Les structures ne peuvent hériter d'aucun autre type, mais elles peuvent implémenter des interfaces.
- Les structures sont copiées lors de l'affectation, ce qui signifie que toutes les données sont copiées dans la nouvelle instance et que les modifications apportées à l'une d'entre elles ne sont pas prises en compte par l'autre.

- Une structure ne peut pas être `null` , bien qu'elle *puisse être* utilisée comme un type nullable:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Les structures peuvent être instanciées avec ou sans l'opérateur `new` .

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

  Cependant, le `new` opérateur doit être utilisé pour utiliser un initialiseur:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

Une structure peut déclarer tout ce qu'une classe peut déclarer, à quelques exceptions près:

- Une structure ne peut pas déclarer un constructeur sans paramètre. `struct` champs d'instance `struct` peuvent être définis via un constructeur paramétré ou individuellement après la construction d'une `struct` . Les membres privés ne peuvent être initialisés que par le constructeur.
- Une structure ne peut pas déclarer les membres protégés, car elle est implicitement scellée.
- Les champs de structure ne peuvent être initialisés que s'ils sont const ou statiques.

Utilisation de la structure

Avec constructeur:

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=0,Y=2,Z=3
```

```
Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

Sans constructeur:

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

Si nous utilisons une structure avec son constructeur, nous n'allons pas avoir de problèmes avec un champ non attribué (chaque champ non attribué a une valeur nulle).

Contrairement aux classes, une structure ne doit pas être construite, c'est-à-dire qu'il n'est pas nécessaire d'utiliser le nouveau mot-clé, sauf si vous devez appeler l'un des constructeurs. Une structure ne nécessite pas le nouveau mot-clé car est un type de valeur et ne peut donc pas être nul.

Interface d'implémentation Struct

```
public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{
    public decimal Length { get; set; }
    public decimal Width { get; set; }

    public decimal Area()
    {
        return Length * Width;
    }
}
```

Les structures sont copiées lors de l'affectation

Les structures Sinse sont des types de valeur que toutes les données sont *copiées* lors de leur affectation, et toute modification de la nouvelle copie ne modifie pas les données de la copie originale. L'extrait de code ci-dessous montre que `p1` est *copié* dans `p2` et que les modifications apportées sur `p1` n'affectent pas l'instance `p2` .

```
var p1 = new Point {
    x = 1,
    y = 2
};

Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

Lire Structs en ligne: https://riptutorial.com/fr/csharp/topic/778/structs

Chapitre 147: System.DirectoryServices.Protocols.LdapConnec

Examples

Connexion SSL LDAP authentifiée, le certificat SSL ne correspond pas au DNS inversé

Configurez des constantes pour le serveur et les informations d'authentification. En supposant LDAPv3, mais il est assez facile de changer cela.

```
// Authentication, and the name of the server.
private const string LDAPUser =
"cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

Créez en fait la connexion en trois parties: un identifiant LdapDirectoryIdentifier (le serveur) et des éléments NetworkCredential.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
        secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//    connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},",
TargetServer));
```

Utilisez le serveur LDAP, par exemple, recherchez quelqu'un par userid pour toutes les valeurs objectClass. ObjectClass est présent pour démontrer une recherche composée: L'esperluette est l'opérateur booléen "et" pour les deux clauses de requête.

```
 SearchRequest searchRequest = new SearchRequest(
        CompanyDN,
        string.Format((&(objectClass=*)(uid={0}))), uid),
        SearchScope.Subtree,
        null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

LDAP anonyme super simple

En supposant LDAPv3, mais il est assez facile de changer cela. Ceci est une création LDAPv3 LdapConnection anonyme et non cryptée.

```
private const string TargetServer = "ldap.example.com";
```

Créez en fait la connexion en trois parties: un identifiant LdapDirectoryIdentifier (le serveur) et des éléments NetworkCredential.

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

Pour utiliser la connexion, quelque chose comme ça amènerait les gens avec le nom de famille Smith

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

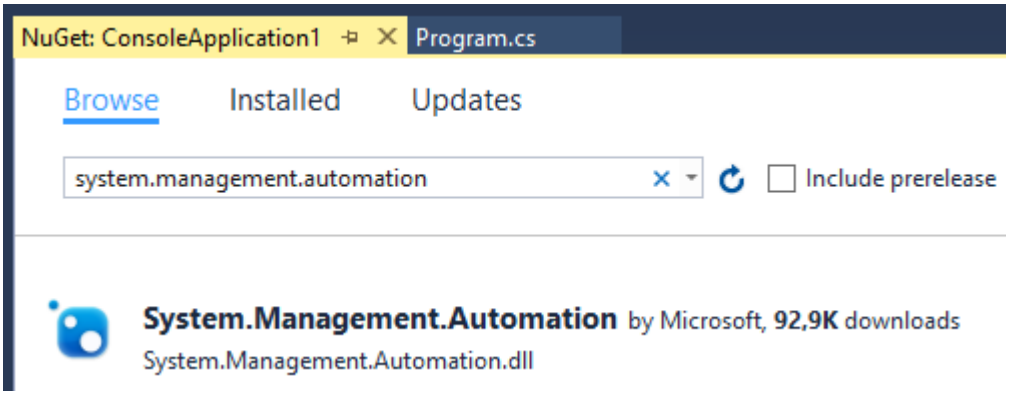
Lire System.DirectoryServices.Protocols.LdapConnection en ligne:
https://riptutorial.com/fr/csharp/topic/5177/system-directoryservices-protocols-ldapconnection

Chapitre 148: System.Management.Automation

Remarques

> L'espace de noms *System.Management.Automation* est l'espace de noms racine pour Windows PowerShell.

System.Management.Automation est une bibliothèque d'extension de Microsoft et peut être ajoutée aux projets Visual Studio via le gestionnaire de packages NuGet ou la console du gestionnaire de packages.



```
PM> Install-Package System.Management.Automation
```

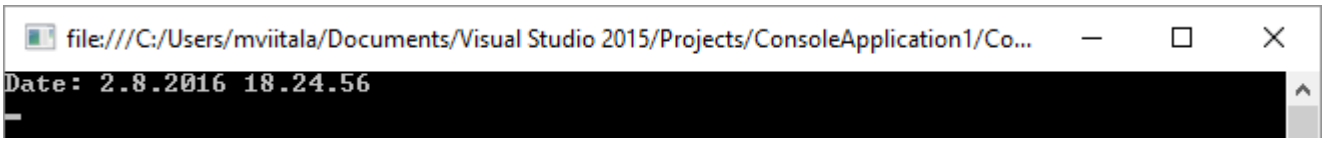
Examples

Invoquer un pipeline synchrone simple

Obtenez la date et l'heure actuelles.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
```

```
}
```



Lire System.Management.Automation en ligne: https://riptutorial.com/fr/csharp/topic/4988/system-management-automation

Chapitre 149: Tableaux

Syntaxe

- **Déclarer un tableau:**

  <type> [] <nom>;

- **Déclarer un tableau à deux dimensions:**

  <type> [,] <nom> = nouveau <type> [<valeur>, <valeur>];

- **Déclaration d'un tableau dentelé:**

  <type> [] <nom> = nouveau <type> [<valeur>];

- **Déclarer un sous-tableau pour un tableau dentelé:**

  <nom> [<valeur>] = nouveau <type> [<valeur>];

- **Initialiser un tableau sans valeurs:**

  <nom> = nouveau <type> [<longueur>];

- **Initialiser un tableau avec des valeurs:**

  <nom> = nouveau <type> [] {<valeur>, <valeur>, <valeur>, ...};

- **Initialisation d'un tableau à deux dimensions avec des valeurs:**

  <nom> = nouveau <type> [,] {{<valeur>, <valeur>}, {<valeur>, <valeur>}, ...};

- **Accéder à un élément à l'index i:**

  <nom> [i]

- **Obtenir la longueur du tableau:**

  <name> .Length

Remarques

En C #, un tableau est un type de référence, ce qui signifie qu'il est *nullable* .

Un tableau a une longueur fixe, ce qui signifie que vous ne pouvez pas y `.Add()` ou `.Remove()` . Pour les utiliser, vous avez besoin d'un tableau dynamique - `List` ou `ArrayList` .

Examples

Covariance de tableau

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

Cette conversion n'est pas sécurisée. Le code suivant déclenchera une exception d'exécution:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0];   // would have been bad if above assignment had succeeded
```

Obtenir et définir des valeurs de tableau

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

Déclarer un tableau

Un tableau peut être déclaré et rempli avec la valeur par défaut en utilisant la syntaxe d'initialisation entre crochets (`[]`). Par exemple, créer un tableau de 10 nombres entiers:

```
int[] arr = new int[10];
```

Les indices en C # sont basés sur zéro. Les indices du tableau ci-dessus seront 0-9. Par exemple:

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Ce qui signifie que le système commence à compter l'index de l'élément à partir de 0. De plus, les accès aux éléments des tableaux se font à **temps constant** . Cela signifie que l'accès au premier élément du tableau a le même coût (en temps) d'accès au deuxième élément, au troisième élément, etc.

Vous pouvez également déclarer une référence à un tableau sans instancier un tableau.

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

Un tableau peut également être créé et initialisé avec des valeurs personnalisées à l'aide de la

syntaxe d'initialisation de collection:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

La `new int[]` peut être omise lors de la déclaration d'une variable de tableau. Il ne s'agit pas d'une *expression* autonome, donc l'utiliser dans le cadre d'un appel différent ne fonctionne pas (pour cela, utilisez la version avec `new`):

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

Tableaux tapés implicitement

Alternativement, en combinaison avec le mot-clé `var` , le type spécifique peut être omis pour que le type du tableau soit déduit:

```
// same as int[]
var arr = new [] { 1, 2, 3 };
// same as string[]
var arr = new [] { "one", "two", "three" };
// same as double[]
var arr = new [] { 1.0, 2.0, 3.0 };
```

Itérer sur un tableau

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

en utilisant foreach:

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

Utiliser un accès non sécurisé avec des pointeurs https://msdn.microsoft.com/en-ca/library/y31yhkeb.aspx

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
```

```
            pInt++;// move pointer to next element
        }
    }
}
```

Sortie:

1
6
3
3
9

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. L'exemple suivant crée un tableau à deux dimensions de dix lignes et dix colonnes:

```
int[,] arr = new int[10, 10];
```

Un tableau de trois dimensions:

```
int[,,] arr = new int[10, 10, 10];
```

Vous pouvez également initialiser le tableau lors de la déclaration:

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]);  // 4
```

Tableaux dentelés

Les tableaux dentelés sont des tableaux qui, à la place des types primitifs, contiennent des tableaux (ou d'autres collections). C'est comme un tableau de tableaux - chaque élément du tableau contient un autre tableau.

Ils sont similaires aux tableaux multidimensionnels, mais ont une légère différence: comme les tableaux multidimensionnels sont limités à un nombre fixe de lignes et de colonnes, avec des tableaux irréguliers, chaque ligne peut avoir un nombre de colonnes différent.

Déclarer un tableau irrégulier

Par exemple, en déclarant un tableau irrégulier avec 8 colonnes:

```
int[][] a = new int[8][];
```

Le second `[]` est initialisé sans numéro. Pour initialiser les sous-tableaux, vous devez le faire

séparément:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Obtenir / définir des valeurs

Maintenant, obtenir l'une des sous-réseaux est facile. Imprimons tous les numéros de la 3ème colonne d' `a` :

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Obtenir une valeur spécifique:

```
a[<row_number>][<column_number>]
```

Définir une valeur spécifique:

```
a[<row_number>][<column_number>] = <value>
```

N'oubliez pas : il est toujours recommandé d'utiliser des tableaux irréguliers (tableaux de tableaux) plutôt que des tableaux multidimensionnels (matrices). C'est plus rapide et plus sûr à utiliser.

Note sur l'ordre des crochets

Considérons un tableau tridimensionnel de tableaux à cinq dimensions de tableaux à une dimension de `int` . Ceci est écrit en C # comme:

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

Dans le système de type CLR, la convention pour l'ordre des parenthèses est inversée, donc avec l' `arr` ci-dessus, nous avons:

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

et également:

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

Vérifier si un tableau contient un autre tableau

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// Échantillon

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

Initialisation d'un tableau rempli avec une valeur non par défaut répétée

Comme nous le savons, nous pouvons déclarer un tableau avec des valeurs par défaut:

```
int[] arr = new int[10];
```

Cela va créer un tableau de 10 entiers avec chaque élément du tableau ayant la valeur `0` (la valeur par défaut de type `int`).

Pour créer un tableau initialisé avec une valeur autre que celle par défaut, nous pouvons utiliser `Enumerable.Repeat`

partir de l' `System.Linq` noms `System.Linq` :

1. Pour créer un tableau `bool` de taille 10 rempli avec **"true"**

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. Pour créer un tableau `int` de taille 5 rempli de **"100"**

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. Pour créer un tableau de `string` de taille 5 rempli de **"C #"**

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Copier des tableaux

Copier un tableau partiel avec la `Array.Copy()` statique `Array.Copy()` , en commençant à l'index 0 dans la source et la destination:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);

// destinationArray will have 11,12 and 3
```

Copier l'intégralité du tableau avec la méthode d'instance `CopyTo()` , en commençant à l'index 0 de la source et de l'index spécifié dans la destination:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone` est utilisé pour créer une copie d'un objet tableau.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

`CopyTo` et `Clone` effectuent tous deux une copie superficielle, ce qui signifie que le contenu contient des références au même objet que les éléments du tableau d'origine.

Créer un tableau de numéros séquentiels

LINQ fournit une méthode qui facilite la création d'une collection remplie de numéros séquentiels. Par exemple, vous pouvez déclarer un tableau contenant les entiers compris entre 1 et 100.

La méthode `Enumerable.Range` nous permet de créer une séquence de nombres entiers à partir d'une position de départ spécifiée et d'un nombre d'éléments.

La méthode prend deux arguments: la valeur de départ et le nombre d'éléments à générer.

```
Enumerable.Range(int start, int count)
```

Notez que le `count` ne peut pas être négatif.

Usage:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

Cela générera un tableau contenant les nombres 1 à 100 (`[1, 2, 3, ..., 98, 99, 100]`).

Étant donné que la méthode `Range` renvoie un `IEnumerable<int>` , nous pouvons utiliser d'autres méthodes LINQ:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

Cela générera un tableau contenant 10 carrés entiers à partir de `4` : `[4, 9, 16, ..., 100, 121]` .

Comparer les tableaux pour l'égalité

LINQ fournit une fonction intégrée pour vérifier l'égalité de deux `IEnumerable` s et cette fonction peut être utilisée sur des tableaux.

La fonction `SequenceEqual` renvoie `true` si les tableaux ont la même longueur et que les valeurs des index correspondants sont égales, et `false` sinon.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

Cela va imprimer:

```
Arrays equal? True
```

Tableaux en tant qu'instances IEnumerable <>

Tous les tableaux implémentent l'interface `IList` non générique (et donc les interfaces de base `ICollection` et `IEnumerable` non génériques).

Plus important encore, les tableaux unidimensionnels implémentent les interfaces génériques `IList<>` et `IReadOnlyList<>` (et leurs interfaces de base) pour le type de données qu'ils contiennent. Cela signifie qu'ils peuvent être traités comme des types énumérables génériques et transmis à

une variété de méthodes sans avoir à les convertir d'abord en une forme non-tableau.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

Après avoir exécuté ce code, la liste `listOfIntegers` contiendra une `List<int>` contenant les valeurs 3, 5 et 7.

Le support `IEnumerable<>` signifie que les tableaux peuvent être interrogés avec LINQ, par exemple `arr1.Select(i => 10 * i)` .

Lire Tableaux en ligne: https://riptutorial.com/fr/csharp/topic/1429/tableaux

Chapitre 150: Tuples

Examples

Créer des tuples

Les tuples sont créés en utilisant les types génériques `Tuple<T1>` - `Tuple<T1,T2,T3,T4,T5,T6,T7,T8>` . Chacun des types représente un tuple contenant 1 à 8 éléments. Les éléments peuvent être de différents types.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Les tuples peuvent également être créés à l' `Tuple.Create` méthodes statiques `Tuple.Create` . Dans ce cas, les types des éléments sont déduits par le compilateur C #.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7.0

Depuis C # 7.0, Tuples peut être facilement créé en utilisant ValueTuple .

```
var tuple = ("foo", 123, true, new MyClass());
```

Les éléments peuvent être nommés pour une décomposition plus facile.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Accès aux éléments de tuple

Pour accéder aux éléments de tuple utilisent `Item1` - `Item8` propriétés. Seules les propriétés ayant un numéro d'index inférieur ou égal à la taille du tuple seront disponibles (c'est-à-dire qu'il est impossible d'accéder à la propriété `Item3` dans `Item3 Tuple<T1,T2>`).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new My Class()
```

Comparer et trier les tuples

Les tuples peuvent être comparés en fonction de leurs éléments.

A titre d'exemple, un enumerable dont les éléments sont de type `Tuple` peut être trié en fonction

d'opérateurs de comparaison définis sur un élément spécifié:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

Ou pour inverser l'utilisation du tri:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Renvoie plusieurs valeurs d'une méthode

Les tuples peuvent être utilisés pour renvoyer plusieurs valeurs d'une méthode sans utiliser de paramètres. Dans l'exemple suivant, `AddMultiply` est utilisé pour renvoyer deux valeurs (somme, produit).

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

Sortie:

53
700

Maintenant, C # 7.0 offre une autre façon de renvoyer plusieurs valeurs à partir de méthodes en utilisant des tuples de valeurs. Plus d'infos sur la structure `ValueTuple` .

Lire Tuples en ligne: https://riptutorial.com/fr/csharp/topic/838/tuples

Chapitre 151: Type de valeur vs type de référence

Syntaxe

- Passage par référence: public void Double (ref int numberToDouble) {}

Remarques

introduction

Types de valeur

Les types de valeur sont les plus simples des deux. Les types de valeur sont souvent utilisés pour représenter les données elles-mêmes. Un entier, un booléen ou un point dans un espace 3D sont tous des exemples de bons types de valeur.

Les types de valeur (structs) sont déclarés à l'aide du mot-clé struct. Voir la section syntaxe pour un exemple de comment déclarer une nouvelle structure.

De manière générale, nous avons 2 mots-clés utilisés pour déclarer les types de valeur:

- Structs
- Énumérations

Types de référence

Les types de référence sont légèrement plus complexes. Les types de référence sont des objets traditionnels au sens de la programmation orientée objet. Ainsi, ils prennent en charge l'héritage (et les avantages de celui-ci) et prennent également en charge les finaliseurs.

En C #, nous avons généralement ces types de référence:

- Des classes
- Les délégués
- Interfaces

Les nouveaux types de référence (classes) sont déclarés à l'aide du mot-clé class. Pour un exemple, voir la section sur la syntaxe pour savoir comment déclarer un nouveau type de référence.

Différences majeures

Les principales différences entre les types de référence et les types de valeur peuvent être vues

ci-dessous.

Des types de valeur existent sur la pile, des types de référence existent sur le tas

C'est la différence souvent mentionnée entre les deux, mais en réalité, lorsque vous utilisez un type de valeur en C #, tel qu'un int, le programme utilisera cette variable pour faire directement référence à cette valeur. Si vous dites int mine = 0, alors la variable mine fait directement référence à 0, ce qui est efficace. Cependant, les types de référence contiennent en réalité (comme leur nom l'indique) une référence à l'objet sous-jacent, ce qui s'apparente aux pointeurs dans d'autres langages tels que C ++.

Vous pourriez ne pas remarquer les effets de ceci immédiatement, mais les effets sont là, sont puissants et subtils. Voir l'exemple sur la modification des types de référence ailleurs pour un exemple.

Cette différence est la principale raison des différences suivantes et mérite d'être connue.

Les types de valeur ne changent pas lorsque vous les modifiez dans une méthode, les types de référence le font

Lorsqu'un type de valeur est transmis dans une méthode en tant que paramètre, si la méthode modifie la valeur de quelque manière que ce soit, la valeur n'est pas modifiée. En revanche, le passage d'un type de référence à cette même méthode et sa modification modifieront l'objet sous-jacent. d'autres objets utilisant ce même objet auront l'objet nouvellement modifié plutôt que leur valeur d'origine.

Voir l'exemple des types de valeur vs types de référence dans les méthodes pour plus d'informations.

Et si je veux les changer?

Transmettez-les simplement dans votre méthode en utilisant le mot-clé "ref" et vous transmettez ensuite cet objet par référence. Ce qui signifie que c'est le même objet en mémoire. Les modifications que vous apporterez seront donc respectées. Voir l'exemple sur le passage par référence pour un exemple.

Les types de valeur ne peuvent pas être null, les types de référence peuvent

Tout comme il est dit, vous pouvez attribuer la valeur NULL à un type de référence, ce qui signifie que la variable que vous avez affectée ne peut pas être associée à un objet réel. Dans le cas des types de valeur, cela n'est toutefois pas possible. Vous pouvez, cependant, utiliser Nullable, pour que votre type de valeur soit nullable, si cela est une exigence, mais si c'est quelque chose que vous envisagez, pensez fermement si une classe peut ne pas être la meilleure approche ici, si c'est la vôtre type.

Examples

Changer les valeurs ailleurs

```
public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

Vous remarquerez que même si la liste printingList a été créée avant les corrections apportées aux noms d'élèves après les fautes de frappe, la méthode PrintPrintingList affiche toujours les noms corrigés:

```
Scott Duke
Vincent Kong
Craig Brett
```

En effet, les deux listes contiennent une liste de références aux mêmes étudiants. SO modifier l'objet objet sous-jacent propage aux utilisations par l'une ou l'autre liste.

Voici à quoi ressemblerait la classe d'élèves.

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

En passant par référence

Si vous souhaitez que les types de valeurs et les types de référence dans les méthodes fonctionnent correctement, utilisez le mot-clé ref dans la signature de votre méthode pour le paramètre que vous souhaitez transmettre par référence, ainsi que lorsque vous appelez la méthode.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Faire ces modifications rendrait le numéro mis à jour comme prévu, ce qui signifie que la sortie de la console pour le numéro serait 8.

Passage par référence en utilisant le mot-clé ref.

De la documentation :

> En C #, les arguments peuvent être transmis aux paramètres par valeur ou par référence. Le passage par référence permet aux membres de la fonction, aux méthodes, aux propriétés, aux indexeurs, aux opérateurs et aux constructeurs de modifier la valeur des paramètres et de conserver cette modification dans l'environnement appelant. Pour transmettre un paramètre par référence, utilisez le mot-clé `ref` ou `out` .

La différence entre `ref` et `out` est que `out` signifie que le paramètre passé doit être assigné avant que la fonction ends.in change les paramètres passés avec `ref` peuvent être modifiés ou laissés inchangés.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
```

```
        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Sortie :

```
Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7
```

Affectation

```
var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well
```

L'affectation à une variable d'une `List<int>` ne crée pas une copie de la `List<int>` . Au lieu de cela, il copie la référence à la `List<int>` . Nous appelons les types qui se comportent de cette façon *les types de référence* .

Différence avec les paramètres de méthode réf et out

Il existe deux manières possibles de passer un type de valeur par référence: `ref` et `out` . La différence est que, en le passant avec `ref` la valeur doit être initialisée mais pas lors du passage

avec `out` . L'utilisation de `out` garantit que la variable a une valeur après l'appel de la méthode:

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
    value += 4 // CS0269: Use of unassigned out parameter `value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter
`value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;   // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165  Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

Le hic est que l'aide `out` le paramètre `must` être initialisé avant de quitter la méthode, donc la méthode suivante est possible `ref` mais pas `out` :

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current
method
```

En effet, si la `condition` ne tient pas, la `value` ne sera pas attribuée.

paramètres de réf vs out

Code

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the
method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Sortie

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30
```

```
Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

Lire Type de valeur vs type de référence en ligne: https://riptutorial.com/fr/csharp/topic/3014/type-de-valeur-vs-type-de-reference

Chapitre 152: Type dynamique

Remarques

Le mot clé `dynamic` déclare une variable dont le type n'est pas connu à la compilation. Une variable `dynamic` peut contenir n'importe quelle valeur et le type de la valeur peut changer au cours de l'exécution.

Comme indiqué dans le livre "Métaprogrammation en .NET", C # n'a pas de type de support pour le mot clé `dynamic` :

> La fonctionnalité activée par le mot clé `dynamic` est un ensemble intelligent d'actions de compilation qui émettent et utilisent des objets `CallSite` dans le conteneur de site de la portée d'exécution locale. Le compilateur gère ce que les programmeurs perçoivent comme des références d'objets dynamiques via ces instances `CallSite` . Les paramètres, les types de retour, les champs et les propriétés qui reçoivent un traitement dynamique au moment de la compilation peuvent être marqués de certaines métadonnées pour indiquer qu'ils ont été générés pour une utilisation dynamique, mais le type de données sous-jacent sera toujours `System.Object` .

Examples

Créer une variable dynamique

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

Retour dynamique

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
```

```
}
```

Créer un objet dynamique avec des propriétés

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

Gestion de types spécifiques inconnus à la compilation

Les résultats équivalents suivants en sortie:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

```
class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

L'avantage de la dynamique, l'ajout d'un nouveau type à gérer nécessite simplement l'ajout d'une surcharge de DebugToStringInternal du nouveau type. Élimine également le besoin de le lancer manuellement au type.

Lire Type dynamique en ligne: https://riptutorial.com/fr/csharp/topic/762/type-dynamique

Chapitre 153: Types anonymes

Examples

Créer un type anonyme

Les types anonymes n'étant pas nommés, les variables de ces types doivent être implicitement typées (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

Si les noms de membres ne sont pas spécifiés, ils sont définis sur le nom de la propriété / variable utilisée pour initialiser l'objet.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Notez que les noms ne peuvent être omis que lorsque l'expression dans la déclaration de type anonyme est un accès de propriété simple; pour les appels de méthode ou les expressions plus complexes, un nom de propriété doit être spécifié.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

Anonyme vs dynamique

Les types anonymes permettent la création d'objets sans avoir à définir explicitement leurs types à l'avance, tout en conservant une vérification de type statique.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Inversement, `dynamic` a la vérification de type dynamique, optant pour des erreurs d'exécution, au lieu d'erreurs de compilation.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Méthodes génériques avec types anonymes

Les méthodes génériques permettent l'utilisation de types anonymes via l'inférence de type.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

Cela signifie que les expressions LINQ peuvent être utilisées avec des types anonymes:

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

Instancier des types génériques avec des types anonymes

L'utilisation de constructeurs génériques nécessiterait que les types anonymes soient nommés, ce qui n'est pas possible. Alternativement, des méthodes génériques peuvent être utilisées pour permettre l'inférence de type.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

Dans le cas de `List<T>` , les tableaux typés implicitement peuvent être convertis en une `List<T>` via la méthode `ToList` LINQ:

```
var list2 = new[] {anon, anon2}.ToList();
```

Égalité de type anonyme

L'égalité de type anonyme est donnée par la méthode d'instance `Equals` . Deux objets sont égaux s'ils ont le même type et les mêmes valeurs (via `a.Prop.Equals(b.Prop)`) pour chaque propriété.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Deux types anonymes sont considérés comme identiques si et seulement si leurs propriétés portent le même nom et le même type et apparaissent dans le même ordre.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have diferent types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

Tableaux tapés implicitement

Des tableaux de types anonymes peuvent être créés avec un typage implicite.

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

Lire Types anonymes en ligne: https://riptutorial.com/fr/csharp/topic/765/types-anonymes

Chapitre 154: Types intégrés

Examples

Type de référence immuable - chaîne

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

Type de valeur - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

Type de valeur - short, int, long (entiers signés 16 bits, 32 bits, 64 bits)

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;

// assigning a signed long to its minimum value (note the long postfix)
long l = -9223372036854775808L;
```

```
// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

Il est également possible de rendre ces types nullables, ce qui signifie que, en plus des valeurs habituelles, null peut également être affecté. Si une variable de type nullable n'est pas initialisée, elle sera nulle au lieu de 0. Les types nullables sont marqués en ajoutant un point d'interrogation (?) Après le type.

```
int a; //This is now 0.
int? b; //This is now null.
```

Type de valeur - ushort, uint, ulong (entiers 16 bits non signés, 32 bits, 64 bits)

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;

// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

Il est également possible de rendre ces types nullables, ce qui signifie que, en plus des valeurs habituelles, null peut également être affecté. Si une variable de type nullable n'est pas initialisée, elle sera nulle au lieu de 0. Les types nullables sont marqués en ajoutant un point d'interrogation (?) Après le type.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Type de valeur - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

Le mot clé bool est un alias de System.Boolean. Il est utilisé pour déclarer des variables pour stocker les valeurs booléennes, `true` et `false` .

Comparaisons avec les types de valeur encadrés

Si des types de valeur sont affectés à des variables de type `object` ils sont *encadrés* - la valeur est stockée dans une instance de `System.Object` . Cela peut entraîner des conséquences imprévues lors de la comparaison de valeurs avec `==` , par exemple:

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

Cela peut être évité en utilisant la méthode `Equals` surchargée, qui donnera le résultat attendu.

```
var comparison2 = left.Equals(right); // true
```

Alternativement, on pourrait faire la même chose en désencapsulant les variables `left` et `right` afin de comparer les valeurs `int` :

```
var comparison3 = (int)left == (int)right; // true
```

Conversion de types de valeur encadrés

Les types de valeur encadrés ne peuvent être déballés que dans leur `Type` origine, même si une conversion des deux `Type` est valide, par exemple:

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

Cela peut être évité en déballant d'abord dans le `Type` origine, par exemple:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Lire Types intégrés en ligne: https://riptutorial.com/fr/csharp/topic/42/types-integres

Chapitre 155: Types nullables

Syntaxe

- `Nullable<int> i = 10;`
- int? j = 11;
- int? k = null;
- DateTime? DateOfBirth = DateTime.Now;
- décimal? Quantité = 1,0 m;
- bool? IsAvailable = true;
- carboniser? Lettre = 'a';
- (type)? Nom de variable

Remarques

Les types nullables peuvent représenter toutes les valeurs d'un type sous-jacent, ainsi que `null` .

La syntaxe `T?` est un raccourci pour `Nullable<T>`

Les valeurs nullables sont en `System.ValueType` objets `System.ValueType` , elles peuvent donc être encapsulées ou non. En outre, la valeur `null` d'un objet nullable n'est pas la même que la valeur `null` d'un objet de référence, c'est simplement un indicateur.

Lors de la mise en boîte d'un objet nullable, la valeur null est convertie en référence `null` et la valeur non null est convertie en type sous-jacent non nullable.

```
DateTime? dt = null;
var o = (object)dt;
var result = (o == null); // is true

DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var dt2 = (DateTime)dt; // correct cause o contains DateTime value
```

La seconde règle conduit à un code correct mais paradoxal:

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

En bref:

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

Examples

Initialiser un nullable

Pour `null` valeurs `null` :

```
Nullable<int> i = null;
```

Ou:

```
int? i = null;
```

Ou:

```
var i = (int?)null;
```

Pour les valeurs non nulles:

```
Nullable<int> i = 0;
```

Ou:

```
int? i = 0;
```

Vérifier si un Nullable a une valeur

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Ce qui est le même que:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Récupère la valeur d'un type nullable

Donné après nullable `int`

```
int? i = 10;
```

Si une valeur par défaut est nécessaire, vous pouvez en affecter un en utilisant un opérateur de coalescence null , la méthode `GetValueOrDefault` ou vérifier si `HasValue` int `HasValue` avant l'affectation.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

L'utilisation suivante est toujours *dangereuse* . Si `i` est nul à l'exécution, une `System.InvalidOperationException` sera lancée. Au moment de la conception, si une valeur n'est pas définie, vous obtiendrez une erreur d' `Use of unassigned local variable 'i'` .

```
int j = i.Value;
```

Obtenir une valeur par défaut à partir d'un nullable

La méthode `.GetValueOrDefault()` renvoie une valeur même si la propriété `.HasValue` est `.HasValue` false (contrairement à la propriété Value, qui génère une exception).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

Sortie:

```
0
1
```

Vérifier si un paramètre de type générique est un type nullable

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

La valeur par défaut des types nullables est null

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Sortie:

nul

Utilisation efficace des Nullables sous-jacents argument

Tout type nullable est un type **générique** . Et tout type nullable est un type de **valeur** .

Il y a quelques astuces qui permettent d' **utiliser efficacement** le résultat de la méthode Nullable.GetUnderlyingType lors de la création de code lié à des fins de réflexion / génération de code:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

L'usage:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsNullable())
```

```
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsNullable(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Sortie:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS Le `NullableTypesCache` est défini comme suit:

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type,
Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

Lire Types nullables en ligne: https://riptutorial.com/fr/csharp/topic/1240/types-nullables

Chapitre 156: Un aperçu des collections c

Examples

HashSet

Ceci est une collection d'éléments uniques, avec la recherche O (1).

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

À titre de comparaison, le fait d' `Contains` une liste contenue produit des performances moins bonnes:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains` utilise une table de hachage, de sorte que les recherches sont extrêmement rapides, quel que soit le nombre d'éléments de la collection.

SortedSet

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

T [] (Tableau de T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
}
```

```
// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";


// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two
```

liste

`List<T>` est une liste d'un type donné. Les éléments peuvent être ajoutés, insérés, supprimés et adressés par index.

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]);    // 5
```

`List<T>` peut être considéré comme un tableau que vous pouvez redimensionner. L'énumération de la collection dans l'ordre est rapide, tout comme l'accès à des éléments individuels via leur index. Pour accéder à des éléments en fonction d'un aspect de leur valeur ou d'une autre clé, un `Dictionary<T>` fournira une recherche plus rapide.

dictionnaire

Dictionary <TKey, TValue> est une carte. Pour une clé donnée, il peut y avoir une valeur dans le dictionnaire.

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException
```

```
int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

Clé en double lors de l'initialisation de la collection

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

Empiler

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
Console.WriteLine(stack.Pop()); // prints 3
```

LinkedList

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

Notez que `LinkedList<T>` représente la liste *doublement* liée. Donc, c'est simplement une collection de nœuds et chaque nœud contient un élément de type `T` Chaque nœud est lié au nœud précédent et au nœud suivant.

Queue

```
// Initalize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

> Thread Safe Head Up! Utilisez ConcurrentQueue dans des environnements multi-thread.

Lire Un aperçu des collections c # en ligne: https://riptutorial.com/fr/csharp/topic/2344/un-apercu-des-collections-c-sharp

Chapitre 157: Utiliser json.net

Introduction

Utilisation de la classe JSON.net JsonConverter .

Examples

Utiliser JsonConverter sur des valeurs simples

Exemple d'utilisation de JsonCoverter pour désérialiser la propriété d'exécution de la réponse api dans un objet Timespan dans le modèle Movies

JSON (http://www.omdbapi.com/?i=tt1663662)

```
{
    Title: "Pacific Rim",
    Year: "2013",
    Rated: "PG-13",
    Released: "12 Jul 2013",
    Runtime: "131 min",
    Genre: "Action, Adventure, Sci-Fi",
    Director: "Guillermo del Toro",
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham
(story)",
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and
a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to
save the world from the apocalypse.",
    Language: "English, Japanese, Cantonese, Mandarin",
    Country: "USA",
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",
    Poster: "https://images-na.ssl-images-
amazon.com/images/M/MV5BMTY3MTI5NjQ4Nl5BMl5BanBnXkFtZTcwOTU1OTU0OQ@@._V1_SX300.jpg",
    Ratings: [{
            Source: "Internet Movie Database",
            Value: "7.0/10"
        },
        {
            Source: "Rotten Tomatoes",
            Value: "71%"
        },
        {
            Source: "Metacritic",
            Value: "64/100"
        }
    ],
    Metascore: "64",
    imdbRating: "7.0",
```

```
    imdbVotes: "398,198",
    imdbID: "tt1663662",
    Type: "movie",
    DVD: "15 Oct 2013",
    BoxOffice: "$101,785,482.00",
    Production: "Warner Bros. Pictures",
    Website: "http://pacificrimmovie.com",
    Response: "True"
}
```

Modèle de film

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

RuntimeSerializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
```

```
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}
```

L'appel

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

Collecte tous les champs de l'objet JSON

```
using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
```

```
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}
```

Usage:

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");
```

Démo
Pour cet objet JSON

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

Le résultat attendu sera:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

Lire Utiliser json.net en ligne: https://riptutorial.com/fr/csharp/topic/9879/utiliser-json-net

Chapitre 158: Utiliser la déclaration

Introduction

Fournit une syntaxe pratique qui garantit l'utilisation correcte des objets IDisposables .

Syntaxe

- en utilisant (jetable) {}
- using (IDisposable disposable = new MyDisposable ()) {}

Remarques

L'objet dans l'instruction `using` doit implémenter l'interface `IDisposable` .

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

Des exemples plus complets d'implémentation `IDisposable` peuvent être trouvés dans la documentation MSDN .

Examples

Utilisation des principes de base de l'instruction

`using` du sucre syntaxique vous permet de garantir qu'une ressource est nettoyée sans avoir besoin d'un bloc `try-finally` explicite. Cela signifie que votre code sera beaucoup plus propre et que vous ne perdrez pas de ressources non gérées.

Standard `Dispose` motif de nettoyage, pour les objets qui mettent en œuvre l' `IDisposable` interface (dont le `FileStream` classe de base de `Stream` fait en .NET):

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;
```

```
        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

`using` simplifie votre syntaxe en masquant le `try-finally` explicite:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

Tout comme `finally` blocs s'exécutent toujours indépendamment des erreurs ou des retours, en `using` toujours les appels `Dispose()` , même en cas d'erreur:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}
```

Remarque: L' appel de `Dispose` étant garanti quel que soit le flux de code, assurez-vous que `Dispose` ne lève jamais une exception lorsque vous implémentez `IDisposable` . Sinon, une exception réelle serait remplacée par la nouvelle exception, ce qui entraînerait un cauchemar de débogage.

Retourner de l'utilisation du bloc

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

En raison de la sémantique de `try..finally` à laquelle le bloc `using` traduit, l'instruction `return`

fonctionne comme prévu - la valeur de retour est évaluée avant que `finally` bloc n'est exécuté et que la valeur ne soit éliminée. L'ordre d'évaluation est le suivant:

1. Evaluer le corps d' `try`
2. Évaluer et mettre en cache la valeur renvoyée
3. Exécuter enfin le bloc
4. Renvoie la valeur de retour mise en cache

Cependant, vous ne pouvez pas retourner la variable `disposable` lui - même, car elle non valide est disposé référence - voir exemple connexe .

Instructions d'utilisation multiples avec un bloc

Il est possible d'utiliser plusieurs imbriquées en `using` des déclarations sans ajout de plusieurs niveaux d'accolades imbriquées. Par exemple:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

Une alternative est d'écrire:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Ce qui est exactement équivalent au premier exemple.

Note: Nichée en `using` des déclarations peuvent déclencher la règle d' analyse du code Microsoft CS2002 (voir cette réponse pour la clarification) et générer un avertissement. Comme expliqué dans la réponse liée, il est généralement préférable de nidifier à l' `using` instructions.

Lorsque les types de l'instruction `using` sont du même type, vous pouvez les délimiter par des virgules et spécifier le type une seule fois (bien que cela soit rare):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

Cela peut également être utilisé lorsque les types ont une hiérarchie partagée:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

Le mot-clé `var` *ne peut pas* être utilisé dans l'exemple ci-dessus. Une erreur de compilation se produirait. Même la déclaration séparée par des virgules ne fonctionnera pas lorsque les variables déclarées ont des types de différentes hiérarchies.

Gotcha: retourner la ressource que vous disposez

Ce qui suit est une mauvaise idée car cela éliminerait la variable de `db` avant de la renvoyer.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

Cela peut également créer des erreurs plus subtiles:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

Cela semble correct, mais le problème est que l'évaluation de l'expression LINQ est paresseuse et ne sera éventuellement exécutée que lorsque le `DBContext` sous-jacent aura déjà été supprimé.

Donc en bref, l'expression n'est pas évaluée avant de quitter l' `using` . Une solution possible à ce problème, qui fait encore l' utilisation de l' `using` , est de provoquer l'expression d'évaluer immédiatement en appelant une méthode qui énumérera le résultat. Par exemple `ToList()` , `ToArray()` , etc. Si vous utilisez la version la plus récente d'Entity Framework, vous pouvez utiliser les contreparties `async` telles que `ToListAsync()` ou `ToArrayAsync()` .

Vous trouverez ci-dessous l'exemple en action:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

Il est important de noter qu'en appelant `ToList()` ou `ToArray()` , l'expression sera évaluée avec empressement, ce qui signifie que toutes les personnes ayant l'âge spécifié seront chargées en mémoire même si vous ne les parcourez pas.

L'utilisation d'instructions est null-safe

Vous n'avez pas à vérifier l'objet `IDisposable` pour `null` . `using` ne lancera pas une exception et `Dispose()` ne sera pas appelé:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Gotcha: Exception dans la méthode Dispose masquant d'autres erreurs dans Utilisation des blocs

Considérons le bloc de code suivant.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

Vous pouvez vous attendre à voir "Impossible d'effectuer l'opération" imprimé sur la console mais vous verriez réellement "Impossible de s'en débarrasser". comme la méthode Dispose est encore appelée même après la première exception.

Cela vaut la peine d’être conscient de cette subtilité, car cela peut masquer la véritable erreur qui a empêché la destruction de l’objet et le rendre plus difficile à déboguer.

Utilisation des instructions et des connexions à la base de données

Le mot-clé `using` garantit que la ressource définie dans l'instruction existe uniquement dans la portée de l'instruction elle-même. Toutes les ressources définies dans l'instruction doivent implémenter l'interface `IDisposable` .

Celles-ci sont extrêmement importantes pour les connexions qui implémentent l'interface `IDisposable` , car elles permettent de garantir que les connexions sont non seulement correctement fermées mais que leurs ressources sont libérées après que l'instruction `using` est hors de portée.

Classes de données communes `IDisposable`

Bon nombre des éléments suivants sont des classes liées aux données qui implémentent l'interface `IDisposable` et sont des candidats parfaits pour une instruction `using` :

- `SqlConnection` , `SqlCommand` , `SqlDataReader` , etc.
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` , etc.
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` , etc.
- `DbContext`

Tous ces éléments sont couramment utilisés pour accéder aux données via C # et seront couramment rencontrés lors de la création d'applications centrées sur les données. De nombreuses autres classes non mentionnées qui implémentent les mêmes `FooConnection` , `FooCommand` , `FooDataReader` peuvent se comporter de la même manière.

Modèle d'accès commun pour les connexions ADO.NET

Un modèle commun pouvant être utilisé pour accéder à vos données via une connexion ADO.NET peut se présenter comme suit:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

Ou si vous réalisiez une simple mise à jour et que vous n’avez pas besoin de lecteur, le même concept de base s’appliquerait:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

Utilisation des instructions avec DataContexts

De nombreux ORM, tels que Entity Framework, exposent des classes d'abstraction utilisées pour interagir avec les bases de données sous-jacentes sous la forme de classes telles que `DbContext` . Ces contextes implémentent généralement également l'interface `IDisposable` et devraient en tirer parti en `using` instructions lorsque cela est possible:

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

Utilisation de la syntaxe Dispose pour définir une étendue personnalisée

Pour certains cas d'utilisation, vous pouvez utiliser la syntaxe `using` pour définir une étendue personnalisée. Par exemple, vous pouvez définir la classe suivante pour exécuter du code dans une culture spécifique.

```
public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
```

Vous pouvez ensuite utiliser cette classe pour définir des blocs de code qui s'exécutent dans une culture spécifique.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
```

```
using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM
```

Note: comme nous n'utilisons pas l'instance de `CultureContext` nous créons, nous ne lui assignons pas de variable.

Cette technique est utilisée par l' assistant `BeginForm` dans ASP.NET MVC.

Exécuter du code dans un contexte de contrainte

Si vous avez du code (une *routine*) à exécuter dans un contexte spécifique (contrainte), vous pouvez utiliser l'injection de dépendance.

L'exemple suivant montre la contrainte d'exécution sous une connexion SSL ouverte. Cette première partie serait dans votre bibliothèque ou votre framework, que vous n'exposeriez pas au code client.

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
```

```
        }
    }
}
```

Maintenant, le code client qui veut faire quelque chose sous SSL mais ne veut pas gérer tous les détails SSL. Vous pouvez maintenant faire ce que vous voulez dans le tunnel SSL, par exemple échanger une clé symétrique:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

Vous exécutez cette routine comme suit:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

Pour ce faire, vous avez besoin de la clause `using()` car c'est le seul moyen (sauf un bloc `try..finally`) de garantir que le code client (`ExchangeSymmetricKey`) ne se ferme jamais sans éliminer correctement les ressources disponibles. Sans la clause `using()` , vous ne sauriez jamais si une routine peut briser la contrainte du contexte pour disposer de ces ressources.

Lire Utiliser la déclaration en ligne: https://riptutorial.com/fr/csharp/topic/38/utiliser-la-declaration

Chapitre 159: Utiliser la directive

Remarques

Le mot-clé `using` est à la fois une directive (cette rubrique) et une instruction.

Pour l'instruction `using` (c.-à-d. Pour encapsuler la portée d'un objet `IDisposable` , en veillant à ce qu'en dehors de cette portée, l'objet soit proprement éliminé), veuillez vous reporter à Utilisation de l'instruction .

Examples

Utilisation de base

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console;  //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

Référence à un espace de noms

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace.  i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Associer un alias à un espace de noms

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
//prefixing them with only the defined alias and not the full namespace.  i.e:

//...
var sb = new st.StringBuilder();
```

```
//instead of
var sb = new System.Text.StringBuilder();
```

Accéder aux membres statiques d'une classe

6,0

Vous permet d'importer un type spécifique et d'utiliser les membres statiques du type sans les qualifier avec le nom du type. Cela montre un exemple utilisant des méthodes statiques:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

Et cela montre un exemple utilisant des propriétés et des méthodes statiques:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Associer un alias pour résoudre des conflits

Si vous utilisez plusieurs espaces de noms pouvant comporter des classes de même nom (telles que `System.Random` et `UnityEngine.Random`), vous pouvez utiliser un alias pour spécifier que `Random` provient de l'un ou l'autre sans avoir à utiliser l'intégralité de l'espace de nommage dans l'appel. .

Par exemple:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

Cela empêchera le compilateur de déterminer quelle `Random` évaluer la nouvelle variable. Au lieu de cela, vous pouvez faire:

```
using UnityEngine;
using System;
```

```
using Random = System.Random;

Random rnd = new Random();
```

Cela ne vous empêche pas d'appeler l'autre par son espace de nom complet, comme ceci:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` sera une variable `System.Random` et `unityRandom` sera une variable `UnityEngine.Random` .

Utiliser des directives d'alias

Vous pouvez utiliser `using` afin de définir un alias pour un espace de noms ou un type. Plus de détails peuvent être trouvés ici .

Syntaxe:

```
using <identifier> = <namespace-or-type-name>;
```

Exemple:

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Lire Utiliser la directive en ligne: https://riptutorial.com/fr/csharp/topic/52/utiliser-la-directive

Chapitre 160: Utiliser SQLite en C

Examples

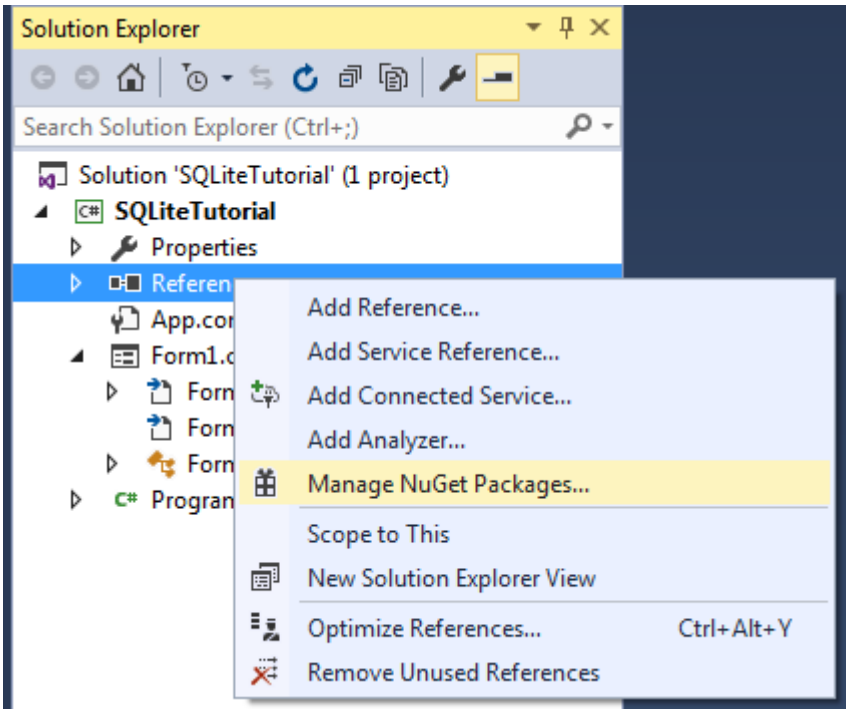
Créer un CRUD simple en utilisant SQLite en C

Tout d'abord, nous devons ajouter le support SQLite à notre application. Il y a deux façons de le faire

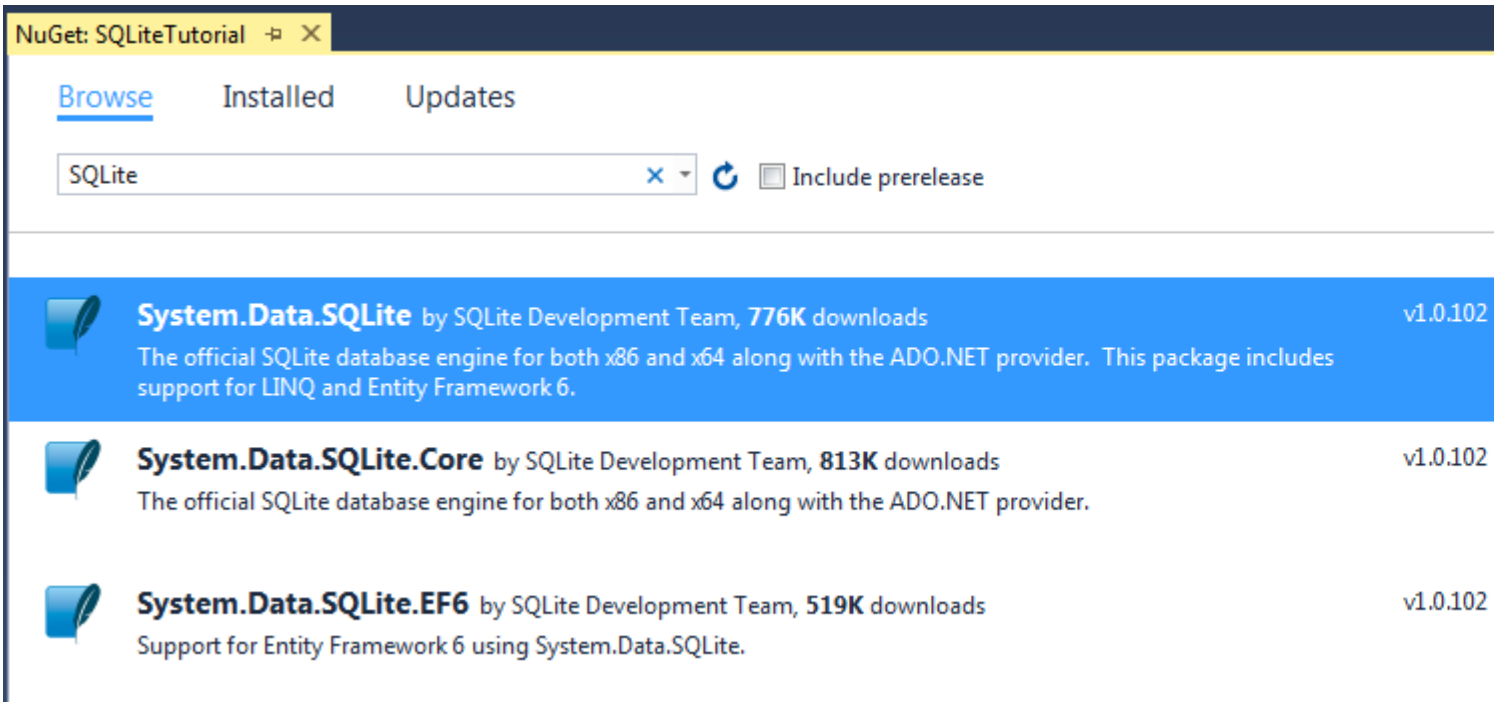
- Téléchargez la DLL correspondant à votre système depuis la page de téléchargement de SQLite , puis ajoutez-la manuellement au projet
- Ajouter une dépendance SQLite via NuGet

Nous allons le faire la deuxième façon

Ouvrez d'abord le menu NuGet



et recherchez **System.Data.SQLite** , sélectionnez-le et cliquez sur **Installer**



L'installation peut également être effectuée à partir de la console du gestionnaire de paquets avec

```
PM> Install-Package System.Data.SQLite
```

Ou uniquement pour les fonctionnalités de base

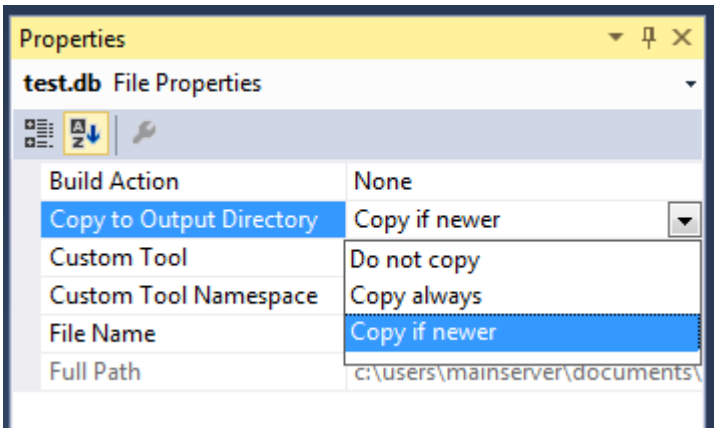
```
PM> Install-Package System.Data.SQLite.Core
```

C'est tout pour le téléchargement, donc nous pouvons aller directement dans le codage.

Commencez par créer une base de données SQLite simple avec cette table et ajoutez-la en tant que fichier au projet.

```
CREATE TABLE User(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL
);
```

N'oubliez pas non plus de définir la propriété **Copier dans le répertoire de sortie** du fichier sur **Copier si la version** de **Copy est toujours récente** , en fonction de vos besoins.



Créer une classe appelée User, qui sera l'entité de base de notre base de données

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

Nous allons écrire deux méthodes pour l'exécution des requêtes, la première pour l'insertion, la mise à jour ou la suppression de la base de données

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }

        return numberOfRowsAffected;
    }
}
```

et le second pour la lecture de la base de données

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;
```

```
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);

            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

Passons maintenant à nos méthodes **CRUD**

Ajouter un utilisateur

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName,
@lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Modification de l'utilisateur

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE
Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
```

```
}
```

Supprimer l'utilisateur

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}
```

Obtenir l'utilisateur par `Id`

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

Exécution de la requête

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
    {
       cmd.CommandText = "query";
       using (SqlDataReader dr = cmd.ExecuteReader())
       {
```

```
            while(dr.Read())
            {
                //do stuff
            }
        }
    }
}
```

Remarque : `FailIfMissing` vous `FailIfMissing` sur true, le fichier `data.db` est manquant. Cependant, le fichier sera vide. Donc, toutes les tables requises doivent être recréées.

Lire Utiliser SQLite en C # en ligne: https://riptutorial.com/fr/csharp/topic/4960/utiliser-sqlite-en-c-sharp

Chapitre 161: Vérifié et décoché

Syntaxe

- vérifié (a + b) // expression vérifiée
- décoché (a + b) // expression non vérifiée
- vérifié {c = a + b; c + = 5; } // bloc vérifié
- non cochée {c = a + b; c + = 5; } // bloc non vérifié

Examples

Vérifié et décoché

Les instructions C # s'exécutent dans un contexte vérifié ou non. Dans un contexte vérifié, le débordement arithmétique déclenche une exception. Dans un contexte non contrôlé, le dépassement arithmétique est ignoré et le résultat est tronqué.

```
short m = 32767;
short n = 32767;
int result1 =  checked((short)(m + n));   //will throw an OverflowException
int result2 =  unchecked((short)(m + n)); // will return -2
```

Si aucun de ces éléments n'est spécifié, le contexte par défaut reposera sur d'autres facteurs, tels que les options du compilateur.

Vérifié et décoché comme une portée

Les mots-clés peuvent également créer des étendues afin de (dé) vérifier plusieurs opérations.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

Lire Vérifié et décoché en ligne: https://riptutorial.com/fr/csharp/topic/2394/verifie-et-decoche

Chapitre 162: Windows Communication Foundation

Introduction

Windows Communication Foundation (WCF) est un cadre pour la création d'applications orientées services. En utilisant WCF, vous pouvez envoyer des données en tant que messages asynchrones d'un point de terminaison de service à un autre. Un point de terminaison de service peut faire partie d'un service disponible en permanence hébergé par IIS ou d'un service hébergé dans une application. Les messages peuvent être aussi simples qu'un seul caractère ou mot envoyé en XML, ou aussi complexe qu'un flux de données binaires.

Examples

Exemple de démarrage

Le service décrit les opérations qu'il effectue dans un contrat de service qu'il expose publiquement en tant que métadonnées.

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

L'implémentation du service calcule et renvoie le résultat approprié, comme indiqué dans l'exemple de code suivant.

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

Le service expose un noeud final pour communiquer avec le service, défini à l'aide d'un fichier de configuration (Web.config), comme illustré dans l'exemple de configuration suivant.

```
<services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
        <!-- ICalculator is exposed at the base address provided by
```

```
         host: http://localhost/servicemodelsamples/service.svc.  -->
       <endpoint address=""
              binding="wsHttpBinding"
              contract="StackOverflow.ServiceModel.Samples.ICalculator" />
       ...
    </service>
</services>
```

Le framework n'expose pas les métadonnées par défaut. En tant que tel, le service active le ServiceMetadataBehavior et expose un point de terminaison d'échange de métadonnées à l' adresse http: //localhost/servicemodelsamples/service.svc/mex . La configuration suivante illustre cela.

```
<system.serviceModel>
  <services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is explosed at
       http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
   attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Le client communique à l'aide d'un type de contrat donné en utilisant une classe client générée par l'outil de métadonnées ServiceModel (Svcutil.exe).

Exécutez la commande suivante à partir de l'invite de commande SDK dans le répertoire client pour générer le proxy typé:

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

Comme le service, le client utilise un fichier de configuration (App.config) pour spécifier le noeud final avec lequel il souhaite communiquer. La configuration du noeud final client consiste en une adresse absolue pour le noeud final de service, la liaison et le contrat, comme illustré dans l'exemple suivant.

```
<client>
```

```
    <endpoint
        address="http://localhost/servicemodelsamples/service.svc"
        binding="wsHttpBinding"
        contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

L'implémentation du client instancie le client et utilise l'interface typée pour commencer à communiquer avec le service, comme indiqué dans l'exemple de code suivant.

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

Lire Windows Communication Foundation en ligne:
https://riptutorial.com/fr/csharp/topic/10760/windows-communication-foundation

Chapitre 163: XDocument et l'espace de noms System.Xml.Linq

Examples

Générer un document XML

Le but est de générer le document XML suivant:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

Modifier un fichier XML

Pour modifier un fichier XML avec `XDocument` , vous devez charger le fichier dans une variable de type `XDocument` , le modifier en mémoire, puis l'enregistrer, en remplaçant le fichier d'origine. Une erreur courante consiste à modifier le XML en mémoire et à attendre que le fichier sur le disque

change.

Étant donné un fichier XML:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Vous voulez modifier la couleur de la banane en marron:

1. Nous devons connaître le chemin d'accès au fichier sur le disque.
2. Une surcharge de `XDocument.Load` reçoit un URI (chemin de fichier).
3. Étant donné que le fichier xml utilise un espace de noms, nous devons interroger avec le nom de l'élément ET de l'espace de noms.
4. Une requête Linq utilisant la syntaxe C # 6 pour tenir compte de la possibilité de valeurs nulles. Chaque `.` utilisé dans cette requête peut retourner un ensemble nul si la condition ne trouve aucun élément. Avant C # 6, vous le feriez en plusieurs étapes, en vérifiant la valeur null en cours de route. Le résultat est l'élément `<Fruit>` qui contient la banane. En fait, un `IEnumerable<XElement>` , c'est pourquoi l'étape suivante utilise `FirstOfDefault()` .
5. Maintenant, nous extrayons l'élément FruitColor de l'élément Fruit que nous venons de trouver. Ici, nous supposons qu'il n'y en a qu'un, ou que nous nous préoccupons uniquement du premier.
6. Si ce n'est pas nul, nous définissons FruitColor sur "Brown".
7. Enfin, nous sauvegardons le `XDocument` , en écrasant le fichier d'origine sur le disque.

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
    Elements(ns + "FruitName")?.
    Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
```

```
    elColor.Value = "Brown";
}

// 7.
xdoc.Save(xmlFilePath);
```

Le fichier ressemble maintenant à ceci:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Brown</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Générer un document XML en utilisant une syntaxe fluide

Objectif:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

Lire XDocument et l'espace de noms System.Xml.Linq en ligne:
https://riptutorial.com/fr/csharp/topic/1866/xdocument-et-l-espace-de-noms-system-xml-linq

Chapitre 164: XmlDocument et l'espace de noms System.Xml

Examples

Interaction de base du document XML

```
public static void Main()
{
    var xml  = new XmlDocument();
    var root = xml.CreateElement("element");
        // Creates an attribute, so the element will now be "<element attribute='value' />"
        root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
            day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

Lecture du document XML

Un exemple de fichier XML

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
```

```
    </Sample>
```

Lecture de ce fichier XML:

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

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();

        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

XmlDocument vs XDocument (exemple et comparaison)

Il existe plusieurs manières d'interagir avec un fichier Xml.

1. Document XML
2. XDocument
3. XmlReader / XmlWriter

Avant LINQ to XML, nous avons utilisé XMLDocument pour des manipulations en XML, comme l'ajout d'attributs, d'éléments, etc. Maintenant, LINQ to XML utilise XDocument pour le même genre de chose. Les syntaxes sont beaucoup plus simples que XMLDocument et requièrent une quantité minimale de code.

Aussi XDocument est plus rapide que XmlDocument. XmlDoucument est une solution ancienne et sale pour interroger un document XML.

Je vais montrer quelques exemples de classes XmlDocument et de classes XDocument :

Charger un fichier XML

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

Créer XmlDocument

XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
 XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

Changer InnerText de noeud en XML

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

Enregistrer le fichier après modification

Assurez-vous de sécuriser le XML après tout changement.

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

Récupérer des valeurs à partir de XML

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

Récupérer la valeur de tous les éléments enfants où attribut = quelque chose.

XmlDocument

```
List<string> valueList = new List<string>();
    foreach (XmlNode n in nodelist)
    {
        if(n.Attributes["type"].InnerText == "City")
        {
            valueList.Add(n.Attributes["type"].InnerText);
        }
    }
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

Ajouter un noeud

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to safe the document*/
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new
XElement("SecondLevelNode"));

 /*After a change make sure to safe the document*/
_doc.Save(fileName);
```

Lire XmlDocument et l'espace de noms System.Xml en ligne:
https://riptutorial.com/fr/csharp/topic/1528/xmldocument-et-l-espace-de-noms-system-xml

Chapitre 165: Y compris les ressources de police

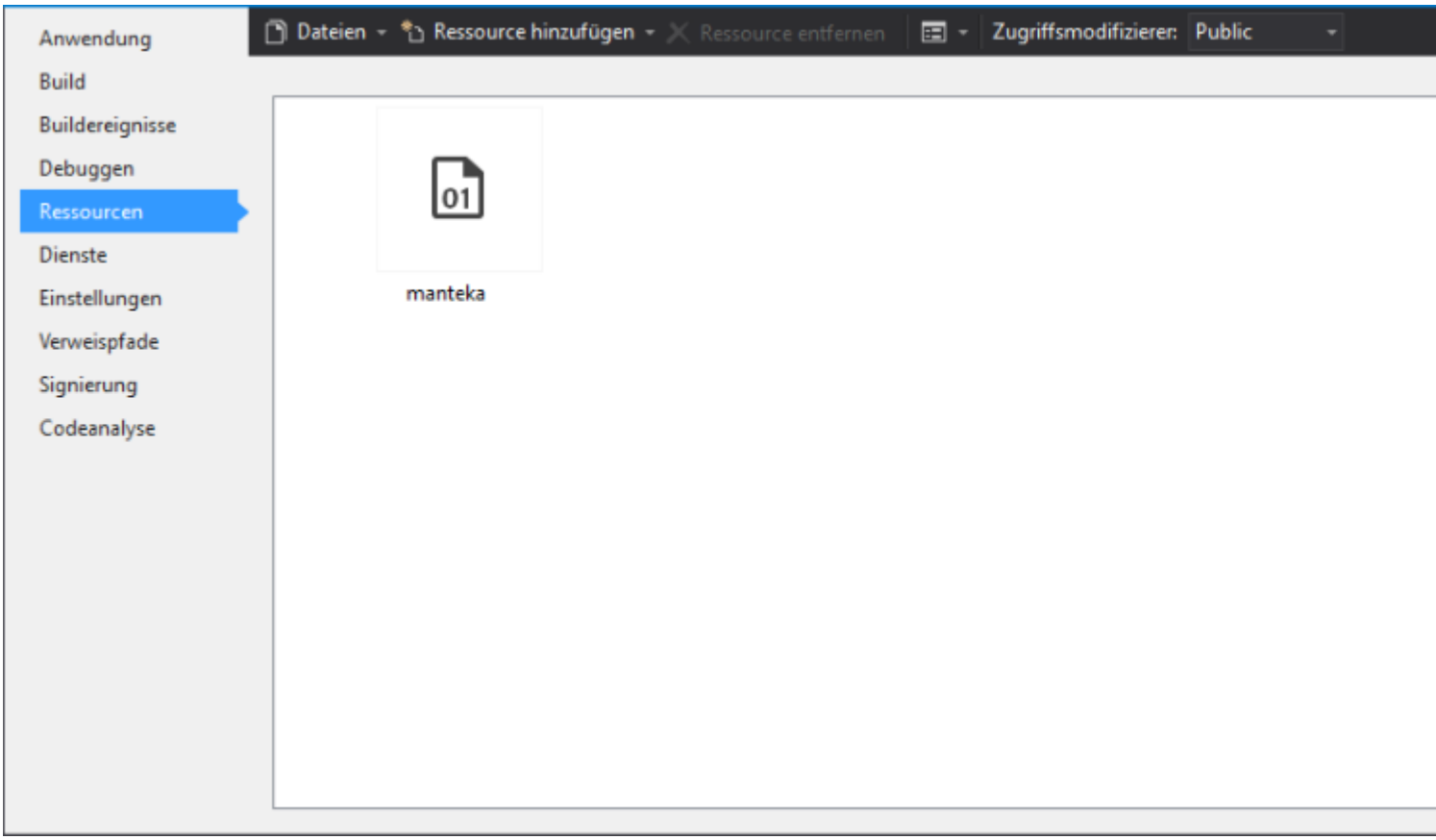
Paramètres

Paramètre	Détails
fontbytes	tableau d'octets du binaire .ttf

Examples

Instancier «Fontfamily» à partir des ressources

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Méthode d'intégration

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
```

```
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
    return pfc.Families[0];
}
```

Utilisation avec un bouton

```
    public static class Res
    {
        /// <summary>
        /// URL: https://www.behance.net/gallery/2846011/Manteka
        /// </summary>
        public static FontFamily Maneteke =
GetResourceFontFamily(Properties.Resources.manteka);

        public static FontFamily GetResourceFontFamily(byte[] fontbytes)
        {
            PrivateFontCollection pfc = new PrivateFontCollection();
            IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
            Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
            pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
            Marshal.FreeCoTaskMem(fontMemPointer);
            return pfc.Families[0];
        }
    }

    public class FlatButton : Button
    {
        public FlatButton() : base()
        {
            Font = new Font(Res.Maneteke, Font.Size);
        }

        protected override void OnFontChanged(EventArgs e)
        {
            base.OnFontChanged(e);
            this.Font = new Font(Res.Maneteke, this.Font.Size);
        }
    }
```

Lire Y compris les ressources de police en ligne: https://riptutorial.com/fr/csharp/topic/9789/y-compris-les-ressources-de-police

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage C #	4444, A. Raza, A_Arnold , aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Confiqure, crunchy, David G., David Pine, DavidG, DAXaholic , Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch , Frankenstine Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis , Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k , Kevin Montrose, Kritner , Krzyserious, leumas1960, M Monis

		Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput , Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki , Zze
2	Accéder au dossier partagé du réseau avec le nom d'utilisateur et le mot de passe	Mohsin khan
3	Accéder aux bases de données	ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864
4	Alias de types intégrés	Racil Hilan, Rahul Nikate , Stephen Leppik
5	Analyse de regex	C4u
6	Annotation des données	Maxime, Mikko Viitala, The_Outsider, Will Ray
7	Arbres d'expression	Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer , Matas Vaitkevicius, matteeyah, meJustAndrew, Nathan

		Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Рахул Маквана
8	Arguments nommés	Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen Leppik
9	Arguments nommés et facultatifs	RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov
10	AssemblyInfo.cs Exemples	Adi Lester, Ameya Deshpande, AndreyAkinshin, Boggin, Dodzi Dzakuma, dove, Joel Martinez, pinkfloydx33, Ralf Bönning, Theodoros Chatzigiannakis, Wasabi Fan
11	Async / wait, Backgroundworker, Exemples de tâches et de threads	Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi
12	Async-Await	Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean, Craig Brett, Donald Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft, Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev,

		Stefano d'Antonio, Stephen Leppik, SynerCoder, trashr0x, Tseng, user2321864, Vincent
13	BackgroundWorker	Bovaz, Draken, ephtee, Jacobr365, Will
14	Bibliothèque parallèle de tâches	Benjamin Hodgson, Brandon, Collin Stevens, i3arnon, Mokhtar Ashour , Murtuza Vohra
15	BigInteger	4444, Ed Marty, James Hughes, Rob, The_Outsider
16	BindingList	Bovaz, Stephen Leppik, yumaikas
17	C # Script	mehrandvd, Squidward, Stephen Leppik
18	Chronomètres	Adam, demonplus, dotctor, Gavin Greenwalt , Jeppe Stig Nielsen, Sondre
19	Classe partielle et méthodes	Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider
20	Classes statiques	MCronin, The_Outsider, Xiaoy312
21	CLSCompliantAttribute	mybirthname, Rob
22	Code non sécurisé dans .NET	Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz
23	Comment utiliser les structures C # pour créer un type d'union (similaire aux unions C)	DLeh, Milton Hernandez, Squidward, usr

24	Commentaires et régions	Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward
25	Commentaires sur la documentation XML	Alexander Mandt, James , jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn
26	Concaténation de cordes	Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi, Rob, RubberDuck, Testing123, The_Outsider
27	Constructeurs et finaliseurs	Adam Sills, Adi Lester, Adriano Repetti, Andrei Rînea, Andrew Diamond, Arjan Einbu, Avia, BackDoorNoBaby, BanksySan, Ben Fogel, Benjamin Hodgson, Benjol, Bogdan Gavril, Bovaz, Carlos Muñoz, Dan Hulme, Daryl, DLeh, Dmitry Bychenko, drusellers, Ehsan Sajjad, Fernando Matsumoto, guntbert, hatchet, Ian, Jeremy Kato, Jon Skeet, Julien Roncaglia, kamilk, Konamiman, ltiveron, Michael Richardson, Neel, Oly, Pavel Mayorov, Pavel Sapehin, Pavel Voronin, Peter Hommel, pinkfloydx33, Robert Columbia, RomCoo, Roy Dictus, Sam, Saravanan Sachi, Seph, Sklivvz, The_Cthulhu_Kid, Tim Medora, usr, Verena

		Haunschmid, void, Wouter, ZenLulz
28	Constructions de flux de données TPL (Task Parallel Library)	Droritos, Stephen Leppik
29	Contexte de synchronisation dans Async-Await	codeape, Mark Shevchenko, RamenChef
30	Contrats de code	MegaTron
31	Contrats de code et assertions	Roy Dictus
32	Conventions de nommage	Ben Aaronson, Callum Watkins, PMF, ZenLulz
33	Conversion de type	Community, connor, Ehsan Sajjad, Lijo
34	Cordes Verbatim	Alan McBee, Amitay Stern, Andrew Diamond, Aphelion, Arjan Einbu, avb, Bryan Crosby, Charlie H, David G., devuxer, DLeh, Ehsan Sajjad, Freelex, goric, Jared Hooper, Jeremy Kato, Jonas S, Kevin Montrose, Kilazur, Mateen Ulhaq, Ricardo Amores, Rion Williams, Sam Johnson, Sophie Jackson-Lee, Squirrel, th1rdey3
35	Courant	Danny Bogers, jlawcordova, Jon Schneider, Nuri Tasdemir, Pushpendra
36	Création d'une application console à l'aide d'un éditeur de texte brut et du compilateur C # (csc.exe)	delete me
37	Créer son propre MessageBox dans l'application Windows Form	Mansel Davies, Vaibhav_Welcomes_You
38	Cryptographie (System.Security.Cryptography)	glaubergft, MikeS159, Ogglas, Pete

39	Débordement	Akshay Anand, Nuri Tasdemir, tonirush
40	Déclaration de verrouillage	Aaron Hudon, Alexey Groshev, Andrei Rînea, Benjamin Hodgson, Botond Balázs, Christopher Currens, Cihan Yakar, David Ben Knoble, Denis Elkhov, Diligent Key Presser, George Duckett, George Polevoy, Jargon, Jasmin Solanki, Jivan, Mark Shevchenko, Matas Vaitkevicius, Mikko Viitala, Nuri Tasdemir, Oluwafemi, Pavel Mayorov, Richard, Rob, Scott Hannen, Squidward, Vahid Farahmandian
41	Délégués Func	Theodoros Chatzigiannakis, Valentin
42	Des minuteries	Adam, Akshay Anand, Benjamin Kozuch, ephtee, RamenChef, Thennarasan
43	Diagnostic	Jasmin Solanki, Luke Ryan, TylerH
44	Directives du préprocesseur	Andrei, Gilad Naaman, Matas Vaitkevicius, qJake, RamenChef, theB, volvis
45	En boucle	Alisson, Andrei Rînea, B Hawkins, Benjamin Hodgson, Botond Balázs , connor, Dialecticus, DJCubed, Freelex, Jon Schneider, Oluwafemi, Racil Hilan, Squidward, Testing123, Tolga

		Evcimen
46	Enum	Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked, Kyle Trauberman, Martin Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, Tanner Swett, TarkaDaal , Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai
47	Est égal à et GetHashCode	Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew, Rob, scher, Timitry, viggity
48	Événements	Aaron Hudon, Adi Lester , Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng , NotEnoughData, OliPro007, paulius_l, PSGuy, Reza Aghaei, Roy Dictus, Squidward, Steven, vbnet3d
49	Exécution de requêtes HTTP	Gordon Bell, Jon Schneider, Mark

		Shevchenko
50	Expressions conditionnelles	Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz
51	Expressions lambda	Andrei Rînea, Benjamin Hodgson, Benjol, David L, David Pine, Federico Allocati, Feelbad Soussi Wolfgun DZ, Fernando Matsumoto, H. Pauwelyn , haim770, Matas Vaitkevicius, Matt Sherman, Michael Mairegger, Michael Richardson, NotEnoughData, Oly, RubberDuck, S.L. Barth, Sunny R Gupta, Tagc, Thriggle
52	Extensions Réactives (Rx)	stefankmitph
53	Fichier et flux I / O	BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgun DZ, intox, Mikko Viitala, Sender, Squidward, Tolga Evcimen, Wasabi Fan
54	FileSystemWatcher	Sondre
55	Filetage	Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepdelaw, Mamta D, Matthijs Wessels, Mellow , RamenChef, Zoba
56	Filtres d'action	Lokesh_Ram
57	Fonction avec plusieurs valeurs de retour	Adam, Alexey Mitev,

		Durgpal Singh, Tolga Evcimen
58	Fonctionnalités C # 3.0	0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob
59	Fonctionnalités C # 5.0	Abob, alex.b, H. Pauwelyn
60	Fonctionnalités C # 7.0	Adil Mammadov, afuna, Amitay Stern, Amr Badawy, Andreas Pähler , Andrew Diamond, Avi Turner, Benjamin Hodgson, Blorgbeard, bluray, Botond Balázs, Bovaz, Cerbrus, Clueless, Conrad.Dean, Dale Chen, David Pine, Degusto, Didgeridoo, Diligent Key Presser, ECC-Dan, Emre Bolat, fallaciousreasoning, ferday, Florian Greinacher, ganchito55, Ginkgo, H. Pauwelyn, Henrik H, Icy Defiance, Igor Ševo, iliketocode, Jatin Sanghvi, Jean-Bernard Pellerin, Jesse Williams, Jon Schoning, Kimmax, Kobi, Kris Vandermotten, Kritner, leppie, Llwyd, Maakep, maf-soft, Marc Gravell, MarcinJuraszek, Mariano Desanze, Matt Rowland, Matt Thomas, MemphiZ, mnoronha, MotKohn, Name, Nate Barbettini, Nico, Niek, nietras, NikolayKondratyev, Nuri Tasdemir, PashaPash, Pavel Mayorov, PeteGO, petrzjunior, Philippe, Pratik, Priyank Gadhiya, Pyritie, qJake, Raidri,

		Rakitić, RamenChef, Ray Vega, RBT, René Vogt, Rob, samuelesque , Squidward, Stavm, Stefano, Stefano d'Antonio, Stilgar, Tim Pohlmann, Uriil, user1304444, user2321864, user3185569, uTeisT, Uwe Keim, Vlad, Vlad, Wai Ha Lee, Wasabi Fan , WerWet, wezten, Wojciech Czerniak, Zze
61	Fonctions C # 4.0	Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima, Sibeesh Venu, Squidward, Theodoros Chatzigiannakis
62	Fonctions C # 6.0	A_Arnold, Aaron Anodide, Aaron Hudon, Adil Mammadov, Adriano Repetti, AER, AGB, Akshay Anand, Alan McBee, Alex Logan, Amitay Stern, anaximander, andre_ss6 , Andrea, AndroidMechanic, Ares, Arthur Rizzo, Ashwin Ramaswami, avishayp, Balagurunathan Marimuthu, Bardia, Ben Aaronson, Blubberguy22 , Bobson, bpoiss, Bradley Uffner, Bret Copeland, C4u, Callum Watkins, Chad Levy, Charlie H, ChrFin, Community, Conrad.Dean, Cyprien Autexier, Dan, Daniel Minnaar, Daniel Stradowski, DarkV1,

dasblinkenlight, David, David G., David Pine, Deepak gupta, DLeh, dotctor, Durgpal Singh, Ehsan Sajjad, el2iot2, Emre Bolat, enrico.bacis, Erik Schierboom, fabriciorissetto, faso, Franck Dernoncourt, FrankerZ, Gabor Kecskemeti, Gary, Gates Wong, Geoff, GingerHead, Gordon Bell, Guillaume Pascal, H. Pauwelyn, hankide, Henrik H, iliketocode, Iordanis , Irfan, Ivan Yurchenko, J. Steen, Jacob Linney, Jamie Rees, Jason Sturges, Jeppe Stig Nielsen, Jim, JNYRanger, Joe, Joel Etherton, John Slegers, Johnbot, Jojodmo, Jonas S, Juan, Kapep, ken2k, Kit, Konamiman, Krikor Ailanjian, Lafexlos, LaoR , Lasse Vågsæther Karlsen, M.kazem Akhgary, Mafii, Magisch, Makyen, MANISH KUMAR CHOUDHARY, Marc, MarcinJuraszek, Mark Shevchenko, Matas Vaitkevicius, Mateen Ulhaq, Matt, Matt, Matt, Matt Thomas, Maximillian Laumeister, mbrdev, Mellow, Michael Mairegger, Michael Richardson, Michał Perłakowski, mike z, Minhas Kamal, Mitch Talmadge, Mohammad Mirmostafa, Mr.Mindor, mshsayem, MuiBienCarlota, Nate

		Barbettini, Nicholas Sizer , nik, nollidge, Nuri Tasdemir, Oliver Mellet, Orlando William, Osama AbuSitta, Panda, Parth Patel, Patrick, Pavel Voronin, PSN, qJake, QoP, Racil Hilan, Radouane ROUFID, Rahul Nikate, Raidri, Rajeev, Rakitić, ravindra, rdans, Reeven, Richa Garg, Richard, Rion Williams, Rob, Robban, Robert Columbia, Ryan Hilbert, ryanyuyu, Sam, Sam Axe, Samuel, Sender, Shekhar, Shoe, Slayther, solidcell, Squidward, Squirrel, stackptr, stark, Stilgar, Sunny R Gupta, Suren Srapyan, Sworgkh, syb0rg, takrl, Tamir Vered, Theodoros Chatzigiannakis, Timothy Shields, Tom Droste, Travis J, Trent, Trikaldarshi, Troyen, Tushar patel, tzachs, Uri Agassi, Uriil, uTeisT, vcsjones, Ven, viggity, Vishal Madhvani, Vlad, Wai Ha Lee, Xiaoy312, Yury Kerbitskov, Zano, Ze Rubeus, Zimm1
63	Fonctions de hachage	Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz
64	Fonderie	Benjamin Hodgson, MSE , RamenChef, StriplingWarrior

65	Garbage Collector dans .Net	Andrei Rînea, da_sann, Eamon Charles, J3soon, Luke Ryan, Squidward, Suren Srapyan
66	Générateur de requêtes Lambda générique	4444, PedroSouki
67	Génération de code T4	lloyd, Pavel Mayorov
68	Génération de nombres aléatoires en C #	A. Can Aydemir, Adi Lester, Alexander Mandt , DLeh, J3soon, Rob
69	Génériques	AGB, andre_ss6, Ben Aaronson, Benjamin Hodgson, Benjol, Bobson, Carsten, darth_phoenixx, dymanoid, Eamon Charles, Ehsan Sajjad, Gajendra, GregC, H. Pauwelyn, ja72, Jim, Kroltan, Matas Vaitkevicius, mehmetgil, meJustAndrew, Mord Zuber, Mujassir Nasir, Oly, Pavel Voronin, Richa Garg, Sam, Sebi, Sjoerd222888, Theodoros Chatzigiannakis, user3185569, VictorB, void, Wallace Zhang
70	Gestion de FormatException lors de la conversion d'une chaîne en d'autres types	Rakitić, un-lucky
71	Gestion des exceptions	0x49D1, Abdul Rehman Sayed, Adam Lear, Adil Mammadov, Andrew Diamond, Aseem Gautam, Athafoud, Botond Balázs, Collin Stevens, Danny Chen, Dmitry Bychenko, dove, Eldar Dordzhiev, fabriciorissetto, faso, flq, George Duckett, Gilad

		Naaman, Gudradain, Jack, James Hughes, Jamie Rees, John Meyer , Jonesopolis, MadddinTribleD, Marimba, Matas Vaitkevicius, Matt, matteeyah, Mendhak, Michael Bisbjerg, Nate Barbettini, Nathaniel Ford, nik0lias, niksofteng , Oly, Pavel Pája Halbich , Pavel Voronin, PMF, Racil Hilan, raidensan, Rasa , Robert Columbia, RomCoo, Sam Hanley, Scott Koland, Squidward , Steve Dunn, Thulani Chivandikwa, vesi
72	Gestionnaire d'authentification C #	Abbas Galiyakotwala
73	Guid	Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void
74	Héritage	Almir Vuk, andre_ss6, Andrew Diamond, Barathon, Ben Aaronson , Ben Fogel, Benjol, David L, deloreyk, Ehsan Sajjad, harriyott, ja72, Jon Ericson, Karthik, Konamiman, MarcE, Matas Vaitkevicius, Pete Uh, Rion Williams, Robert Columbia, Steven , Suren Srapyan, VirusParadox, Yehuda Shapira
75	IClonable	ja72, Rob
76	IComparable	alex

77	Identité ASP.NET	HappyCoding, Skullomania
78	IEnumerable	4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider
79	ILGenerator	Aleks Andreev, thehennyy
80	Immutabilité	Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer
81	Importer les contacts Google	4444, Supraj v
82	Indexeur	A_Arnold, Ehsan Sajjad, jHilscher
83	Initialisation des propriétés	Blorgbeard, hatchet, jaycer, Michael Sorens, Parth Patel, Stephen Leppik
84	Initialiseurs d'objets	Andrei, Kroltan, LeopardSkinPillBoxHat, Marco, Nick DeVore, Stephen Leppik
85	Initialiseurs de collection	Aphelion, ASh, Bart Jolling, Chronocide, CodeCaster, CyberFox, DLeh, Jacob Linney, Jeromy Irvine, Jonas S, Matas Vaitkevicius, Rob, robert demartino, rudygt, Squidward, Tamir Vered, TarkaDaal, Thulani Chivandikwa, WMios
86	Injection de dépendance	Buh Buh, iaminvinicble, Kyle Trauberman, Wiktor Dębski
87	Interface IDisposable	Aaron Hudon, Adam, BatteryBackupUnit, binki, Bogdan Gavril, Bryan Crosby, ChrisWue,

		Dmitry Bychenko, Ehsan Sajjad, H. Pauwelyn, Jarrod Dixon, Josh Peterson, Matas Vaitkevicius, Maxime, Nicholas Sizer, OliPro007, Pavel Mayorov, pinkfloydx33, pyrocumulus, RamenChef, Rob, Thennarasan, Will Ray
88	Interface INotifyPropertyChanged	mbrdev, Stephen Leppik, Vlad
89	Interface IQueryable	lucavgobbi, Michiel van Oosterhout, RamenChef, Rob
90	Interfaces	Avia, Botond Balázs, CyberFox, harriyott, hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511
91	Interopérabilité	Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward
92	Interpolation de chaîne	Arjan Einbu, ATechieThought, avs099 , bluray, Brendan L, Dave Zych, DLeh, Ehsan Sajjad, fabriciorissetto, Guilherme de Jesus Santos, H. Pauwelyn, Jon Skeet, Nate Barbettini, RamenChef, Rion Williams, Squidward, Stephen Leppik, Tushar patel,

		Wasabi Fan
93	La mise en réseau	Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano
94	Lecture et écriture de fichiers .zip	4444, DLeh, Naveen Gogineni, Nisarg Shah
95	Les attributs	Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili
96	Les délégués	Aaron Hudon, Adam, Ben Aaronson, Benjamin Hodgson, Bradley Uffner , CalmBit, Cihan Yakar, CodeWarrior, EyasSH, Huseyin Durmus, Jasmin Solanki, Jeppe Stig Nielsen, Jon G, Jonas S, Matt, NikolayKondratyev, niksofteng, Rajput, Richa Garg, Sam Farajpour Ghamari, Shog9, Stu, Thulani Chivandikwa, trashr0x
97	Les itérateurs	Botond Balázs, Lijo, Nate Barbettini, Tagc
98	Les méthodes	Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky
99	Les opérateurs	Adam Houldsworth, Adi Lester, Adil Mammadov, Akshay Anand, Alan

		McBee, Avi Turner, Ben Fogel, Blorgbeard, Blubberguy22, Chris Jester-Young, David Basarab, DLeh, Dmitry Bychenko, dotctor, Ehsan Sajjad, fabriciorissetto, Fernando Matsumoto, H. Pauwelyn, Henrik H, Jake Farley, Jasmin Solanki, Jephron, Jeppe Stig Nielsen, Jesse Williams, Joe, JohnLBevan, Jon Schneider, Jonas S, Kevin Montrose, Kimmax , lokusking, Matas Vaitkevicius, meJustAndrew, Mikko Viitala, mmushtaq, Mohamed Belal, Nate Barbettini, Nico, Oly, pascalhein, Pavel Voronin, petelids, Philip C, Racil Hilan, RhysO, Robert Columbia, Rodolfo Fadino Junior, Sachin Joseph, Sam, slawekwin, slinzerthegod , Squidward, Testing123, TyCobb, Wasabi Fan, Xiaoy312, Zaheer Ul Hassan
100	Linq to Objects	brijber, Christian Gollhardt, FortyTwo, Kevin Green, Raphael Pantaleão, Simon Halsey, Tanveer Badar
101	LINQ to XML	Denis Elkhov, Stephen Leppik, Uali
102	Lire et comprendre les empilements	S.L. Barth
103	Littéraux	jaycer, NotEnoughData, Racil Hilan

104	Manipulation de cordes	Blachshma, Jon Schneider, sferencik, The_Outsider
105	Méthodes d'extension	Aaron Hudon, AbdulRahman Ansari, Adi Lester, Adil Mammadov, AGB, AldoRomo88, anaximander, Aphelion, Ashwin Ramaswami, ATechieThought, Ben Aaronson, Benjol, binki, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, Brett Veenstra, brijber, Callum Watkins, Chad McGrath, Charlie H, Chris Akridge, Chronocide, CorrectorBot, cubrr, Dan-Cook, Daniel Stradowski, David G., David Pine, Deepak gupta, diiN__________, DLeh, Dmitry Bychenko, DoNot, DWright, Đan, Ehsan Sajjad, ekolis, el2iot2, Elton, enrico.bacis, Erik Schierboom, ethorn10, extremeboredom, Ezra, fahadash, Federico Allocati, Fernando Matsumoto, FrankerZ, gdziadkiewicz, Gilad Naaman, GregC, Gudradain, H. Pauwelyn, HimBromBeere, Hsu Wei Cheng, Icy Defiance, Jamie Rees, Jeppe Stig Nielsen, John Peters, John Slegers, Jon Erickson, Jonas S, Jonesopolis, Kev, Kevin Avignon, Kevin DiTraglia , Kobi, Konamiman,

		krillgar, Kurtis Beavers, Kyle Trauberman, Lafexlos, LMK, lothlarias, Lukáš Lánský, Magisch, Marc, MarcE, Marek Musielak, Martin Zikmund, Matas Vaitkevicius, Matt, Matt Dillard, Maximilian Ast, mbrdev, MDTech.us_MAN, meJustAndrew, Michael Benford, Michael Freidgeim, Michael Richardson, Michał Perłakowski, Nate Barbettini, Nick Larsen, Nico, Nisarg Shah, Nuri Tasdemir, Parth Patel, pinkfloydx33, PMF, Prashanth Benny, QoP, Raidri, Reddy, Reeven, Ricardo Amores, Richard , Rion Williams, Rob, Robert Columbia, Ryan Hilbert, ryanyuyu, S. Tarık Çetin, Sam Axe, Shoe , Sibeesh Venu, solidcell , Sondre, Squidward, Steven, styfle, SysVoid, Tanner Swett, Timothy Rascher, TKharaishvili, T-moty, Tobbe, Tushar patel, unarist, user3185569, user40521 , Ven, Victor Tomaili, viggity
106	Méthodes DateTime	AbdulRahman Ansari, C4u, Christian Gollhardt, Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz , Oluwafemi, Pavel Yermalovich, Sondre,

		theinarasu, Thulani Chivandikwa
107	Microsoft.Exchange.WebServices	Bassie
108	Mise en cache	Aliaksei Futryn, th1rdey3
109	Mise en œuvre d'un modèle de conception de décorateur	Jan Bońkowski
110	Mise en œuvre d'un modèle de conception de poids mouche	Jan Bońkowski
111	Mise en œuvre singleton	Aaron Hudon, Adam, Adi Lester, Andrei Rînea, cbale, Disk Crasher, Ehsan Sajjad, Krzysztof Branicki, lothlarias, Mark Shevchenko, Pavel Mayorov, Sklivvz, snickro, Squidward, Squirrel, Stephen Leppik , Victor Tomaili, Xandrmoro
112	Modèles de conception créative	DWright, Jan Bońkowski, Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel
113	Modèles de conception structurelle	Timon Post
114	Modificateurs d'accès	Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz
115	Mots clés	4444, A_Arnold, Aaron Hudon, Ade Stringer, Adi Lester, Aditya Korti, Adriano Repetti, AJ., Akshay Anand, Alex Filatov, Alexander Pacha , Amir Pourmand, Andrei Rînea, Andrew Diamond, Angela, Anna, Avia, Bart , Ben, Ben Fogel,

		Benjamin Hodgson, Bjørn-Roger Kringsjå, Botz3000, Brandon, brijber, BrunoLM, BunkerMentality, BurnsBA, bwegs, Callum Watkins, Chris, Chris Akridge, Chris H., Chris Skardon, ChrisPatrick, Chuu, Cihan Yakar, cl3m , Craig Brett, Daniel, Daniel J.G., Danny Chen , Darren Davies, Daryl, dasblinkenlight, David, David G., David L, David Pine, DAXaholic, deadManN, DeanoMachino, digitlworld, Dmitry Bychenko, dotctor, DPenner1, Drew Kennedy, DrewJordan, Ehsan Sajjad, EJoshuaS , Elad Lachmi, Eric Lippert, EvenPrime, F_V, Felix, fernacolo, Fernando Matsumoto, forsvarir, Francis Lord, Gavin Greenwalt, gdoron , George Duckett, Gilad Naaman, goric, greatwolf , H. Pauwelyn, Happypig375, Icemanind , Jack, Jacob Linney, Jake, James Hughes, Jcoffman, Jeppe Stig Nielsen, jHilscher, João Lourenço, John Slegers, JohnD, Jon Schneider, Jon Skeet, JoshuaBehrens, Kilazur, Kimmax, Kirk Woll, Kit, Kjartan, kjhf, Konamiman , Kyle Trauberman, kyurthich, levininja, lokusking, Mafii, Mamta D, Mango Wong, MarcE,

		MarcinJuraszek, Marco Scabbiolo, Martin, Martin Klinke, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matěj Pokorný, Mat's Mug, Matthew Whited, Max, Maximilian Ast, Medeni Baykal, Michael Mairegger, Michael Richardson, Michel Keijzers, Mihail Shishkov , mike z, Mr.Mindor, Myster, Nicholas Sizer, Nicholaus Lawson, Nick Cox, Nico, nik, niksofteng, NotEnoughData, numaroth, Nuri Tasdemir , pascalhein, Pavel Mayorov, Pavel Pája Halbich, Pavel Yermalovich, Pavieł Kraskoŭski, Paweł Mach, petelids, Peter Gordon, Peter L., PMF, Rakitić, RamenChef, ranieuwe, Razan, RBT, Renan Gemignani, Ringil, Rion Williams, Rob, Robert Columbia, ro binmckenzie, RobSiklos, Romain Vincent, RomCoo, ryanyuyu, Sain Pradeep, Sam, Sándor Mátyás Márton, Sanjay Radadiya, Scott, sebingel, Skipper, Sobieck, sohnryang, somebody, Sondre, Squidward, Stephen Leppik, Sujay Sarma, Suyash Kumar Singh, svick, TarkaDaal, th1rdey3, Thaoden, Theodoros

		Chatzigiannakis, Thorsten Dittmar, Tim Ebenezer, titol, tonirush, topolm, Tot Zam, user3185569, Valentin, vcsjones, void, Wasabi Fan, Wavum, Woodchipper, Xandrmoro, Zaheer Ul Hassan, Zalomon, Zohar Peled
116	nom de l'opérateur	Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan
117	NullReferenceException	4444, Agramer, Ashutosh, krimog, Kyle Trauberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato
118	O (n) Algorithme de rotation circulaire d'un tableau	AFT
119	ObservableCollection	demonplus, GeralexGR, Jonathan Anctil, MuiBienCarlota
120	Opérateur d'égalité	Vadim Martynov
121	Opérateur de coalescence nulle	aashishkoirala, Ankit Rana, Aristos, Bradley Uffner, David Arno, David G., David Pine, demonplus, Denis Elkhov, Diligent Key Presser, Eamon Charles, Ehsan Sajjad, eouw0o83hf, Fernando Matsumoto, H. Pauwelyn , Jodrell, Jon Schneider,

		Jonesopolis, Martin Zikmund, Mike C, Nate Barbettini, Nic Foster, petelids, Prateek, Rahul Nikate, Rion Williams, Rob, smead, tonirush, Wasabi Fan, Will Ray
122	Opérateurs Null-Conditionnels	Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan
123	Opérations sur les chaînes communes	Austin T French, Blachshma, bluish, CharithJ, Chief Wiggum, cyberj0g, Daryl, deloreyk , jaycer, Jaydip Jadhav, Jon G, Jon Schneider, juergen d, Konamiman, Maniero, Paul Weiland, Racil Hilan, RoelF, Stefan Steiger, Steven, The_Outsider, tiedied61, un-lucky, WizardOfMenlo
124	Parallèle LINQ (PLINQ)	Adi Lester
125	Plate-forme de compilation .NET (Roslyn)	4444, Lukáš Lánský
126	Pointeurs	Jeppe Stig Nielsen, Theodoros Chatzigiannakis
127	Pointeurs & Code dangereux	Aaron Hudon, Botond Balázs, undefined
128	Polymorphisme	Ade Stringer, ganchito55 , H. Pauwelyn, Karthik, Maximilian Ast, void
129	Pour commencer: Json avec C #	Neo Vijay, Rob, VitorCioletti
130	Prise asynchrone	Timon Post

131	Programmation fonctionnelle	Andrei Epure, Boggin, Botond Balázs, richard
132	Programmation orientée objet en C #	Yashar Aliabasi
133	Propriétés	Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled
134	Récursivité	Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes, Yves Schelpe
135	Réflexion	Alexander Mandt, Aman Sharma, artemisart, Aseem Gautam, Axarydax, Benjamin Hodgson, Botond Balázs , Carson McManus, Cigano Morrison Mendez , Cihan Yakar, da_sann, DVJex, Ehsan Sajjad, H. Pauwelyn, Haim Bendanan, HimBromBeere, James Ellis-Jones, James Hughes, Jamie Rees, Jan Peldřimovský, Johny Skovdal, JSF, Kobi, Konamiman, Kristijan, Lovy, Matas Vaitkevicius , Mourndark, Nuri Tasdemir, pinkfloydx33, Rekshino, René Vogt, Sachin Chavan, Shuffler, Sjoerd222888, Sklivvz, Tamir Vered, Thriggle, Travis J, uygar.raf, Vadim Ovchinnikov, wablab, Wai Ha Lee

136	Rendement	Aaron Hudon, Andrew Diamond, Ben Aaronson , ChrisPatrick, Damon Smithies, David G., David Pine, Dmitry Bychenko, dotctor, Ehsan Sajjad, erfanrazi, Gajendra, George Duckett, H. Pauwelyn, HimBromBeere, Jeremy Kato, João Lourenço, Joe Amenta, Julien Roncaglia, just.ru, Karthik AMR, Mark Shevchenko, Michael Richardson, MuiBienCarlota, Myster, Nate Barbettini, Noctis, Nuri Tasdemir, Olivier De Meulder, OP313, ravindra, Ricardo Amores, Rion Williams, rocky, Sompom, Tot Zam, un-lucky, Vlad, void, Wasabi Fan, Xiaoy312, ZenLulz
137	Rendre un thread variable sûr	Wyck
138	Requêtes LINQ	Adam Clifford, Ade Stringer, Adi Lester, Adil Mammadov, Akshay Anand, Aleksey L., Alexey Koptyaev, AMW, anaximander, Andrew Piliser, Ankit Vijay, Aphelion, bbonch, Benjamin Hodgson, bmadtiger, BOBS, BrunoLM, Budi, bumbeishvili, callisto, cbale, Chad McGrath, Chris, Chris H., coyote, Daniel Argüelles, Daniel Corzo, darcyq, David, David G., David Pine, DavidG, die maus,

Diligent Key Presser, Dmitry Bychenko, Dmitry Egorov, dotctor, Ehsan Sajjad, Erick, Erik Schierboom, EvenPrime, fabriciorissetto, faso, Finickyflame, Florin M, forsvarir, fubo, gbellmann, Gene, Gert Arnold, Gilad Green, H. Pauwelyn, Hari Prasad, hellyale, HimBromBeere, hWright, iliketocode, Ioannis Karadimas, Jagadisha B S, James Ellis-Jones, jao, jiaweizhang, Jodrell, Jon Bates, Jon G, Jon Schneider, Jonas S, karaken12, KevinM, Koopakiller, leppie, LINQ , Lohitha Palagiri, ltiveron, Mafii, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matt, Maxime, mburleigh, Meloviz, Mikko Viitala, Mohammad Dehghan, mok, Nate Barbettini, Neel, Neha Jain, Néstor Sánchez A., Nico, Noctis , Pavel Mayorov, Pavel Yermalovich, Paweł Hemperek, Pedro, Phuc Nguyen, pinkfloydx33, przno, qJake, Racil Hilan , rdans, Rémi, Rion Williams, rjdevereux, RobPethi, Ryan Abbott, S. Rangeley, S.Akbari, S.L. Barth, Salvador Rubio Martinez, Sanjay Radadiya, Satish Yadav, sebingel, Sergio Domínguez, SilentCoder, Sivanantham Padikkasu,

		slawekwin, Sondre, Squidward, Stephen Leppik, Steve Trout, Tamir Vered, techspider, teo van kot, th1rdey3, Theodoros Chatzigiannakis, Tim Iles , Tim S. Van Haren, Tobbe, Tom, Travis J, tungns304, Tushar patel, user1304444, user3185569, Valentin, varocarbas, VictorB, Vitaliy Fedorchenko, vivek nuna, void, wali, wertzui, WMios, Xiaoy312, Yaakov Ellis, Zev Spitz
139	Résolution de surcharge	Dunno, Petr Hudeček, Stephen Leppik, TorbenJ
140	Runtime Compile	Artificial Stupidity, Stephen Leppik, Tommy
141	Séquences d'échappement de chaîne	Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush
142	Sérialisation Binaire	David, Maxim, RamenChef, Stephen Leppik
143	String.Format	Aaron Hudon, Akshay Anand, Alexander Mandt , Andrius, Aseem Gautam, Benjol, BrunoLM, Dmitry Egorov , Don Vince, Dweeberly, ebattulga, ejhn5, gdoron, H. Pauwelyn, Hossein Narimani Rad, Jasmin Solanki, Marek Musielak,

		Mark Shevchenko, Matas Vaitkevicius, Mendhak, MGB, nikchi, Philip C, Rahul Nikate, Raidri, RamenChef, Richard, Richard, Rion Williams, ryanyuyu, teo van kot, Vincent, void, Wyck
144	StringBuilder	ATechieThought, brijber, Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider
145	Structs	abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad Lachmi, ganchito55, Jon Schneider, NikolayKondratyev
146	System.DirectoryServices.Protocols.LdapConnection	Andrew Stollak
147	System.Management.Automation	Mikko Viitala
148	Tableaux	A_Arnold, Aaron Hudon, Alexey Groshev, Anas Tasadduq, Andrii Abramov, Baddie, Benjamin Hodgson, bluray, coyote, D.J., das_keyboard, Fernando Matsumoto, granmirupa, Jaydip Jadhav, Jeppe Stig Nielsen, Jon Schneider, Ogoun, RamenChef, Robert Columbia, Shyju, The_Outsider, Thomas Weller, tonirush, Tormod Haugene, Wasabi Fan, Wen Qin, Xiobiq, Yotam Salmon
149	Tuples	Bovaz, Chawin, EFrank, H. Pauwelyn, Mark Benovsky, Muhammad

		Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011
150	Type de valeur vs type de référence	Abdul Rehman Sayed, Adam, Amir Pourmand, Blubberguy22, Chronocide, Craig Brett, docesam, GWigWam, matiaslauriti, meJustAndrew, Michael Mairegger, Michele Ceo, Moe Farag, Nate Barbettini, RamenChef, Rob, scher, Snympi, Tagc, Theodoros Chatzigiannakis
151	Type dynamique	Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob
152	Types anonymes	Fernando Matsumoto, goric, Stephen Leppik
153	Types intégrés	Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee
154	Types nullables	Benjamin Hodgson, Braydie, DmitryG, Gordon Bell, Jasmin Solanki, Jon Schneider, Konstantin Vdovkin, Maximilian Ast, Mikko Viitala, Nicholas Sizer, Patrick Hofman, Pavel Mayorov, pinkfloydx33, Vitaliy Fedorchenko
155	Un aperçu des collections c #	Aaron Hudon, Andrew Diamond, Denuath, Jeremy Kato, Jon Schneider, Jorge, Juha

		Palomäki, Leon Husmann, Michael Mairegger, Michael Richardson, Nikita, rene, Rob, Sebi, TarkaDaal, wertzui, Will Ray
156	Utiliser json.net	Aleks Andreev, Snipzwolf
157	Utiliser la déclaration	Adam Houldsworth, Ahmar, Akshay Anand, Alex Wiese, andre_ss6, Aphelion, Benjol, Boris Callens, Bradley Grainger, Bradley Uffner, bubbleking, Chris Marisic , ChrisWue, Cristian T, cubrr, Dan Ling, Danny Chen, dav_i, David Stockinger, dazerdude, Denis Elkhov, Dmitry Bychenko, Erik Schierboom, Florian Greinacher, gdoron, H. Pauwelyn, Herbstein, Jon Schneider, Jon Skeet, Jonesopolis, JT., Ken Keenan, Kev, Kobi, Kyle Trauberman, Lasse Vågsæther Karlsen, LegionMammal978, Lorentz Vedeler, Martin, Martin Zikmund, Maxime , Nuri Tasdemir, Peter K, Philip C, pid, René Vogt, Rion Williams, Ryan Abbott, Scott Koland, Sean, Sparrow, styfle, Sunny R Gupta, Sworgkh, Thaoden, The_Cthulhu_Kid, Tom Droste, Tot Zam, Zaheer Ul Hassan
158	Utiliser la directive	Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit,

		Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam
159	Utiliser SQLite en C #	Carmine, NikolayKondratyev, th1rdey3, Tim Yusupov
160	Vérifié et décoché	Botond Balázs, Rahul Nikate, Sam Johnson, ZenLulz
161	Windows Communication Foundation	NtFreX
162	XDocument et l'espace de noms System.Xml.Linq	Crowcoder, Jon Schneider
163	XmlDocument et l'espace de noms System.Xml	Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr
164	Y compris les ressources de police	Bales, Facebamm