eBook Gratuit

APPRENEZ .NET Framework

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

#.net

Table des matières

À propos

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

It is an unofficial and free .NET Framework 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 .NET Framework.

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 .NET Framework

Remarques

Le .NET Framework est un ensemble de bibliothèques et un environnement d'exécution, conçus à l'origine par Microsoft. Tous les programmes .NET compilent en un bytecode appelé Microsoft Intermediate Language (MSIL). Le MSIL est exécuté par le Common Language Runtime (CLR).

Vous trouverez ci-dessous plusieurs exemples de "Hello World" dans différentes langues prenant en charge le .NET Framework. "Hello World" est un programme qui affiche "Hello World" sur le périphérique d'affichage. Il est utilisé pour illustrer la syntaxe de base pour construire un programme de travail. Il peut également être utilisé pour vérifier que le compilateur, l'environnement de développement et l'environnement d'exécution d'un langage fonctionnent correctement.

Liste des langues supportées par .NET

Versions

.NET

Version	Date de sortie
1.0	2002-02-13
1.1	2003-04-24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013-10-17
4.5.2	2014-05-05
4.6	2015-07-20
4.6.1	2015-11-17

Version	Date de sortie
4.6.2	2016-08-02
4.7	2017-04-05

Cadre compact

Version	Date de sortie
1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3,9	2013-06-01

Micro Framework

Version	Date de sortie
4.2	2011-10-04
4.3	2012-12-04
4.4	2015-10-20

Examples

Bonjour tout le monde en C

```
using System;

class Program
{
    // The Main() function is the first function to be executed in a program
    static void Main()
    {
        // Write the string "Hello World to the standard out
        Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` a plusieurs surcharges. Dans ce cas, la chaîne "Hello World" est le paramètre et le résultat sera "Hello World" dans le flux de sortie standard lors de l'exécution. D'autres

surcharges peuvent appeler le `.ToString` de l'argument avant d'écrire dans le flux. Consultez la documentation de .NET Framework pour plus d'informations.

Démo en direct en action sur le violon .NET

Introduction à C #

Bonjour tout le monde dans Visual Basic .NET

```
Imports System

Module Program
    Public Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

Démo en direct en action sur le violon .NET

Introduction à Visual Basic .NET

Bonjour tout le monde en fa

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

Démo en direct en action sur le violon .NET

Introduction à F #

Bonjour tout le monde en C ++ / CLI

```
using namespace System;

int main(array<String^>^ args)
{
    Console::WriteLine("Hello World");
}
```

Bonjour tout le monde dans PowerShell

```
Write-Host "Hello World"
```

Introduction à PowerShell

Bonjour tout le monde à Nemerle

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

Bonjour tout le monde en oxygène

```
namespace HelloWorld;

interface

type
  App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
  Console.WriteLine('Hello World');
end;

end.
```

Bonjour tout le monde à Boo

```
print "Hello World"
```

Bonjour tout le monde en Python (IronPython)

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

Bonjour tout le monde en IL

```
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  }

  .method public hidebysig specialname rtspecialname
         instance void  .ctor() cil managed
  {
```

```
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Lire Démarrer avec .NET Framework en ligne: https://riptutorial.com/fr/dot-net/topic/14/demarrer-avec--net-framework

Chapitre 2: .NET Core

Introduction

.NET Core est une plate-forme de développement à usage général gérée par Microsoft et la communauté .NET sur GitHub. Il est multi-plateforme, prend en charge Windows, macOS et Linux et peut être utilisé dans des scénarios de périphériques, de cloud et intégrés / IoT.

Lorsque vous pensez à .NET Core, vous devez penser aux éléments suivants (déploiement flexible, outils multi-plateformes, ligne de commande, open source).

Une autre chose intéressante est que même s'il est open source, Microsoft le soutient activement.

Remarques

En soi, .NET Core inclut un modèle d'application unique - les applications de console - qui est utile pour les outils, les services locaux et les jeux de texte. Des modèles d'application supplémentaires ont été construits sur .NET Core pour étendre ses fonctionnalités, telles que:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Xamarin.Forms

En outre, .NET Core implémente la bibliothèque standard .NET et prend donc en charge les bibliothèques standard .NET.

La bibliothèque .NET Standard est une spécification de l'API qui décrit l'ensemble cohérent des API .NET auxquelles les développeurs peuvent s'attendre dans chaque implémentation .NET. Les implémentations .NET doivent implémenter cette spécification pour être considérée comme compatible avec la bibliothèque .NET Standard et pour prendre en charge les bibliothèques qui ciblent la bibliothèque .NET Standard.

Examples

Application de console de base

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine("\nHello, {0}, on {1:d} at {1:t}", name, date);
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
```

```
}
```

Lire .NET Core en ligne: https://riptutorial.com/fr/dot-net/topic/9059/-net-core

Chapitre 3: ADO.NET

Introduction

ADO (ActiveX Data Objects) .Net est un outil fourni par Microsoft qui permet d'accéder à des sources de données telles que SQL Server, Oracle et XML via ses composants. Les applications frontales .Net peuvent récupérer, créer et manipuler des données, une fois qu'elles sont connectées à une source de données via ADO.Net avec les privilèges appropriés.

ADO.Net fournit une architecture sans connexion. C'est une approche sécurisée pour interagir avec une base de données, car la connexion ne doit pas être maintenue pendant toute la session.

Remarques

Une remarque sur le paramétrage des SQL avec `Parameters.AddWithValue` : `AddWithValue` n'est jamais un bon point de départ. Cette méthode repose sur l'inférence du type de données à partir de ce qui est transmis. Avec cela, vous pouvez vous retrouver dans une situation où la conversion empêche votre requête d' utiliser un index . Notez que certains types de données SQL Server, tels que `char` / `varchar` (sans précéder "n") ou `date` n'ont pas de type de données .NET correspondant. Dans ces cas, vous devez utiliser `Add` avec le type de données approprié .

Examples

Exécuter des instructions SQL en tant que commande

```
// Uses Windows authentication. Replace the Trusted_Connection parameter with
// User Id=...;Password=...; to use SQL Server authentication instead. You may
// want to find the appropriate connection string for your server.
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// Most ADO.NET objects are disposable and, thus, require the using keyword.
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // Use parameters instead of string concatenation to add user-supplied
    // values to avoid SQL injection and formatting issues. Explicitly supply datatype.

    // System.Data.SqlDbType is an enumeration. See Note1
    command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // Execute the SQL statement. Use ExecuteScalar and ExecuteReader instead
    // for query that return results (or see the more specific examples, once
    // those have been added).

    connection.Open();
```

```
    command.ExecuteNonQuery();
}
```

Remarque 1: Veuillez consulter l' énumération SqlDbType pour la variante spécifique à MSFT SQL Server.

Note 2: Veuillez voir MySqlDbType Enumeration pour la variante spécifique à MySQL.

Meilleures pratiques - Exécution d'instructions SQL

```
public void SaveNewEmployee(Employee newEmployee)
{
    // best practice - wrap all database connections in a using block so they are always
closed & disposed even in the event of an Exception
    // best practice - retrieve the connection string by name from the app.config or
web.config (depending on the application type) (note, this requires an assembly reference to
System.configuration)
    using(SqlConnection con = new
SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].Connectio

    {
        // best practice - use column names in your INSERT statement so you are not dependent
on the sql schema column order
        // best practice - always use parameters to avoid sql injection attacks and errors if
malformed text is used like including a single quote which is the sql equivalent of escaping
or starting a string (varchar/nvarchar)
        // best practice - give your parameters meaningful names just like you do variables in
your code
        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName,
DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // best practice - always specify the database data type of the column you are
using
            // best practice - check for valid values in your code and/or use a database
constraint, if inserting NULL then use System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value =
newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value =
newEmployee.LastName ?? (object) System.DBNull.Value});

            // best practice - always use the correct types when specifying your parameters,
Value is assigned to a DateTime instance and not a string representation of a Date
            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value =
newEmployee.DateOfBirth });

            // best practice - open your connection as late as possible unless you need to
verify that the database connection is valid and wont fail and the proceeding code execution
takes a long time (not the case here)
            con.Open();
            sc.ExecuteNonQuery();
        }

        // the end of the using block will close and dispose the SqlConnection
        // best practice - end the using block as soon as possible to release the database
connection
    }
}
```

```
// supporting class used as parameter for example
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

Meilleure pratique pour travailler avec ADO.NET

- La règle de base est d'ouvrir la connexion pour un minimum de temps. Fermez la connexion explicitement une fois l'exécution de votre procédure terminée, cela renverra l'objet de connexion au pool de connexions. Taille maximale du pool de connexion par défaut = 100. Le regroupement de connexions améliore les performances de la connexion physique à SQL Server. Regroupement de connexions dans SQL Server
- Enveloppez toutes les connexions de base de données dans un bloc using afin qu'elles soient toujours fermées et éliminées même en cas d'exception. Voir l' utilisation de Statement (C # Reference) pour plus d'informations sur l'utilisation des instructions
- Récupère les chaînes de connexion par nom depuis app.config ou web.config (selon le type d'application)
    - Cela nécessite une référence d'assembly à `System.configuration`
    - Voir Chaînes de connexion et fichiers de configuration pour plus d'informations sur la structure de votre fichier de configuration.
- Toujours utiliser les paramètres pour les valeurs entrantes pour
    - Eviter les attaques par injection SQL
    - Évitez les erreurs si du texte mal formé est utilisé, y compris un guillemet simple qui équivaut à SQL pour échapper ou démarrer une chaîne (varchar / nvarchar)
    - Permettre au fournisseur de base de données de réutiliser les plans de requête (non pris en charge par tous les fournisseurs de bases de données), ce qui augmente l'efficacité
- Lorsque vous travaillez avec des paramètres
    - Le type de paramètres SQL et la non-concordance de taille sont une cause fréquente d'échec d'insertion / mise à jour / sélection
    - Donnez des noms significatifs à vos paramètres SQL, tout comme vous faites des variables dans votre code
    - Indiquez le type de données de base de données de la colonne que vous utilisez, cela garantit que les types de paramètres incorrects ne sont pas utilisés, ce qui pourrait entraîner des résultats inattendus
    - Validez vos paramètres entrants avant de les transmettre à la commande (comme dit le proverbe, " garbage in, garbage out "). Valider les valeurs entrantes le plus tôt possible dans la pile
    - Utilisez les types corrects lors de l'attribution de vos valeurs de paramètre, par exemple: n'affectez pas la valeur de chaîne d'un DateTime, mais assignez une instance DateTime réelle à la valeur du paramètre
    - Spécifiez la taille des paramètres de type chaîne. Cela est dû au fait que SQL Server peut réutiliser les plans d'exécution si le type *et la* taille des paramètres correspondent.

Utilisez -1 pour MAX
    - N'utilisez pas la méthode AddWithValue , la raison principale est qu'il est très facile d'oublier de spécifier le type de paramètre ou la précision / l'échelle si nécessaire. Pour plus d'informations, voir Peut-on arrêter d'utiliser AddWithValue déjà?
- Lors de l'utilisation de connexions de base de données
    - Ouvrez la connexion le plus tard possible et fermez-la dès que possible. Ceci est une directive générale lorsque vous travaillez avec une ressource externe
    - Ne partagez jamais les instances de connexion à la base de données (exemple: le fait qu'un singleton héberge une instance partagée de type `SqlConnection`). Demandez à votre code de toujours créer une nouvelle instance de connexion à la base de données en cas de besoin, puis de supprimer le code d'appel et de le «jeter» lorsqu'il est terminé. La raison en est
        - La plupart des fournisseurs de bases de données disposent d'une sorte de pool de connexions, ce qui fait que la création de nouvelles connexions gérées est peu coûteuse.
        - Il élimine les erreurs futures si le code commence à fonctionner avec plusieurs threads

Utilisation d'interfaces communes pour extraire des classes spécifiques à un fournisseur distant

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
//you will probably get the above two values in the ConnectionStringSettings object from
.config file

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

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

Lire ADO.NET en ligne: https://riptutorial.com/fr/dot-net/topic/3589/ado-net

Chapitre 4: Analyse de date et heure

Examples

ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Notez que le passage de `CultureInfo.CurrentCulture` tant que troisième paramètre est identique à la transmission de `null` . Ou, vous pouvez passer une culture spécifique.

Format des chaînes

La chaîne d'entrée peut être dans n'importe quel format correspondant à la chaîne de format

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Tous les caractères qui ne sont pas des spécificateurs de format sont traités comme des littéraux

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Cas important pour les spécificateurs de format

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

24/11/2015 11:01:30

Notez que les valeurs de mois et de minute ont été analysées dans les mauvaises destinations.

Les chaînes de format à caractère unique doivent être l'un des formats standard

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

Des exceptions

ArgumentNullException

```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// Single-character format strings must be one of the standard formats
var date = DateTime.ParseExact("2015-11-24", "q", null);

// Format strings must match the input exactly* (see next section)
var date = DateTime.ParseExact("2015-11-24", "d", null); // Expects 11/24/2015 or 24/11/2015
for most cultures
```

Gestion de plusieurs formats possibles

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
  new [] { "s", "t", "u", "yyyy-MM-dd" }, // Will succeed as long as input matches one of
these
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

Gérer les différences de culture

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Jour: 11; Mois: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Jour: 10; Mois 11

TryParse

Cette méthode accepte une chaîne en entrée, tente de l'analyser dans un `DateTime` et renvoie un résultat booléen indiquant le succès ou l'échec. Si l'appel réussit, la variable transmise en tant `out` paramètre `out` est renseignée avec le résultat analysé.

Si l'analyse échoue, la variable transmise en tant `out` paramètre `out` est définie sur la valeur par défaut, `DateTime.MinValue` .

TryParse (string, out DateTime)

```
DateTime parsedValue;
```

```
if (DateTime.TryParse("monkey", out parsedValue))
{
   Console.WriteLine("Apparently, 'monkey' is a date/time value. Who knew?");
}
```

Cette méthode tente d'analyser la chaîne d'entrée en fonction des paramètres régionaux du système et des formats connus tels que ISO 8601 et d'autres formats courants.

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

Comme cette méthode n'accepte pas les informations de culture, elle utilise les paramètres régionaux du système. Cela peut conduire à des résultats inattendus.

```
// System set to en-US culture
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

Faux

```
// System set to en-GB culture
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

Faux

```
// System set to en-GB culture
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

Vrai

Notez que si vous êtes aux États-Unis, vous pourriez être surpris que le résultat analysé soit le 10 novembre et non le 11 octobre.

TryParse (chaîne, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
    Console.WriteLine("Apparently, ' monkey ' is a date/time value. Who knew?");
}
```

Contrairement à sa méthode fraternelle, cette surcharge permet de spécifier une culture et un style spécifiques. La `IFormatProvider` `null` pour le paramètre `IFormatProvider` utilise la culture système.

Des exceptions

Notez qu'il est possible que cette méthode lance une exception sous certaines conditions. Celles-ci concernent les paramètres introduits pour cette surcharge: `IFormatProvider` et `DateTimeStyles` .

- `NotSupportedException` : `IFormatProvider` spécifie une culture neutre
- `ArgumentException` : `DateTimeStyles` n'est pas une option valide ou contient des indicateurs incompatibles tels que `AssumeLocal` et `AssumeUniversal` .

TryParseExact

Cette méthode se comporte comme une combinaison de `TryParse` et de `ParseExact` : elle permet de `ParseExact` des formats personnalisés et renvoie un résultat booléen indiquant le succès ou l'échec au lieu de générer une exception si l'analyse échoue.

TryParseExact (chaîne, chaîne, IFormatProvider, DateTimeStyles, out DateTime)

Cette surcharge tente d'analyser la chaîne d'entrée en fonction d'un format spécifique. La chaîne d'entrée doit correspondre à ce format pour pouvoir être analysée.

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); //
true
```

TryParseExact (string, string [], IFormatProvider, DateTimeStyles, out DateTime)

Cette surcharge tente d'analyser la chaîne d'entrée par rapport à un tableau de formats. La chaîne d'entrée doit correspondre à au moins un format pour pouvoir être analysée.

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null,
DateTimeStyles.None, out parsedValue); // true
```

Lire Analyse de date et heure en ligne: https://riptutorial.com/fr/dot-net/topic/58/analyse-de-date-et-heure

Chapitre 5: Arbres d'expression

Remarques

Les arbres d'expression sont des structures de données utilisées pour représenter des expressions de code dans .NET Framework. Ils peuvent être générés par code et parcourus par programme pour traduire le code dans une autre langue ou l'exécuter. Le générateur le plus populaire des arbres d'expression est le compilateur C # lui-même. Le compilateur C # peut générer des arborescences d'expression si une expression lambda est affectée à une variable de type Expression <Func <... >>. Habituellement, cela se produit dans le contexte de LINQ. Le consommateur le plus populaire est le fournisseur LINQ d'Entity Framework. Il consomme les arbres d'expression donnés à Entity Framework et génère un code SQL équivalent qui est ensuite exécuté sur la base de données.

Examples

Arbre d'expression simple généré par le compilateur C

Considérons le code C # suivant

```
Expression<Func<int, int>> expression = a => a + 1;
```

Comme le compilateur C # voit que l'expression lambda est affectée à un type d'expression plutôt qu'à un type délégué, il génère un arbre d'expression à peu près équivalent à ce code

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
                                                Expression.Add(
                                                    parameterA,
                                                    Expression.Constant(1)),
                                                parameterA);
```

La racine de l'arbre est l'expression lambda qui contient un corps et une liste de paramètres. Le lambda a 1 paramètre appelé "a". Le corps est une expression unique du type CLR BinaryExpression et NodeType of Add. Cette expression représente l'addition. Il a deux sous-expressions désignées par Gauche et Droite. Left est l'expression de paramètre pour le paramètre "a" et right est une expression constante avec la valeur 1.

L'utilisation la plus simple de cette expression consiste à l'imprimer:

```
Console.WriteLine(expression); //prints a => (a + 1)
```

Qui imprime le code C # équivalent.

L'arbre d'expression peut être compilé en un délégué C # et exécuté par le CLR

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //prints 3
```

Habituellement, les expressions sont traduites dans d'autres langages tels que SQL, mais peuvent également être utilisées pour invoquer des membres privés, protégés et internes de types publics ou non publics, en alternative à Reflection.

construire un prédicat de champ de formulaire == valeur

Pour construire une expression comme `_ => _.Field == "VALUE"` à l'exécution.

Étant donné un prédicat `_ => _.Field` et une valeur de chaîne `"VALUE"` , créez une expression qui vérifie si le prédicat est vrai ou non.

L'expression convient pour:

- `IQueryable<T>` , `IEnumerable<T>` pour tester le prédicat.
- framework d'entité ou `Linq` to `SQL` pour créer une clause `Where` qui teste le prédicat.

Cette méthode va générer une expression `Equal` appropriée qui teste si `Field` est égal ou non à `"VALUE"` .

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
        expression,
        memberAccessor.Parameters);
    return predicate;
}
```

Le prédicat peut être utilisé en incluant le prédicat dans une méthode d'extension `Where` .

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

Expression pour récupérer un champ statique

Ayant un exemple de type comme ceci:

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

Nous pouvons récupérer la valeur de StaticPublicField:

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

Il peut alors être compilé dans un délégué pour récupérer la valeur du champ.

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

// Le résultat de fieldValue est StaticPublicFieldValue

Classe InvocationExpression

La classe InvocationExpression permet l'invocation d'autres expressions lambda qui font partie du même arbre d'expression.

Vous les créez avec la méthode statique `Expression.Invoke` .

Problème Nous voulons obtenir les éléments qui ont "voiture" dans leur description. Nous devons vérifier la valeur null avant de chercher une chaîne à l'intérieur, mais nous ne voulons pas qu'elle soit appelée de manière excessive, car le calcul pourrait être coûteux.

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
            whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
            Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
```

```
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

        Console.WriteLine("Property extractor is called {0} times.",
countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
        count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
            Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

        Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
                Expression.AndAlso(
                    Expression.Not(
                        Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
                    Expression.Call(extracted, "Contains", null,
Expression.Constant(searchTerm))
                ),
                extracted);

        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
            Expression.Invoke(
                coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
            elementParameter
        );
    }
}
```

Sortie

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso
extracted.Contains("car")), Invoke(e => e.Description, element))
car
cargo
Madagascar
Predicate is called 5 times.
```

La première chose à noter est la manière dont l'accès à la propriété, enveloppé dans un Invoke:

```
Invoke(e => e.Description, element)
```

, et c'est la seule partie qui touche `e.Description` , et à la place, le paramètre `extracted` de type `string` est passé au suivant:

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

Une autre chose importante à noter ici est `AndAlso` . Il ne calcule que la partie gauche, si la première partie renvoie "false". C'est une erreur courante d'utiliser l'opérateur binaire «And» au lieu de cela, qui calcule toujours les deux parties, et qui échouerait avec une exception NullReferenceException dans cet exemple.

Lire Arbres d'expression en ligne: https://riptutorial.com/fr/dot-net/topic/2657/arbres-d-expression

Chapitre 6: Bibliothèque parallèle de tâches (TPL)

Remarques

Buts et cas d'utilisation

Le but de la bibliothèque parallèle de tâches est de simplifier le processus d'écriture et de maintenance du code multithread et parallèle.

Quelques cas d'utilisation *:

- Garder une interface utilisateur réactive en exécutant un travail d'arrière-plan sur une tâche distincte
- Répartition de la charge de travail
- Autoriser une application cliente à envoyer et recevoir des requêtes en même temps (reste, TCP / UDP, ect)
- Lire et / ou écrire plusieurs fichiers à la fois

* Le code doit être considéré au cas par cas pour le multithreading. Par exemple, si une boucle ne comporte que quelques itérations ou ne fait qu'une petite partie du travail, la surcharge du parallélisme peut dépasser les avantages.

TPL avec .Net 3.5

Le TPL est également disponible pour .Net 3.5 inclus dans un package NuGet, il s'appelle Task Parallel Library.

Examples

Boucle de base producteur-consommateur (BlockingCollection)

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
        collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

Il est à noter que si vous n'appelez pas `collection.CompleteAdding();` , vous pouvez continuer à ajouter à la collection même si votre tâche client est en cours d'exécution. Il suffit d'appeler `collection.CompleteAdding();` lorsque vous êtes sûr qu'il n'y a plus d'ajouts. Cette fonctionnalité peut être utilisée pour créer un modèle Producteur multiple vers un consommateur unique dans lequel vous disposez de plusieurs sources alimentant des éléments dans BlockingCollection et un seul consommateur en retirant des éléments et en faisant quelque chose. Si votre BlockingCollection est vide avant que vous appeliez l'ajout complet, le Enumerable de `collection.GetConsumingEnumerable()` bloquera jusqu'à ce qu'un nouvel élément soit ajouté à la collection ou à BlockingCollection.CompleteAdding (); est appelé et la file d'attente est vide.

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
        Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    Console.WriteLine("Consumer completed!");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("Everything completed!");
```

Tâche: instanciation de base et attente

Une tâche peut être créée en instanciant directement la classe de `Task` ...

```
var task = new Task(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});

Console.WriteLine("Starting task...");
task.Start();
task.Wait();
Console.WriteLine("Task completed!");
```

... ou en utilisant la méthode `Task.Run` statique:

```
Console.WriteLine("Starting task...");
var task = Task.Run(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});
task.Wait();
Console.WriteLine("Task completed!");
```

Notez que seulement dans le premier cas, il est nécessaire d'appeler explicitement `Start` .

Tâche: WaitAll et capture de variable

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);

foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

Tâche: WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("Task {0} finished", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

Note: Le `WaitAll` final est nécessaire car `WaitAny` ne fait pas observer d'exceptions.

Tâche: gestion des exceptions (à l'aide de l'attente)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

Console.WriteLine("Starting tasks...");
try
{
    Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
    Console.WriteLine("Task(s) failed!");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
```

```
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Tâche: gérer les exceptions (sans utiliser Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

var tasks = new[] {task1, task2};

Console.WriteLine("Starting tasks...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
        Console.WriteLine("Task failed: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Tâche: annuler en utilisant CancellationToken

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
            Console.Write("{0} ", i++);
            Thread.Sleep(1000);
            myCancellationToken.ThrowIfCancellationRequested();
        }
    },
    cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();
```

```
cancellationTokenSource.Cancel();
try
{
    task.Wait();
}
catch(AggregateException ex)
{
    ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//Canceled
```

Comme alternative à `ThrowIfCancellationRequested` , la demande d'annulation peut être détectée avec `IsCancellationRequested` et une `IsCancellationRequested` `OperationCanceledException` peut être lancée manuellement:

```
//New task delegate
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
    Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}Ouch, I have been cancelled!!");
throw new OperationCanceledException(myCancellationToken);
```

Notez comment le jeton d'annulation est transmis au constructeur de tâches dans le paramètre `cancellationToken` . Cela est nécessaire pour que la tâche passe à l'état `Canceled` , et non à l'état `Faulted` , lorsque `ThrowIfCancellationRequested` est `ThrowIfCancellationRequested` . De plus, pour la même raison, le jeton d'annulation est explicitement fourni dans le constructeur d' `OperationCanceledException` dans le second cas.

Task.WhenAny

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
    Console.WriteLine("I'm task " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("The winner is: task " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("All tasks finished!");
```

Task.WhenAll

```
var random = new Random();
```

```
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
}));

Task<int[]> task = Task.WhenAll(tasks);
int[] results = await task;

Console.WriteLine(string.Join(",", results.Select(n => n.ToString())));
// Output: 1,2,3,4,5
```

Parallel.Invoke

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
    Console.WriteLine("I'm task " + n);
    if((n & 1) == 0)
        throw new Exception("Exception from task " + n);
})).ToArray();

try
{
    Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("Task failed: " + inner.Message);
}
```

Parallel.ForEach

Cet exemple utilise `Parallel.ForEach` pour calculer la somme des nombres entre 1 et 10000 en utilisant plusieurs threads. Pour atteindre la sécurité des threads, `Interlocked.Add` est utilisé pour additionner les nombres.

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Parallel.For

Cet exemple utilise `Parallel.For` pour calculer la somme des nombres entre 1 et 10000 en utilisant plusieurs threads. Pour atteindre la sécurité des threads, `Interlocked.Add` est utilisé pour

additionner les nombres.

```
using System.Threading;

int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Contexte d'exécution fluide avec AsyncLocal

Lorsque vous devez transmettre des données de la tâche parente à ses tâches enfants, de manière à ce `AsyncLocal` à l'exécution, utilisez la classe `AsyncLocal` :

```
void Main()
{
    AsyncLocal<string> user = new AsyncLocal<string>();
    user.Value = "initial user";

    // this does not affect other tasks - values are local relative to the branches of
execution flow
    Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
        Console.WriteLine(user.Value); // outputs "initial user"
        Task.Run(() =>
        {
            // outputs "initial user" - value has flown from main method to this task without
being changed
            Console.WriteLine(user.Value);
        }).Wait();

        user.Value = "user from task1";

        Task.Run(() =>
        {
            // outputs "user from task1" - value has flown from main method to task1
            // than value was changed and flown to this task.
            Console.WriteLine(user.Value);
        }).Wait();
    });

    task1.Wait();

    // ouputs "initial user" - changes do not propagate back upstream the execution flow
    Console.WriteLine(user.Value);
}
```

Note: Comme on peut le voir dans l'exemple ci-dessus, `AsynLocal.Value` a une `copy on read` sémantique de `copy on read` , mais si vous `AsynLocal.Value` un type de référence et modifiez ses propriétés, vous affecterez d'autres tâches. Par conséquent, la meilleure pratique avec `AsyncLocal`

consiste à utiliser des types de valeur ou des types immuables.

Parallel.ForEach dans VB.NET

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                               ProcessRowParallel(currentRow, state)
                                           End Sub)
```

Tâche: Renvoyer une valeur

La tâche qui renvoie une valeur a le type de retour de la `Task< TResult >` où TResult est le type de valeur à renvoyer. Vous pouvez interroger le résultat d'une tâche par sa propriété Result.

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

Si la tâche est exécutée de manière asynchrone dans l'attente de la tâche, le résultat est renvoyé.

```
public async Task DoSomeWork()
{
    WebClient client = new WebClient();
    // Because the task is awaited, result of the task is assigned to response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

Lire Bibliothèque parallèle de tâches (TPL) en ligne: https://riptutorial.com/fr/dot-net/topic/55/bibliotheque-parallele-de-taches--tpl-

Chapitre 7: Cadre d'extensibilité gérée

Remarques

L'un des grands avantages de MEF par rapport aux autres technologies prenant en charge le modèle d'inversion de contrôle est qu'il prend en charge la résolution des dépendances qui ne sont pas connues au moment de la conception, sans nécessiter beaucoup de configuration (le cas échéant).

Tous les exemples nécessitent une référence à l'assembly System.ComponentModel.Composition.

De plus, tous les exemples (de base) les utilisent comme exemples d'objets métier:

```
using System.Collections.ObjectModel;

namespace Demo
{
    public sealed class User
    {
        public User(int id, string name)
        {
            this.Id = id;
            this.Name = name;
        }

        public int Id { get; }
        public string Name { get; }
        public override string ToString() => $"User[Id: {this.Id}, Name={this.Name}]";
    }

    public interface IUserProvider
    {
        ReadOnlyCollection<User> GetAllUsers();
    }
}
```

Examples

Exportation d'un type (de base)

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
        public ReadOnlyCollection<User> GetAllUsers()
        {
```

```
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

Cela pourrait être défini pratiquement n'importe où; tout ce qui compte, c'est que l'application sache où chercher (via les ComposablePartCatalogs créés).

Importation (de base)

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
                Console.WriteLine(user);
            }
        }
    }
}
```

C'est un type dépendant d'un `IUserProvider` , qui peut être défini n'importe où. Comme dans l'exemple précédent, tout ce qui compte, c'est que l'application sache où chercher l'exportation correspondante (via les ComposablePartCatalogs qu'elle crée).

Connexion (basique)

Voir les autres exemples ci-dessus.

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
```

```
            {
                exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // at this point, writer's userProvider field is null
                container.ComposeParts(writer);

                // now, it should be non-null (or an exception will be thrown).
                writer.PrintAllUsers();
            }
        }
    }
}
```

Tant que quelque chose dans le chemin de recherche de l'assembly de l'application comporte `[Export(typeof(IUserProvider))]` , l'importation correspondante de `UserWriter` sera satisfaite et les utilisateurs seront imprimés.

D'autres types de catalogues (par exemple, `DirectoryCatalog`) peuvent être utilisés à la place (ou en plus) d' `ApplicationCatalog` , pour rechercher dans d'autres emplacements des exportations satisfaisant les importations.

Lire Cadre d'extensibilité gérée en ligne: https://riptutorial.com/fr/dot-net/topic/62/cadre-d-extensibilite-geree

Chapitre 8: Chiffrement / Cryptographie

Remarques

.NET Framework fournit l'implémentation de nombreux algorithmes cryptographiques. Ils comprennent essentiellement des algorithmes symétriques, des algorithmes asymétriques et des hachages.

Examples

RijndaelManaged

Espace de noms requis: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
            RijndaelManaged aesAlg = null;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor,
CryptoStreamMode.Write)) {
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
```

```
                }
              }
            } finally {
              aesAlg?.Clear();
            }
            return encrypt.ToArray();
          }

          private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
            RijndaelManaged aesAlg = null;
            string plaintext;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
              using (var msDecrypt = new MemoryStream(cipherText)) {
                using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                  using (var srDecrypt = new StreamReader(csDecrypt))
                    plaintext = srDecrypt.ReadToEnd();
                }
              }
            } finally {
              aesAlg?.Clear();
            }
            return plaintext;
          }
        }
```

Usage

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

Remarque:

- Rijndael est le prédécesseur de l'algorithme cryptographique symétrique standard AES.

Chiffrer et déchiffrer les données à l'aide d'AES (en C #)

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
```

```
            string original = "Here is some data to encrypt!";

            // Create a new instance of the Aes class.
            // This generates a new key and initialization vector (IV).
            using (Aes myAes = Aes.Create())
            {
                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                            myAes.Key,
                                                            myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                              myAes.Key,
                                                              myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original:   {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e.Message);
        }
    }

    static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
    {
        // Check arguments.
        if (plainText == null || plainText.Length <= 0)
            throw new ArgumentNullException("plainText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        byte[] encrypted;

        // Create an Aes object with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decrytor to perform the stream transform.
            ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                aesAlg.IV);

            // Create the streams used for encryption.
            using (MemoryStream msEncrypt = new MemoryStream())
            {
                using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                                 encryptor,
                                                                 CryptoStreamMode.Write))
                {
                    using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                    {
                        //Write all data to the stream.
                        swEncrypt.Write(plainText);
                    }
```

```
                    encrypted = msEncrypt.ToArray();
                }
            }
        }

        // Return the encrypted bytes from the memory stream.
        return encrypted;
    }

    static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
    {
        // Check arguments.
        if (cipherText == null || cipherText.Length <= 0)
            throw new ArgumentNullException("cipherText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        // Declare the string used to hold the decrypted text.
        string plaintext = null;

        // Create an Aes object with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decrytor to perform the stream transform.
            ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                aesAlg.IV);

            // Create the streams used for decryption.
            using (MemoryStream msDecrypt = new MemoryStream(cipherText))
            {
                using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                 decryptor,
                                                                 CryptoStreamMode.Read))
                {
                    using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                    {

                        // Read the decrypted bytes from the decrypting stream
                        // and place them in a string.
                        plaintext = srDecrypt.ReadToEnd();
                    }
                }
            }
        }

        return plaintext;
    }
}
}
```

Cet exemple provient de MSDN .

Il s’agit d’une application de démonstration de console qui montre comment chiffrer une chaîne à l’aide du chiffrement **AES** standard et comment la déchiffrer par la suite.

(**AES = Advanced Encryption Standard** , une spécification pour le cryptage des données électroniques établie par l'Institut national américain des normes et de la technologie (NIST) en 2001, qui reste la norme de facto pour le cryptage symétrique)

Remarques:

- Dans un scénario de chiffrement réel, vous devez choisir un mode de chiffrement approprié (pouvant être attribué à la propriété `Mode` en sélectionnant une valeur dans l'énumération `CipherMode`). **N'utilisez jamais** le mode `CipherMode.ECB` (mode livre de codes électronique), car cela génère un flux de chiffrement faible
- Pour créer une `Key` (et non une `Key` faible), utilisez un générateur aléatoire cryptographique ou utilisez l'exemple ci-dessus (**Créer une clé à partir d'un mot de passe**). La **taille de clé** recommandée est de 256 bits. Les tailles de clé prises en charge sont disponibles via la propriété `LegalKeySizes` .
- Pour initialiser le vecteur d'initialisation `IV` , vous pouvez utiliser un SALT comme indiqué dans l'exemple ci-dessus (**SALT aléatoire**)
- Les tailles de bloc prises en charge sont disponibles via la propriété `SupportedBlockSizes` , la taille du bloc peut être attribuée via la propriété `BlockSize` .

Utilisation: voir la méthode Main ().

Créer une clé à partir d'un mot de passe / SALT aléatoire (en C #)

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // Get a password from the user.
        Console.WriteLine("Enter a password to produce a key:");

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // Create a TripleDESCryptoServiceProvider object.
        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
            Console.WriteLine("Creating a key with PasswordDeriveBytes...");

            // Create a PasswordDeriveBytes object and then create
            // a TripleDES key from the password and salt.
            PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
```

```
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("Operation complete.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // Clear the buffers
            ClearBytes(pwd);
            ClearBytes(salt);

            // Clear the key.
            tdes.Clear();
        }

        Console.ReadLine();
    }

    #region Helper methods

    /// <summary>
    /// Generates a random salt value of the specified length.
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // Create a buffer
        byte[] randBytes;

        if (length >= 1)
        {
            randBytes = new byte[length];
        }
        else
        {
            randBytes = new byte[1];
        }

        // Create a new RNGCryptoServiceProvider.
        RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // Fill the buffer with random bytes.
        rand.GetBytes(randBytes);

        // return the bytes.
        return randBytes;
    }

    /// <summary>
    /// Clear the bytes in a buffer so they can't later be read from memory.
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // Check arguments.
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }
```

```
        // Set each byte in the buffer to 0.
        for (int x = 0; x < buffer.Length; x++)
        {
            buffer[x] = 0;
        }
    }

    #endregion
}
```

Cet exemple provient de MSDN.

Il s'agit d'une démonstration de console qui montre comment créer une clé sécurisée basée sur un mot de passe défini par l'utilisateur et comment créer un SALT aléatoire basé sur le générateur aléatoire cryptographique.

Remarques:

- La fonction intégrée `PasswordDeriveBytes` utilise l'algorithme standard PBKDF1 pour générer une clé à partir du mot de passe. Par défaut, il utilise 100 itérations pour générer la clé permettant de ralentir les attaques par force brute. Le SALT généré aléatoirement renforce la clé.
- La fonction `CryptDeriveKey` convertit la clé générée par `PasswordDeriveBytes` en une clé compatible avec l'algorithme de chiffrement spécifié (ici "TripleDES") en utilisant l'algorithme de hachage spécifié (ici "SHA1"). La clé dans cet exemple est 192 octets, et le vecteur d'initialisation IV provient du fournisseur de cryptage triple-DES.
- Généralement, ce mécanisme est utilisé pour protéger une clé générée aléatoirement par un mot de passe, qui crypte une grande quantité de données. Vous pouvez également l'utiliser pour fournir plusieurs mots de passe d'utilisateurs différents pour donner accès aux mêmes données (protégées par une autre clé aléatoire).
- Malheureusement, `CryptDeriveKey` ne prend actuellement pas en charge AES. Voir ici **REMARQUE: pour** résoudre ce problème, vous pouvez créer une clé AES aléatoire pour le chiffrement des données à protéger avec AES et stocker la clé AES dans un conteneur TripleDES qui utilise la clé générée par `CryptDeriveKey` . Mais cela limite la sécurité à TripleDES, ne profite pas des plus grandes tailles de clés d'AES et crée une dépendance à TripleDES.

Utilisation: Voir la méthode Main ().

Cryptage et décryptage à l'aide de la cryptographie (AES)

Code de déchiffrement

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;
```

```
        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

                cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

Code de cryptage

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }

                cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

Usage

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);
```

```
var decrypted = Decrypt(encrypted);
```

Lire Chiffrement / Cryptographie en ligne: https://riptutorial.com/fr/dot-net/topic/7615/chiffrement---cryptographie

Chapitre 9: Classe SpeechRecognitionEngine pour reconnaître la parole

Syntaxe

- SpeechRecognitionEngine ()
- SpeechRecognitionEngine.LoadGrammar (Grammaire grammaticale)
- SpeechRecognitionEngine.SetInputToDefaultAudioDevice ()
- SpeechRecognitionEngine.RecognizeAsync (mode RecognizeMode)
- Constructeur de grammaire()
- GrammarBuilder.Append (choix de choix)
- Choix (params string [] choix)
- Grammaire (générateur GrammarBuilder)

Paramètres

`LoadGrammar` : Paramètres	Détails
grammaire	La grammaire à charger. Par exemple, un objet `DictationGrammar` pour autoriser la dictée de texte libre.
`RecognizeAsync` : Paramètres	**Détails**
mode	Le `RecognizeMode` pour la reconnaissance actuelle: `Single` pour une seule reconnaissance, `Multiple` pour autoriser plusieurs.
`GrammarBuilder.Append` : Paramètres	**Détails**
les choix	Ajoute quelques choix au générateur de grammaire. Cela signifie que, lorsque l'utilisateur saisit la parole, le dispositif de reconnaissance peut suivre différentes "branches" d'une grammaire.
Constructeur de `Choices` : paramètres	**Détails**
les choix	Un tableau de choix pour le générateur de grammaire. Voir `GrammarBuilder.Append` .
Constructeur de `Grammar` : Paramètre	**Détails**
constructeur	Le `GrammarBuilder` pour construire une `Grammar` partir de.

Remarques

Pour utiliser `SpeechRecognitionEngine` , votre version de Windows doit avoir la reconnaissance vocale activée.

Vous devez ajouter une référence à `System.Speech.dll` avant de pouvoir utiliser les classes de conversation.

Examples

Reconnaissance asynchrone de la parole pour la dictée de texte libre

```
using System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Reconnaissance asynchrone de la parole basée sur un ensemble restreint de phrases

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("I am", "You are", "He is", "She is", "We are", "They are"));
builder.Append(new Choices("friendly", "unfriendly"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Lire Classe SpeechRecognitionEngine pour reconnaître la parole en ligne: https://riptutorial.com/fr/dot-net/topic/69/classe-speechrecognitionengine-pour-reconnaitre-la-parole

Chapitre 10: Classe System.IO.File

Syntaxe

- source de chaîne;
- destination de chaîne;

Paramètres

Paramètre	Détails
`source`	Le fichier à déplacer vers un autre emplacement.
`destination`	Le répertoire dans lequel vous souhaitez déplacer la `source` (cette variable doit également contenir le nom (et l'extension de fichier) du fichier.

Examples

Supprimer un fichier

Supprimer un fichier (si vous avez les autorisations requises) est aussi simple que:

```
File.Delete(path);
```

Cependant, beaucoup de choses peuvent mal tourner:

- Vous n'avez pas les autorisations requises (une `UnauthorizedAccessException` est levée).
- Le fichier peut être utilisé par quelqu'un d'autre (`IOException` est lancé).
- Le fichier ne peut pas être supprimé en raison d'une erreur de bas niveau ou d'un support en lecture seule (`IOException` est `IOException`).
- Le fichier n'existe plus (`IOException` est `IOException`).

Notez que le dernier point (le fichier n'existe pas) est généralement *contourné* avec un extrait de code comme celui-ci:

```
if (File.Exists(path))
    File.Delete(path);
```

Cependant, ce n'est pas une opération atomique et le fichier peut être supprimé par quelqu'un d'autre entre l'appel à `File.Exists()` et avant `File.Delete()` . L'approche appropriée pour gérer les opérations d'E / S nécessite la gestion des exceptions (en supposant qu'une autre procédure puisse être prise en cas d'échec de l'opération):

```
if (File.Exists(path))
```

```
{
    try
    {
        File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // Someone else deleted this file

        // Something went wrong...
    }
    catch (UnauthorizedAccessException exception)
    {
        // I do not have required permissions
    }
}
```

Notez que ces erreurs d'E / S sont parfois transitoires (fichier en cours d'utilisation, par exemple) et qu'une connexion réseau est impliquée, elle peut alors se rétablir automatiquement sans aucune action de notre part. Il est alors courant de *réessayer* plusieurs fois une opération d'E / S avec un léger délai entre chaque tentative:

```
public static void Delete(string path)
{
    if (!File.Exists(path))
        return;

    for (int i=1; ; ++i)
    {
        try
        {
            File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

        // You may handle UnauthorizedAccessException but this issue
        // will probably won't be fixed in few seconds...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // ms
```

Remarque: dans l'environnement Windows, le fichier ne sera pas réellement effacé lorsque vous appelez cette fonction, si quelqu'un d'autre ouvre le fichier en utilisant `FileShare.Delete` alors le fichier peut être supprimé mais cela ne se produira effectivement que lorsque le propriétaire

fermera le fichier.

Supprimer les lignes indésirables d'un fichier texte

Changer de fichier texte n'est pas facile car son contenu doit être déplacé. Pour les *petits* fichiers, la méthode la plus simple consiste à lire son contenu en mémoire, puis à écrire le texte modifié.

Dans cet exemple, nous lisons toutes les lignes d'un fichier et supprimons toutes les lignes vides, puis nous réécrivons le chemin d'origine:

```
File.WriteAllLines(path,
    File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Si le fichier est trop gros pour le charger en mémoire et que le chemin de sortie est différent du chemin d'entrée:

```
File.WriteAllLines(outputPath,
    File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Convertir l'encodage du fichier texte

Le texte est enregistré codé (voir aussi le sujet Chaînes), parfois vous devrez peut-être modifier son encodage, cet exemple suppose (pour simplifier) que ce fichier n'est pas trop gros et qu'il peut être entièrement lu en mémoire:

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
    File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```

Lorsque vous effectuez des conversions ne faut pas oublier que le fichier peut contenir BOM (Byte Order Mark), afin de mieux comprendre comment il a réussi à se référer Encoding.UTF8.GetString ne prend pas en compte le préambule / nomenclature .

"Touchez" une grande quantité de fichiers (pour mettre à jour la dernière heure d'écriture)

Cet exemple met à jour la dernière heure d'écriture d'un grand nombre de fichiers (à l'aide de `System.IO.Directory.EnumerateFiles` au lieu de `System.IO.Directory.GetFiles()`). Vous pouvez éventuellement spécifier un modèle de recherche (la valeur par défaut est `"*.*"` Et éventuellement rechercher dans une arborescence de répertoires (pas uniquement dans le répertoire spécifié):

```
public static void Touch(string path,
                         string searchPattern = "*.*",
                         SearchOptions options = SearchOptions.None)
{
    var now = DateTime.Now;

    foreach (var filePath in Directory.EnumerateFiles(path, searchPattern, options))
```

```
    {
        File.SetLastWriteTime(filePath, now);
    }
}
```

Énumérer les fichiers antérieurs à une quantité spécifiée

Cet extrait de code est une fonction d'assistance permettant d'énumérer tous les fichiers plus anciens qu'un âge spécifié. Il est utile, par exemple, pour supprimer d'anciens fichiers journaux ou d'anciennes données en cache.

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                                TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
                                SearchOption options = SearchOption.TopDirectoryOnly)
{
    DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

Utilisé comme ceci:

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

Peu de choses à noter:

- La recherche est effectuée à l'aide de `Directory.EnumerateFiles()` au lieu de `Directory.GetFiles()` . L'énumération est *active,* vous n'avez donc pas besoin d'attendre que toutes les entrées du système de fichiers aient été récupérées.
- Nous vérifions l'heure de la dernière écriture, mais vous pouvez utiliser l'heure de création ou la dernière heure d'accès (par exemple, pour supprimer les fichiers cachés *inutilisés* , notez que le temps d'accès peut être désactivé).
- La granularité n'est pas uniforme pour toutes ces propriétés (temps d'écriture, temps d'accès, temps de création), consultez MSDN pour plus de détails à ce sujet.

Déplacer un fichier d'un endroit à un autre

File.Move

Pour déplacer un fichier d'un emplacement à un autre, une simple ligne de code peut y parvenir:

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

Cependant, il y a beaucoup de choses qui pourraient mal tourner avec cette opération simple. Par exemple, que se passe-t-il si l'utilisateur qui exécute votre programme ne dispose pas d'un lecteur nommé «C»? Et s'ils le faisaient, mais ils ont décidé de le renommer en «B» ou «M»?

Que se passe-t-il si le fichier source (le fichier dans lequel vous souhaitez déplacer) a été déplacé à votre insu - ou s'il n'existe tout simplement pas?

Cela peut être contourné en vérifiant d'abord si le fichier source existe:

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
    File.Move(source, destination);
}
```

Cela garantira que le fichier existe à ce moment-là et peut être déplacé vers un autre emplacement. Parfois, un simple appel à `File.Exists` ne suffira pas. Si ce n'est pas le cas, vérifiez à nouveau, indiquez à l'utilisateur que l'opération a échoué ou gérez l'exception.

Une `FileNotFoundException` n'est pas la seule exception que vous êtes susceptible de rencontrer.

Voir ci-dessous pour les exceptions possibles:

Type d'exception	La description
`IOException`	Le fichier existe déjà ou le fichier source est introuvable.
`ArgumentNullException`	La valeur des paramètres Source et / ou Destination est nulle.
`ArgumentException`	La valeur des paramètres Source et / ou Destination est vide ou contient des caractères non valides.
`UnauthorizedAccessException`	Vous n'avez pas les autorisations requises pour effectuer cette action.
`PathTooLongException`	La source, la destination ou le ou les chemins spécifiés dépassent la longueur maximale. Sous Windows, la longueur d'un chemin doit être inférieure à 248 caractères, alors que les noms de fichiers doivent comporter moins de 260 caractères.
`DirectoryNotFoundException`	Le répertoire spécifié est introuvable.
`NotSupportedException`	Les chemins d'accès ou les noms de fichiers source ou destination sont dans un format non valide.

Lire Classe System.IO.File en ligne: https://riptutorial.com/fr/dot-net/topic/5395/classe-system-io-file

Chapitre 11: Clients HTTP

Remarques

Les RFC HTTP / 1.1 actuellement pertinentes sont:

- 7230: Syntaxe et routage des messages
- 7231: Sémantique et contenu
- 7232: demandes conditionnelles
- 7233: demandes de plage
- 7234: mise en cache
- 7235: Authenticaion
- 7239: Extension HTTP transférée
- 7240: Préférer l'en-tête pour HTTP

Il y a aussi les RFC d'information suivants:

- 7236: Enregistrements de schémas d'authentification
- 7237: Enregistrements de méthode

Et la RFC expérimentale:

- 7238: Code d'état du protocole de transfert hypertexte 308 (redirection permanente)

Protocoles connexes:

- 4918: extensions HTTP pour WebDAV (Web Distributed Authoring and Versioning)
- 4791: Extensions de calendrier à WebDAV (CalDAV)

Examples

Lecture de la réponse GET sous forme de chaîne à l'aide de System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
    responseData = responseReader.ReadToEnd();
}
```

Lecture de la réponse GET sous forme de chaîne à l'aide de System.Net.WebClient

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
    responseData = client.DownloadString(requestUri);
}
```

Lecture de la réponse GET sous forme de chaîne à l'aide de System.Net.HttpClient

`HttpClient` est disponible via NuGet: Microsoft HTTP Client Libraries .

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
       response.EnsureSuccessStatusCode();
       responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

Envoi d'une requête POST avec une charge utile de chaîne à l'aide de System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
  Method = requestMethod,
  ContentType = contentType,
};

byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

Envoi d'une requête POST avec une charge utile de chaîne à l'aide de System.Net.WebClient

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";
```

```
byte[] responseBody;
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
    client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

Envoi d'une requête POST avec une charge utile de chaîne à l'aide de System.Net.HttpClient

`HttpClient` est disponible via NuGet: Microsoft HTTP Client Libraries .

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
    RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```

Downloader HTTP basique utilisant System.Net.Http.HttpClient

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
                Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
                    Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
                }
            }
```

```
        }
    }

    static void Main(string[] args)
    {
        try
        {
            Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
                ex = ((AggregateException)ex).Flatten().InnerExceptions.First();

            Console.WriteLine("--- Error: " +
                (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Basic HTTP downloader");
            Console.WriteLine();
            Console.WriteLine("Usage: httpget <url>[<:port>] <file>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

        Console.WriteLine("Done!");
    }
}
```

Lire Clients HTTP en ligne: https://riptutorial.com/fr/dot-net/topic/32/clients-http

Chapitre 12: CLR

Examples

Introduction au Common Language Runtime

Common Language Runtime (CLR) est un environnement de machine virtuelle et une partie du .NET Framework. Il contient:

- Un langage de bytecode portable appelé **Common Intermediate Language** (**langage** abrégé CIL ou IL)
- Un compilateur Just-In-Time qui génère du code machine
- Un ramasse-miettes de suivi fournissant une gestion automatique de la mémoire
- Prise en charge de sous-processus légers appelés AppDomains
- Mécanismes de sécurité à travers les concepts de code vérifiable et de niveaux de confiance

Le code qui s'exécute dans le CLR est appelé *code géré* pour le distinguer du code exécuté en dehors du CLR (généralement le code natif), appelé *code non géré* . Divers mécanismes facilitent l'interopérabilité entre le code géré et le code non géré.

Lire CLR en ligne: https://riptutorial.com/fr/dot-net/topic/3942/clr

Chapitre 13: Collecte des ordures

Introduction

Dans .Net, les objets créés avec new () sont alloués sur le tas géré. Ces objets ne sont jamais explicitement finalisés par le programme qui les utilise; à la place, ce processus est contrôlé par le ramasse-miettes .Net.

Certains des exemples ci-dessous sont des "cas pratiques" pour montrer le Garbage Collector au travail et des détails importants sur son comportement, tandis que d'autres se concentrent sur la manière de préparer les classes pour une manipulation correcte par le Garbage Collector.

Remarques

Le Garbage Collector vise à réduire le coût du programme en termes de mémoire allouée, mais cela a un coût en termes de temps de traitement. Afin de parvenir à un bon compromis global, un certain nombre d'optimisations doivent être prises en compte lors de la programmation en gardant à l'esprit le Garbage Collector:

- Si la méthode Collect () doit être explicitement appelée (ce qui ne devrait pas être le cas de toute façon), envisagez d'utiliser le mode "optimisé" qui finalise l'objet mort uniquement lorsque la mémoire est réellement nécessaire
- Au lieu d'appeler la méthode Collect (), envisagez d'utiliser les méthodes AddMemoryPressure () et RemoveMemoryPressure (), qui déclenchent une collecte de mémoire uniquement si nécessaire
- Une collection de mémoire n'est pas garantie pour finaliser tous les objets morts; au lieu de cela, le garbage collector gère 3 "générations", un objet parfois "survivant" d'une génération à l'autre
- Plusieurs modèles de threads peuvent s'appliquer, en fonction de divers facteurs, y compris le réglage fin de la configuration, entraînant différents degrés d'interférence entre le thread du garbage collector et les autres threads d'application.

Examples

Un exemple basique de collection (garbage)

Compte tenu de la classe suivante:

```
public class FinalizableObject
{
    public FinalizableObject()
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
```

```
    {
        Console.WriteLine("Instance finalized");
    }
}
```

Un programme qui crée une instance, même sans l'utiliser:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Produit la sortie suivante:

```
<namespace>.FinalizableObject initialized
```

Si rien d'autre ne se produit, l'objet n'est pas finalisé jusqu'à ce que le programme se termine (ce qui libère tous les objets sur le tas géré, finalisant ceux-ci dans le processus).

Il est possible de forcer le récupérateur de place à s'exécuter à un point donné, comme suit:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Qui produit le résultat suivant:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

Cette fois, dès que le récupérateur de place a été appelé, l'objet inutilisé (aka "dead") a été finalisé et libéré du tas géré.

Objets vivants et objets morts - les bases

Règle de base: lorsque la récupération de place se produit, les «objets actifs» sont ceux qui sont encore utilisés, tandis que les «objets morts» sont ceux qui ne sont plus utilisés (toute variable ou champ les référençant) .

Dans l'exemple suivant (par commodité, FinalizableObject1 et FinalizableObject2 sont des sous-classes de FinalizableObject de l'exemple ci-dessus et héritent donc du comportement du message d'initialisation / finalisation):

```
var obj1 = new FinalizableObject1(); // Finalizable1 instance allocated here
var obj2 = new FinalizableObject2(); // Finalizable2 instance allocated here
obj1 = null; // No more references to the Finalizable1 instance
GC.Collect();
```

Le résultat sera:

```
<namespace>.FinalizableObject1 initialized
<namespace>.FinalizableObject2 initialized
<namespace>.FinalizableObject1 finalized
```

Au moment où le récupérateur de place est appelé, FinalizableObject1 est un objet mort et est finalisé, tandis que FinalizableObject2 est un objet actif et qu'il est conservé sur le segment de mémoire géré.

Plusieurs objets morts

Que se passe-t-il si deux (ou plusieurs) objets morts se réfèrent l'un à l'autre? Ceci est illustré dans l'exemple ci-dessous, en supposant que OtherObject est une propriété publique de FinalizableObject:

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // Program no longer references Finalizable1 instance
obj2 = null; // Program no longer references Finalizable2 instance
// But the two objects still reference each other
GC.Collect();
```

Cela produit la sortie suivante:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

Les deux objets sont finalisés et libérés du tas géré même s'ils se référencent l'un l'autre (car aucune autre référence n'existe à aucun d'entre eux à partir d'un objet réel).

Références faibles

Les références faibles sont ... des références, à d'autres objets (alias "cibles"), mais "faibles" car elles n'empêchent pas ces objets d'être collectés. En d'autres termes, les références faibles ne comptent pas lorsque le récupérateur de place évalue les objets en tant que "live" ou "dead".

Le code suivant:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Produit la sortie:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

L'objet est libéré du tas géré bien qu'il soit référencé par la variable WeakReference (toujours dans l'étendue lorsque le récupérateur de place a été appelé).

Conséquence n ° 1: à tout moment, il est dangereux de supposer qu'une cible WeakReference est toujours allouée ou non sur le segment de mémoire géré.

Conséquence n ° 2: chaque fois qu'un programme doit accéder à la cible d'une référence faible, le code doit être fourni pour les deux cas, la cible étant toujours allouée ou non. La méthode pour accéder à la cible est TryGetTarget:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference<object>(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if(weak.TryGetTarget(out target))
{
    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

La version générique de WeakReference est disponible depuis .Net 4.5. Toutes les versions du framework fournissent une version non générique, non typée, construite de la même manière et vérifiée comme suit:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if (weak.IsAlive)
{
    target = weak.Target;

    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

Dispose () vs. finalizers

Implémentez la méthode Dispose () (et déclarez la classe contenant comme IDisposable) pour vous assurer que toutes les ressources dont la mémoire est lourde sont libérées dès que l'objet n'est plus utilisé. Le "catch" est qu'il n'ya pas de garantie forte que la méthode Dispose () soit invoquée (contrairement aux finaliseurs qui sont toujours appelés à la fin de la vie de l'objet).

Un scénario est un programme appelant Dispose () sur les objets qu'il crée explicitement:

```
private void SomeFunction()
{
    // Initialize an object that uses heavy external resources
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... Use that object

    // Dispose as soon as no longer used
    disposableObject.Dispose();

    // ... Do other stuff

    // The disposableObject variable gets out of scope here
    // The object will be finalized later on (no guarantee when)
    // But it no longer holds to the heavy external resource after it was disposed
}
```

Un autre scénario consiste à déclarer une classe instanciée par le framework. Dans ce cas, la nouvelle classe hérite généralement d'une classe de base. Par exemple, dans MVC, une classe de contrôleur est créée en tant que sous-classe de System.Web.Mvc.ControllerBase. Lorsque la classe de base implémente l'interface IDisposable, il s'agit d'une bonne indication que Dispose () serait correctement invoqué par le framework, mais là encore, il n'y a pas de garantie forte.

Ainsi, Dispose () ne remplace pas un finaliseur; au lieu de cela, les deux devraient être utilisés à des fins différentes:

- Un finaliseur libère éventuellement des ressources pour éviter les fuites de mémoire qui se produiraient autrement
- Dispose () libère les ressources (éventuellement les mêmes) dès que celles-ci ne sont plus nécessaires, pour alléger la pression sur l'allocation globale de la mémoire.

Élimination correcte et finalisation des objets

Comme Dispose () et les finaliseurs visent des objectifs différents, une classe gérant des ressources externes lourdes de mémoire doit les implémenter. La conséquence est d’écrire la classe pour qu’elle gère bien deux scénarios possibles:

- Lorsque seul le finaliseur est appelé
- Lorsque Dispose () est invoqué en premier et plus tard, le finaliseur est également appelé

Une solution consiste à écrire le code de nettoyage de telle sorte que son exécution une ou deux fois produirait le même résultat qu’une exécution unique. La faisabilité dépend de la nature du nettoyage, par exemple:

- Fermer une connexion à une base de données déjà fermée n'aurait probablement aucun effet, donc cela fonctionne
- La mise à jour de certains «comptes d'utilisation» est dangereuse et produirait un résultat erroné lorsqu'elle est appelée deux fois au lieu d'une fois.

Une solution plus sûre consiste à s'assurer, par conception, que le code de nettoyage est appelé une seule fois, quel que soit le contexte externe. Cela peut être réalisé de manière classique en

utilisant un drapeau dédié:

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
    {
        if(!disposed)
        {
            // Actual code to release resources gets here, then
            disposed = true;
        }
    }
}
```

Alternativement, Garbage Collector fournit une méthode spécifique SuppressFinalize () qui permet d'ignorer le finaliseur après l'appel de Dispose:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

Lire Collecte des ordures en ligne: https://riptutorial.com/fr/dot-net/topic/9636/collecte-des-ordures

Chapitre 14: Collections

Remarques

Il existe plusieurs types de collection:

- `Array`
- `List`
- `Queue`
- `SortedList`
- `Stack`
- dictionnaire

Examples

Création d'une liste initialisée avec des types personnalisés

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Nous avons ici une classe sans constructeur avec deux propriétés: `Name` et une propriété booléenne nullable `Selected` . Si nous voulions initialiser une `List<Model>` , il y a plusieurs manières d'exécuter ceci.

```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

Ici, nous créons plusieurs `new` instances de notre classe `Model` et les initialisons avec des données. Et si on ajoutait un constructeur?

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
        Name = name;
        selected = Selected;
    }
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Cela nous permet d'initialiser notre liste un *peu* différemment.

```
var SelectedEmployees = new List<Model>
{
    new Model("Mark", true),
    new Model("Alexis"),
    new Model("")
};
```

Qu'en est-il d'une classe dont l'une des propriétés est une classe elle-même?

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}

public class ExtendedModel : Model
{
    public ExtendedModel()
    {
        BaseModel = new Model();
    }

    public Model BaseModel { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Notez que nous avons annulé le constructeur de la classe `Model` pour simplifier un peu l'exemple.

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Mark", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
    },
                    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Random"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

Notez que nous pouvons échanger notre `List<ExtendedModel>` avec `Collection<ExtendedModel>` , `ExtendedModel[]` , `object[]` , ou même simplement `[]` .

Queue

Il existe une collection dans .Net utilisée pour gérer les valeurs dans une `Queue` utilisant le concept FIFO (premier entré, premier sorti) . Les bases des files d'attente sont la méthode `Enqueue(T item)` qui est utilisée pour ajouter des éléments dans la file d'attente et `Dequeue()` qui est utilisé pour obtenir le premier élément et le supprimer de la file d'attente. La version générique peut être utilisée comme le code suivant pour une file d'attente de chaînes.

Tout d'abord, ajoutez l'espace de noms:

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

et l'utiliser:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("John");
queue.Enqueue("Paul");
queue.Enqueue("George");
queue.Enqueue("Ringo");

string dequeueValue;
dequeueValue = queue.Dequeue(); // return John
dequeueValue = queue.Dequeue(); // return Paul
dequeueValue = queue.Dequeue(); // return George
dequeueValue = queue.Dequeue(); // return Ringo
```

Il existe une version non générique du type, qui fonctionne avec des objets.

L'espace de nom est:

```
using System.Collections;
```

Adn un exemple de code pour une file d'attente non générique:

```
Queue queue = new Queue();
queue.Enqueue("Hello World"); // string
queue.Enqueue(5); // int
queue.Enqueue(1d); // double
queue.Enqueue(true); // bool
queue.Enqueue(new Product()); // Product object

object dequeueValue;
dequeueValue = queue.Dequeue(); // return Hello World (string)
dequeueValue = queue.Dequeue(); // return 5 (int)
dequeueValue = queue.Dequeue(); // return 1d (double)
dequeueValue = queue.Dequeue(); // return true (bool)
dequeueValue = queue.Dequeue(); // return Product (Product type)
```

Il y a aussi une méthode appelée Peek () qui renvoie l'objet au début de la file sans le retirer des éléments.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

foreach (int element in queue)
{
    Console.WriteLine(i);
}
```

La sortie (sans suppression):

```
10
20
30
40
50
```

Empiler

Il existe une collection dans .Net utilisée pour gérer les valeurs dans une `Stack` qui utilise le concept LIFO (last-in first-out) . Les bases des piles sont la méthode `Push(T item)` qui est utilisée pour ajouter des éléments dans la pile et `Pop()` qui est utilisé pour obtenir le dernier élément ajouté et le supprimer de la pile. La version générique peut être utilisée comme le code suivant pour une file d'attente de chaînes.

Tout d'abord, ajoutez l'espace de noms:

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

et l'utiliser:

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // return Ringo
value = stack.Pop(); // return George
value = stack.Pop(); // return Paul
value = stack.Pop(); // return John
```

Il existe une version non générique du type, qui fonctionne avec des objets.

L'espace de nom est:

```
using System.Collections;
```

Et un exemple de code de pile non générique:

```
Stack stack = new Stack();
stack.Push("Hello World"); // string
stack.Push(5); // int
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product object

object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
```

```
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

Il existe également une méthode appelée Peek () qui renvoie le dernier élément ajouté, mais sans le retirer de la `Stack` .

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

Il est possible d'itérer sur les éléments de la pile et il respectera l'ordre de la pile (LIFO).

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

foreach (int element in stack)
{
   Console.WriteLine(element);
}
```

La sortie (sans suppression):

```
50
40
30
20
10
```

Utilisation des initialiseurs de collection

Certains types de collection peuvent être initialisés au moment de la déclaration. Par exemple, l'instruction suivante crée et initialise les `numbers` avec des nombres entiers:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

En interne, le compilateur C # convertit en réalité cette initialisation en une série d'appels à la méthode Add. Par conséquent, vous ne pouvez utiliser cette syntaxe que pour les collections qui prennent en charge la méthode `Add` .

> Les classes `Stack<T>` et `Queue<T>` ne le prennent pas en charge.

Pour les collections complexes telles que la classe `Dictionary<TKey, TValue>` , qui utilisent des paires clé / valeur, vous pouvez spécifier chaque paire clé / valeur en tant que type anonyme dans la liste d'initialisation.

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

Le premier élément de chaque paire est la clé et le second la valeur.

Lire Collections en ligne: https://riptutorial.com/fr/dot-net/topic/30/collections

Chapitre 15: Compilateur JIT

Introduction

La compilation JIT, ou compilation juste-à-temps, est une approche alternative à l'interprétation du code ou à la compilation préalable. La compilation JIT est utilisée dans le framework .NET. Le code CLR (C #, F #, Visual Basic, etc.) est d'abord compilé en quelque chose appelé Langage interprété, ou IL. C'est un code de niveau inférieur, plus proche du code machine, mais qui n'est pas spécifique à la plate-forme. Au moment de l'exécution, ce code est plutôt compilé en code machine pour le système concerné.

Remarques

Pourquoi utiliser la compilation JIT?

- Meilleure compatibilité: chaque langage CLR n'a besoin que d'un seul compilateur pour IL, et cet IL peut s'exécuter sur toute plate-forme sur laquelle il peut être converti en code machine.
- Vitesse: la compilation JIT tente de combiner la vitesse d'exécution du code compilé en avance et la souplesse de l'interprétation (peut analyser le code qui sera exécuté pour les optimisations potentielles avant la compilation)

Page Wikipedia pour plus d'informations sur la compilation JIT en général:
https://en.wikipedia.org/wiki/Just-in-time_compilation

Examples

Échantillon de compilation IL

Simple Hello World Application:

```
using System;

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

Code IL équivalent (qui sera compilé par JIT)

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
```

```
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 4:0:0:0
}
.assembly HelloWorld
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01
00 08 00 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01
00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

  // --- The following custom attribute is added automatically, do not uncomment -------
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )


  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = (
01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void
[mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) =
( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) =
( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string)
= ( 01 00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20   // ...Copyright ..

20 32 30 31 37 00 00 )                            //  2017..
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string)
= ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string)
= ( 01 00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32   // ..$308b3d86-4172

2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33   // -4022-afcc-3f8e3

32 33 33 63 35 62 30 00 00 )                      // 233c5b0..
  .custom instance void
[mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = ( 01 00 07 31 2E 30
2E 30 2E 30 00 00 )             // ...1.0.0.0..
  .custom instance void
[mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string) = ( 01 00 1C 2E 4E
45 54 46 72 61 6D 65 77 6F 72 6B   // ....NETFramework

2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01   // ,Version=v4.5.2.
```

```
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73   // .T..FrameworkDis

70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72   // playName..NET Fr

61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )          // amework 4.5.2
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method Program::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       8 (0x8)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  ret
  } // end of method Program::.ctor

} // end of class HelloWorld.Program
```

Généré avec l'outil MS ILDASM (désassembleur IL)

Lire Compilateur JIT en ligne: https://riptutorial.com/fr/dot-net/topic/9222/compilateur-jit

Chapitre 16: Contextes de synchronisation

Remarques

Un contexte de synchronisation est une abstraction qui permet de coder pour transmettre des unités de travail à un planificateur, sans qu'il soit nécessaire de savoir comment le travail sera planifié.

Les contextes de synchronisation sont traditionnellement utilisés pour garantir l'exécution du code sur un thread spécifique. Dans les applications WPF et Winforms, un cadre `SynchronizationContext` représentant le thread d'interface utilisateur est fourni par le framework de présentation. De cette manière, `SynchronizationContext` peut être considéré comme un modèle producteur-consommateur pour les délégués. Un thread de travail *produira du* code exécutable (le délégué) et le mettra en file d'attente ou le *consommera* par la boucle de message de l'interface utilisateur.

La bibliothèque parallèle de tâches fournit des fonctionnalités permettant de capturer et d'utiliser automatiquement des contextes de synchronisation.

Examples

Exécuter du code sur le thread d'interface utilisateur après avoir effectué un travail en arrière-plan

Cet exemple montre comment mettre à jour un composant d'interface utilisateur à partir d'un thread d'arrière-plan en utilisant un `SynchronizationContext`

```
void Button_Click(object sender, EventArgs args)
{
    SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
            Thread.Sleep(500); //simulate work being done
            context.Post(ShowProgress, "Work complete on item " + i);
        }
    }
}

void UpdateCallback(object state)
{
    // UI can be safely updated as this method is only called from the UI thread
    this.MyTextBox.Text = state as string;
}
```

Dans cet exemple, si vous `MyTextBox.Text` mettre à jour directement `MyTextBox.Text` dans la boucle `for` , vous obtiendriez une erreur de threading. En publiant l'action `UpdateCallback` sur le `SynchronizationContext` , la zone de texte est mise à jour sur le même thread que le reste de l'interface utilisateur.

En pratique, les mises à jour de progression doivent être effectuées à l'aide d'une instance de `System.IProgress<T>` . L'implémentation par défaut `System.Progress<T>` capture automatiquement le contexte de synchronisation sur `System.Progress<T>` elle est créée.

Lire Contextes de synchronisation en ligne: https://riptutorial.com/fr/dot-net/topic/5407/contextes-de-synchronisation

Chapitre 17: Contrats de code

Remarques

Les contrats de code permettent de compiler ou d'exécuter des conditions pré / post des méthodes et des conditions invariantes pour les objets. Ces conditions peuvent être utilisées pour garantir que les appelants et la valeur de retour correspondent à des états valides pour le traitement de l'application. Les autres utilisations des contrats de code incluent la génération de documentation.

Examples

Conditions préalables

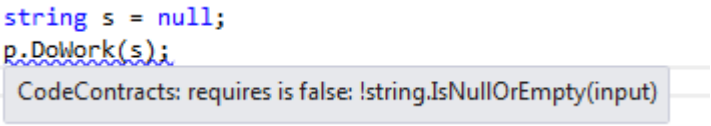
Les conditions préalables permettent aux méthodes de fournir des valeurs minimales requises pour les paramètres d'entrée

Exemple...

```
void DoWork(string input)
{
    Contract.Requires(!string.IsNullOrEmpty(input));

    //do work
}
```

Résultat d'analyse statique ...



Postconditions

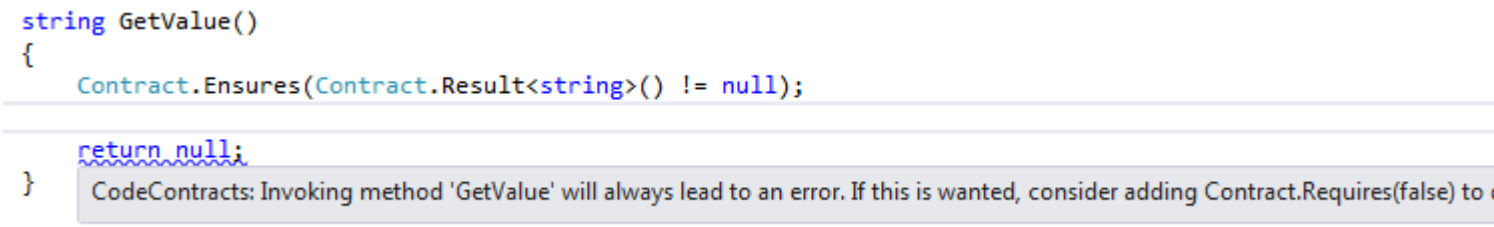
Les postconditions garantissent que les résultats renvoyés par une méthode correspondent à la définition fournie. Cela fournit à l'appelant une définition du résultat attendu. Les postconditions peuvent permettre des implémentations simplifiées, car certains résultats possibles peuvent être fournis par l'analyseur statique.

Exemple...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

Résultat d'analyse statique ...



Contrats pour interfaces

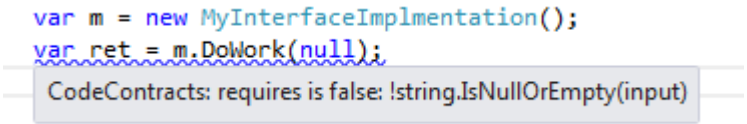
En utilisant des contrats de code, il est possible d'appliquer un contrat à une interface. Cela se fait en déclarant une classe abstraite qui implémente les interfaces. L'interface doit être étiquetée avec le `ContractClassAttribute` et la définition du contrat (la classe abstraite) doit être balisée avec `ContractClassForAttribute`

C # Exemple ...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//Never inherit from this contract defintion class
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
        Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

Résultat d'analyse statique ...

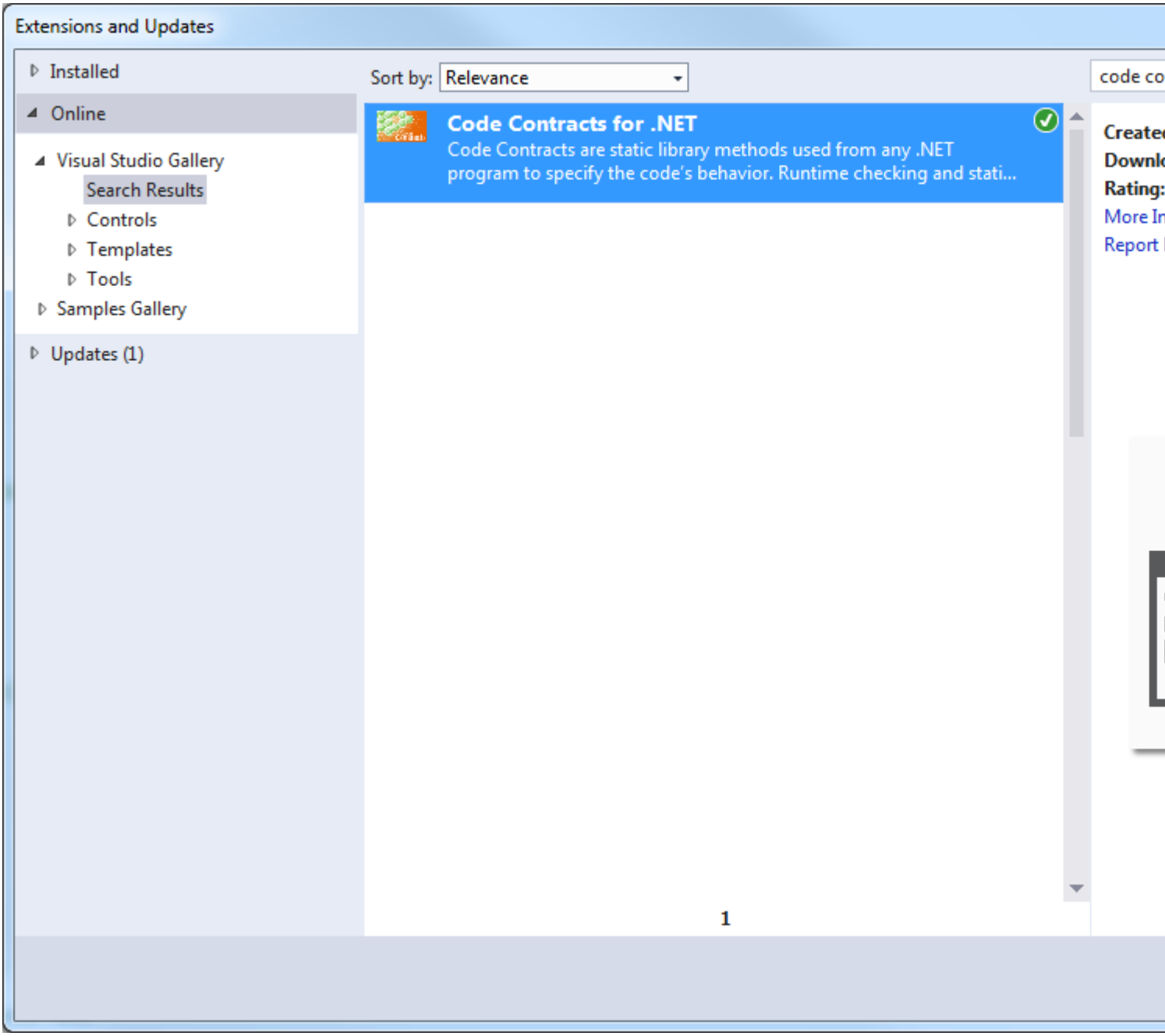


Installation et activation de contrats de code

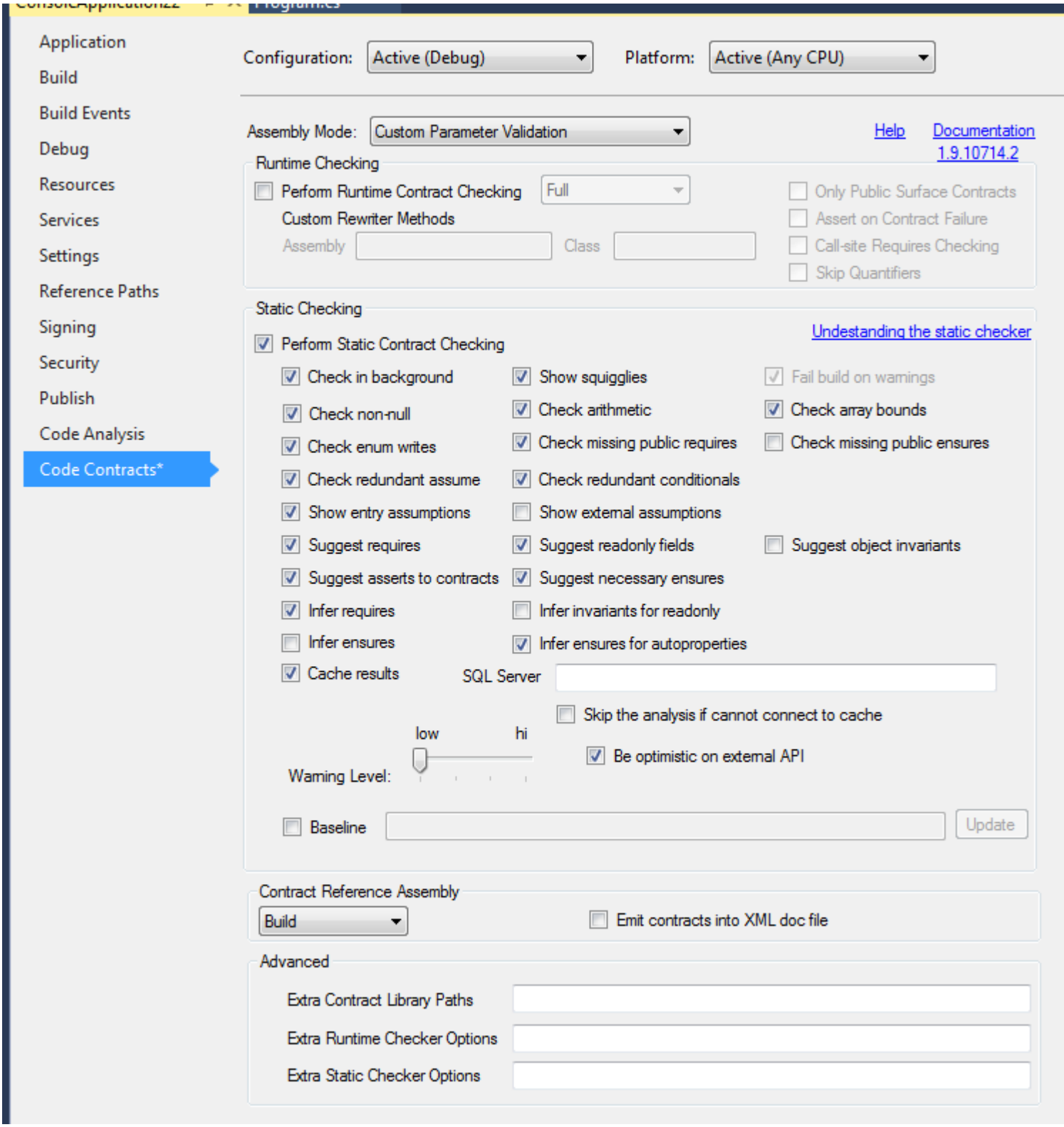
Alors que `System.Diagnostics.Contracts` est inclus dans le .Net Framework. Pour utiliser des

contrats de code, vous devez installer les extensions Visual Studio.

Sous `Extensions and Updates` recherchez les `Code Contracts` puis installez les `Code Contracts Tools`



Une fois les outils installés, vous devez activer les `Code Contracts` dans votre solution Project. Au minimum, vous voulez probablement activer la `Static Checking` (vérifier après la construction). Si vous implémentez une bibliothèque qui sera utilisée par d'autres solutions, vous pouvez également envisager d'activer la `Runtime Checking` .



Lire Contrats de code en ligne: https://riptutorial.com/fr/dot-net/topic/1937/contrats-de-code

Chapitre 18: Cordes

Remarques

Dans les chaînes .NET, `System.String` est une séquence de caractères `System.Char` , chaque caractère est une unité de code codée UTF-16. Cette distinction est importante car la définition du *caractère dans la langue parlée et la* définition du *caractère* .NET (et de nombreuses autres langues) sont différentes.

Un *caractère* , qui devrait être correctement appelé grapheme , est affiché comme un glyphe et est défini par un ou plusieurs points de code Unicode. Chaque point de code est ensuite codé dans une séquence d' unités de code . Maintenant, il devrait être clair pourquoi un seul `System.Char` ne représente pas toujours un graphème, voyons dans la réalité comment ils sont différents:

- Un graphème, en raison de la combinaison de caractères , peut donner deux ou plusieurs points de code: `à` est composé de deux points de code: *U + 0061 LETTRE MINUSCULE LATINE A* et *U + 0300 COMBINING ACCENT ACCENT* . C'est l'erreur la plus courante car `"à ".Length == 2` alors que vous pouvez vous attendre à `1` .
- Il y a des caractères dupliqués, par exemple `à` peut être un seul point de code *U + 00E0 LATIN SMALL LETTER A AVEC GRAVE* ou deux points de code comme expliqué ci-dessus. Évidemment, ils doivent comparer le même: `"\u00e0" == "\u0061\u0300"` (même si `"\u00e0".Length != "\u0061\u0300".Length`). Cela est possible en raison de *la normalisation* de *chaîne* effectuée par la méthode `String.Normalize()` .
- Une séquence Unicode peut contenir une séquence composée ou décomposé, par exemple le caractère 힜 *U + D55C HAN caractère* peut être un seul point de code (codé comme un seul code-unité UTF-16) ou une séquence décomposée de ses syllabes 하, 흫 et 힜 . Ils doivent être comparés à égalité.
- Un point de code peut être codé en plusieurs unités de code: caractère 𠮊 *U +* `2008A` *HAN CHARACTER* est codé sous la forme de deux `System.Char` (`"\ud840\udc8a"`) même s'il ne s'agit que d'un seul point de code: UTF-16 l'encodage n'est pas de taille fixe! Ceci est une source d'innombrables bogues (également des bogues de sécurité sérieux), si par exemple votre application applique une longueur maximale et tronque aveuglément la chaîne à ce moment-là, vous pouvez créer une chaîne non valide.
- Certaines langues ont un digraphe et des trigraphes, par exemple en tchèque `ch` est une lettre autonome (après `h` et avant `i`, lorsque vous commandez une liste de chaînes, vous aurez *fyzika* avant *chimie* .

Il y a beaucoup plus de problèmes concernant la gestion du texte, voir par exemple Comment puis-je effectuer une comparaison caractère par caractère compatible avec Unicode? pour une introduction plus large et plus de liens vers des arguments connexes.

En général, lorsque vous traitez *du* texte *international,* vous pouvez utiliser cette fonction simple pour énumérer des éléments de texte dans une chaîne (en évitant de casser les substituts et l'encodage Unicode):

```
public static class StringExtensions
{
    public static IEnumerable<string> EnumerateCharacters(this string s)
    {
        if (s == null)
            return Enumerable.Empty<string>();

        var enumerator = StringInfo.GetTextElementEnumerator(s.Normalize());
        while (enumerator.MoveNext())
            yield return (string)enumerator.Value;
    }
}
```

Examples

Compter des caractères distincts

Si vous avez besoin de compter des caractères distincts, pour les raisons expliquées dans la section *Notes* , vous ne pouvez pas simplement utiliser la propriété `Length` parce que c'est la longueur du tableau de `System.Char` qui ne sont pas des caractères mais des unités de code ni graphèmes). Si, par exemple, vous écrivez simplement du `text.Distinct().Count()` vous obtiendrez des résultats incorrects, corrigez le code:

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

Une étape supplémentaire consiste à **compter les occurrences de chaque caractère** , si les performances ne sont pas un problème, vous pouvez simplement le faire comme ça (dans cet exemple indépendamment du cas):

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

Compter les caractères

Si vous avez besoin de compter des *caractères* , pour les raisons expliquées dans la section *Notes* , vous ne pouvez pas simplement utiliser la propriété Length parce que c'est la longueur du tableau de `System.Char` qui ne sont pas des caractères mais des unités de code graphèmes). Le code correct est alors:

```
int length = text.EnumerateCharacters().Count();
```

Une petite optimisation peut réécrire spécifiquement la méthode d'extension `EnumerateCharacters()` à cette fin:

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
```

```
            return 0;

        int count = 0;
        var enumerator = StringInfo.GetTextElementEnumerator(text);
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

Compter les occurrences d'un caractère

En raison des raisons expliquées dans la section *Remarques* , vous ne pouvez pas simplement faire cela (sauf si vous voulez compter les occurrences d'une unité de code spécifique):

```
int count = text.Count(x => x == ch);
```

Vous avez besoin d'une fonction plus complexe:

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

Notez que la comparaison de chaînes (contrairement à la comparaison de caractères qui est invariante par culture) doit toujours être effectuée conformément aux règles d'une culture spécifique.

Fendre la chaîne en blocs de longueur fixe

Nous ne pouvons pas briser une chaîne en points arbitraires (parce qu'un `System.Char` peut ne pas être valide seul parce qu'il est un personnage combinant ou partie d'un substitut) puis le code doit tenir compte (notez que avec la *longueur* que je veux dire le nombre de *graphèmes* pas nombre d' *unités* de *code*):

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
```

```
}
```

Convertir une chaîne vers / depuis un autre encodage

Les chaînes .NET contiennent `System.Char` (unités de code UTF-16). Si vous souhaitez enregistrer (ou gérer) du texte avec un autre encodage, vous devez utiliser un tableau de `System.Byte` .

Les conversions sont effectuées par des classes dérivées de `System.Text.Encoder` et `System.Text.Decoder` qui, ensemble, peuvent convertir vers / depuis un autre codage (à partir d' une matrice codée octet *X* `byte[]` à une UTF-16 codé `System.String` et vice -versa).

Comme l'encodeur / décodeur fonctionne généralement très près les uns des autres, ils sont regroupés dans une classe dérivée de `System.Text.Encoding` , les classes dérivées offrent des conversions vers / depuis les encodages courants (UTF-8, UTF-16, etc.).

Exemples:

Convertir une chaîne en UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("This is my text");
```

Convertir des données UTF-8 en chaîne

```
var text = Encoding.UTF8.GetString(data);
```

Modifier l'encodage d'un fichier texte existant

Ce code lit le contenu d'un fichier texte encodé en UTF-8 et le sauvegarde sous forme de code UTF-16. Notez que ce code n'est pas optimal si le fichier est gros car il lira tout son contenu en mémoire:

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

Méthode virtuelle Object.ToString ()

Tout dans .NET est un objet, donc chaque type a la méthode `ToString()` définie dans la classe `Object` qui peut être remplacée. L'implémentation par défaut de cette méthode renvoie simplement le nom du type:

```
public class Foo
{
```

```
}

var foo = new Foo();
Console.WriteLine(foo); // outputs Foo
```

`ToString()` est implicitement appelé lors de la concaténation de la valeur avec une chaîne:

```
public class Foo
{
    public override string ToString()
    {
        return "I am Foo";
    }
}

var foo = new Foo();
Console.WriteLine("I am bar and "+foo);// outputs I am bar and I am Foo
```

Le résultat de cette méthode est également largement utilisé par les outils de débogage. Si, pour une raison quelconque, vous ne souhaitez pas remplacer cette méthode, mais souhaitez personnaliser la manière dont le débogueur affiche la valeur de votre type, utilisez l' attribut DebuggerDisplay (MSDN):

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = FN {"+nameof(Person.FirstName)+"}, LN
{"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```

Immuabilité des cordes

Les chaînes sont immuables. Vous ne pouvez pas changer la chaîne existante. Toute opération sur la chaîne crée une nouvelle instance de la chaîne ayant une nouvelle valeur. Cela signifie que si vous devez remplacer un seul caractère dans une chaîne très longue, la mémoire sera allouée pour une nouvelle valeur.

```
string veryLongString = ...
// memory is allocated
string newString = veryLongString.Remove(0,1); // removes first character of the string.
```

Si vous devez effectuer de nombreuses opérations avec une valeur de chaîne, utilisez la classe `StringBuilder` conçue pour une manipulation efficace des chaînes:

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
    sb.Append(str);
}
var finalString = sb.ToString();
```

Comparer les chaînes

Malgré que `String` soit un type de référence, l'opérateur `==` compare les valeurs de chaîne plutôt que les références.

Comme vous le savez peut-être, `string` n'est qu'un tableau de caractères. Mais si vous pensez que le contrôle et la comparaison de l'égalité des cordes sont rendus caractère par caractère, vous vous trompez. Cette opération est spécifique à la culture (voir Remarques ci-dessous): certaines séquences de caractères peuvent être considérées comme égales en fonction de la culture .

Réfléchissez à deux fois avant de court-circuiter le contrôle de l'égalité en comparant les propriétés de `Length` de deux chaînes!

Utilisez des surcharges de la méthode `String.Equals` qui acceptent une valeur d' énumération `StringComparison` supplémentaire, si vous devez modifier le comportement par défaut.

Lire Cordes en ligne: https://riptutorial.com/fr/dot-net/topic/2227/cordes

Chapitre 19: Des exceptions

Remarques

En relation:

- MSDN: Exceptions et gestion des exceptions (Guide de programmation C #)
- MSDN: gestion et levée des exceptions
- MSDN: CA1031: Ne pas intercepter les types d'exceptions générales
- MSDN: try-catch (Référence C #)

Examples

Prendre une exception

Le code peut et doit prévoir des exceptions dans des circonstances exceptionnelles. Des exemples de ceci incluent:

- Tenter de lire après la fin d'un flux
- Ne pas avoir les autorisations nécessaires pour accéder à un fichier
- Tenter d'effectuer une opération non valide, par exemple en divisant par zéro
- Un timeout se produisant lors du téléchargement d'un fichier à partir d'Internet

L'appelant peut gérer ces exceptions en les "attrapant" et ne devrait le faire que lorsque:

- Il peut en fait résoudre le cas exceptionnel ou se rétablir correctement;
- Il peut fournir un contexte supplémentaire à l'exception qui serait utile si l'exception doit être renvoyée (les exceptions renvoyées sont interceptées par les gestionnaires d'exceptions en amont de la pile d'appels)

Il convient de noter que choisir de *ne* pas attraper une exception est parfaitement valable si l'intention est de le traiter à un niveau supérieur.

La capture d'une exception se fait en encapsulant le code potentiellement lançant dans un bloc `try { ... }` comme suit et en capturant les exceptions qu'il peut gérer dans un bloc `catch (ExceptionType) { ... }` :

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
```

```
    Console.WriteLine("File '{0}' could not be found.", filename);
}
```

Utiliser un bloc

Le bloc `finally { ... }` d'un `try-finally` ou de `try-catch-finally` s'exécutera toujours, qu'une exception soit survenue ou non (sauf lorsqu'une exception `StackOverflowException` a été lancée ou qu'un appel a été effectué à `Environment.FailFast()`).

Il peut être utilisé pour libérer ou nettoyer les ressources acquises dans le bloc `try { ... }` toute sécurité.

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
finally
{
    if (fileStream != null)
    {
        fileStream.Dispose();
    }
}
```

Attraper et retracer les exceptions prises

Lorsque vous souhaitez intercepter une exception et faire quelque chose, mais que vous ne pouvez pas continuer à exécuter le bloc de code actuel en raison de l'exception, vous pouvez renvoyer l'exception au gestionnaire d'exceptions suivant dans la pile d'appels. Il y a de bons et de mauvais moyens pour y parvenir.

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // will throw a DivideByZeroException

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
        // we do not need a reference to the exception
```

```
        Console.WriteLine("Dividing by zero would destroy the universe.");

        // do this to preserve the stack trace:
        throw;
    }
    catch (FormatException ex)
    {
        // only do this if you need to change the type of the Exception to be thrown
        // and wrap the inner Exception

        // remember that the stack trace of the outer Exception will point to the
        // next line

        // you'll need to examine the InnerException property to get the stack trace
        // to the line that actually started the problem

        throw new InvalidOperationException("Watch your format string indexes.", ex);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Something else horrible happened. The exception: " + ex.Message);

        // do not do this, because the stack trace will be changed to point to
        // this location instead of the location where the exception
        // was originally thrown:
        throw ex;
    }
}

static void Main()
{
    try
    {
        AskTheUltimateQuestion();
    }
    catch
    {
        // choose this kind of catch if you don't need any information about
        // the exception that was caught

        // this block "eats" all exceptions instead of rethrowing them
    }
}
```

Vous pouvez filtrer par type d'exception et même par propriétés d'exception (nouveau dans C # 6.0, un peu plus long disponible dans VB.NET (citation requise)):

Documentation / C # / nouvelles fonctionnalités

Filtres d'exception

Comme les exceptions C # 6.0 peuvent être filtrées à l'aide de l'opérateur `when` .

Ceci est similaire à l' aide d' un simple `if` mais ne se déroule pas la pile si la condition à l' intérieur du `when` est pas remplie.

Exemple

```
try
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // Any condition can go in here.
{
  // ...
}
```

La même information peut être trouvée dans les fonctionnalités de C # 6.0 ici: Filtres d'exception

Renvoyer une exception dans un bloc catch

Dans un bloc `catch` , le mot-clé `throw` peut être utilisé seul, sans spécifier de valeur d'exception, pour *renvoyer* l'exception qui vient d'être interceptée. La retransmission d'une exception permet à l'exception d'origine de poursuivre la chaîne de gestion des exceptions, en préservant sa pile d'appels ou les données associées:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is *not* used
  throw;
}
```

Un anti-pattern commun consiste à `throw ex` à la place `throw ex` , ce qui a pour effet de limiter la vue du gestionnaire d'exceptions suivant de la trace de la pile:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is thrown
  //  future stack traces of the exception will not see prior calls
  throw ex;
}
```

En général, l'utilisation de `throw ex` n'est pas souhaitable, car les futurs gestionnaires d'exceptions qui inspectent la trace de la pile ne pourront voir les appels que dans les `throw ex` lancés. En omettant la variable `ex` et en utilisant le mot-clé `throw` seul, l'exception d'origine " va " .

Lancer une exception à partir d'une méthode différente tout en préservant ses informations

Parfois, vous souhaitez intercepter une exception et la lancer à partir d'un thread ou d'une méthode différente tout en préservant la pile d'exception d'origine. Cela peut être fait avec `ExceptionDispatchInfo` :

```
using System.Runtime.ExceptionServices;

void Main()
{
    ExceptionDispatchInfo capturedException = null;
    try
    {
```

```
        throw new Exception();
    }
    catch (Exception ex)
    {
        capturedException = ExceptionDispatchInfo.Capture(ex);
    }

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

    if (capturedException != null)
    {
        // Exception stack trace will show it was thrown from Main() and not from Foo()
        exceptionDispatchInfo.Throw();
    }
}
```

Lire Des exceptions en ligne: https://riptutorial.com/fr/dot-net/topic/33/des-exceptions

Chapitre 20: Diagnostique du systeme

Examples

Chronomètre

Cet exemple montre comment `Stopwatch` peut être utilisé pour évaluer un bloc de code.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
        sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
        sw.Stop();
        Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Exécuter des commandes de shell

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

C'est pour cacher la fenêtre cmd.

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

Envoyer la commande au CMD et recevoir une sortie

Cette méthode permet d'envoyer une `command` à `Cmd.exe` et renvoie la sortie standard (y compris l'erreur standard) sous forme de chaîne:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32", // Directory to make the call from
        WindowStyle = ProcessWindowStyle.Hidden,   // Hide the window
        UseShellExecute = false,                   // Do not use the OS shell to start the
process
        CreateNoWindow = true,                     // Start the process in a new window
        RedirectStandardOutput = true,             // This is required to get STDOUT
        RedirectStandardError = true               // This is required to get STDERR
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Usage

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

Sortie

> string currentUsers = "NOM D'UTILISATEUR NOM DE SESSION ID ETAT IDLE HEURE LOGON HEURE Joe.Bloggs ica-cgp N ° 0 Actif 24692 + 13: 29 25/07/2016 07:50 Jim.McFlannegan ica-cgp # 1 3 Actif. 25/07 / 2016 08:33 Andy.McAnderson ica-cgp # 2 4 Actif 25/07/2016 08:54 John.Smith ica-cgp # 4 5 Actif 14 25/07/2016 08:57 Bob.Bobbington ica-cgp # 5 6 Actif 24692 + 13: 29 25/07/2016 09:05 Tim.Tom ica-cgp # 6 7 Actif 25/07/2016 09:08 Bob.Joges ica-cgp # 7 8 Actif 24692 + 13: 29 25 / 07/2016 09:13 "

Dans certains cas, l'accès au serveur en question peut être limité à certains utilisateurs. Si vous avez les identifiants de connexion pour cet utilisateur, il est possible d'envoyer des requêtes avec cette méthode:

```
private static string SendCommand(string command)
```

```
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // This does not actually work in
conjunction with "runas" - the console window will still appear!
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
        Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Obtenir le mot de passe:

```
static SecureString GetPassword()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
        ss.AppendChar(c);
    }

    return ss;
}
```

Remarques

Les deux méthodes ci-dessus `OutputDataReceived` une concaténation de STDOUT et de STDERR, puisque `OutputDataReceived` et `ErrorDataReceived` s'ajoutent tous deux à la même variable - `cmdOut` .

Lire Diagnostique du systeme en ligne: https://riptutorial.com/fr/dot-net/topic/3143/diagnostique-du-systeme

Chapitre 21: Dictionnaires

Examples

Enumérer un dictionnaire

Vous pouvez énumérer un dictionnaire de trois manières différentes:

Utiliser des paires KeyValue

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
   Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

Utiliser les clés

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Utiliser des valeurs

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Initialisation d'un dictionnaire avec un initialiseur de collection

```
// Translates to `dict.Add(1, "First")` etc.
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// Translates to `dict[1] = "First"` etc.
// Works in C# 6.0.
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

Ajouter à un dictionnaire

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// To safely add items (check to ensure item does not already exist - would throw)
if(!dict.ContainsKey(3))
{
   dict.Add(3, "Third");
}
```

Alternativement, ils peuvent être ajoutés / définis via l'indexeur. (Un indexeur en interne ressemble à une propriété, ayant un get et un set, mais prend un paramètre de n'importe quel type spécifié entre les crochets):

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

Contrairement à la méthode `Add` qui génère une exception, si une clé est déjà contenue dans le dictionnaire, l'indexeur remplace simplement la valeur existante.

Pour utiliser un dictionnaire thread-safe, utilisez `ConcurrentDictionary<TKey, TValue>` :

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Obtenir une valeur d'un dictionnaire

Étant donné ce code de configuration:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Vous voudrez peut-être lire la valeur de l'entrée avec la clé 1. Si la clé n'existe pas, l'obtention d'une valeur lancera une `KeyNotFoundException` . Vous pouvez donc commencer par vérifier cela avec `ContainsKey` :

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

Cela présente un inconvénient: vous allez parcourir votre dictionnaire deux fois (une fois pour vérifier l'existence et une autre pour lire la valeur). Pour un dictionnaire volumineux, cela peut avoir un impact sur les performances. Heureusement, les deux opérations peuvent être effectuées

ensemble:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Faire un dictionnaire avec des clés insensibles à la casse.

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

ConcurrentDictionary (à partir de .NET 4.0)

> Représente une collection thread-safe de paires clé / valeur accessibles simultanément par plusieurs threads.

Créer une instance

La création d'une instance fonctionne à peu près comme avec `Dictionary<TKey, TValue>` , par exemple:

```
var dict = new ConcurrentDictionary<int, string>();
```

Ajout ou mise à jour

Vous pourriez être surpris, il n'y a pas de méthode `Add` , mais à la place, il y a `AddOrUpdate` avec 2 surcharges:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Ajoute une paire clé / valeur si la clé n'existe pas déjà ou met à jour une paire clé / valeur en utilisant la fonction spécifiée si la clé existe déjà.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` - *Utilise les fonctions spécifiées pour ajouter une paire clé / valeur au si la clé n'existe pas déjà ou pour mettre à jour une paire clé / valeur si la clé existe déjà.*

Ajouter ou mettre à jour une valeur, quelle que soit la valeur si elle était déjà présente pour une clé donnée (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Ajout ou mise à jour d'une valeur, mais maintenant modification de la valeur dans update, en fonction de la valeur précédente (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld}
Updated");
```

En utilisant la surcharge (2), nous pouvons également ajouter une nouvelle valeur en utilisant une fabrique:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First",
(updateKey, valueOld) => $"{valueOld} Updated");
```

Obtenir de la valeur

Obtenir une valeur est la même que pour le `Dictionary<TKey,TValue>` :

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Obtenir ou ajouter une valeur

Il y a deux surcharges mehod, qui **obtiendront ou ajouteront** une valeur d'une manière thread-safe.

Récupère la valeur avec la clé 2 ou ajoute la valeur "Second" si la clé est absente:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Utiliser une fabrique pour ajouter une valeur, si value is not present:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

IEnumerable to Dictionary (≥ .NET 3.5)

Créez un dictionnaire <TKey, TValue> à partir d'un IEnumerable <T> :

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

```
public class Fruits
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
var fruits = new[]
{
    new Fruits { Id = 8 , Name = "Apple" },
    new Fruits { Id = 3 , Name = "Banana" },
    new Fruits { Id = 7 , Name = "Mango" },
};
```

```
// Dictionary<int, string>                  key      value
var dictionary = fruits.ToDictionary(x => x.Id, x => x.Name);
```

Supprimer d'un dictionnaire

Étant donné ce code de configuration:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Utilisez la méthode `Remove` pour supprimer une clé et sa valeur associée.

```
bool wasRemoved = dict.Remove(2);
```

L'exécution de ce code supprime la clé `2` et sa valeur du dictionnaire. `Remove` renvoie une valeur booléenne indiquant si la clé spécifiée a été trouvée et supprimée du dictionnaire. Si la clé n'existe pas dans le dictionnaire, rien n'est supprimé du dictionnaire et false est renvoyé (aucune exception n'est levée).

Il est **incorrect** d'essayer de supprimer une clé en définissant la valeur de la clé sur `null` .

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

Cela ne supprimera pas la clé. Il remplacera simplement la valeur précédente par une valeur `null` .

Pour supprimer toutes les clés et valeurs d'un dictionnaire, utilisez la méthode `Clear` .

```
dict.Clear();
```

Après l' exécution `Clear` du dictionnaire `Count` sera de 0, mais la capacité interne reste inchangée.

ContainsKey (TKey)

Pour vérifier si un `Dictionary` a une clé spécifique, vous pouvez appeler la méthode `ContainsKey(TKey)` et fournir la clé de type `TKey` . La méthode renvoie une valeur `bool` lorsque la clé existe dans le dictionnaire. Comme échantillon:

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

Et vérifiez si un `C2` existe dans le Dictionnaire:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
}
```

La méthode ContainsKey est disponible sur la version générique `Dictionary<TKey, TValue>` .

Dictionnaire à la liste

Créer une liste de KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Créer une liste de clés:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Créer une liste de valeurs:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

ConcurrentDictionary augmenté avec Lazy'1 réduit le calcul dupliqué

Problème

ConcurrentDictionary brille quand il s'agit de renvoyer instantanément des clés existantes du cache, la plupart du temps sans verrou, et de se disputer à un niveau granulaire. Mais que se passe-t-il si la création de l'objet est vraiment coûteuse, dépassant le coût de la commutation de contexte et que certains incidents de cache se produisent?

Si la même clé est demandée à plusieurs threads, l'un des objets résultant des opérations en collision sera éventuellement ajouté à la collection, et les autres seront jetés, gaspillant la ressource du processeur pour créer l'objet et la ressource mémoire pour stocker temporairement l'objet . D'autres ressources pourraient également être gaspillées. C'est vraiment mauvais.

Solution

Nous pouvons combiner `ConcurrentDictionary<TKey, TValue>` avec `Lazy<TValue>` . L'idée est que la méthode GetOrAdd de ConcurrentDictionary ne peut que renvoyer la valeur réellement ajoutée à la collection. Les objets perdants de Lazy pourraient aussi être perdus dans ce cas, mais cela ne

pose pas de problème, car l'objet Lazy lui-même est relativement peu coûteux. La propriété Value de Lazy Lazy n'est jamais demandée, car nous ne pouvons demander que la propriété Value de celle ajoutée à la collection, celle renvoyée par la méthode GetOrAdd:

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
        TKey key,
        Func<TKey, TValue> factory)
    {
        return
            d.GetOrAdd(
                key,
                key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

La mise en cache des objets XmlSerializer peut s'avérer particulièrement coûteuse, et le démarrage de l'application soulève également de nombreuses questions. Et il y a plus à cela: si ce sont des sérialiseurs personnalisés, il y aura aussi une fuite de mémoire pour le reste du cycle de vie du processus. Le seul avantage du ConcurrentDictionary dans ce cas est que pour le reste du cycle de vie du processus, il n'y aura pas de verrous, mais le démarrage de l'application et l'utilisation de la mémoire seraient inacceptables. Ceci est un travail pour notre ConcurrentDictionary, augmenté avec Lazy:

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```

Lire Dictionnaires en ligne: https://riptutorial.com/fr/dot-net/topic/45/dictionnaires

Chapitre 22: Ecrire et lire le flux StdErr

Examples

Écrire dans une sortie d'erreur standard à l'aide de la console

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
    stdErr.WriteLine($"Failed to copy '{sourceFileName}': {e.Message}");
}
```

Lecture de l'erreur standard du processus enfant

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
    StartInfo = new ProcessStartInfo
    {
        RedirectStandardError = true,
        FileName = "xcopy.exe",
        Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrong
    System.Console.Error.WriteLine($"Child process error: \r\n {errors}");
```

Lire Ecrire et lire le flux StdErr en ligne: https://riptutorial.com/fr/dot-net/topic/10779/ecrire-et-lire-le-flux-stderr

Chapitre 23: Expressions régulières (System.Text.RegularExpressions)

Examples

Vérifiez si le motif correspond à l'entrée

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

Options de passage

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

Match simple et remplacer

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```

Match en groupes

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

    Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
```

```
}
```

Supprimer les caractères non alphanumériques de la chaîne

```
public string Remove()
{
    string input = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

Trouver tous les matchs

En utilisant

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

Code

```
static void Main(string[] args)
{
    string input = "Carrot Banana Apple Cherry Clementine Grape";
    // Find words that start with uppercase 'C'
    string pattern = @"\bC\w*\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
        Console.WriteLine(m.Value);
}
```

Sortie

```
Carrot
Cherry
Clementine
```

Lire Expressions régulières (System.Text.RegularExpressions) en ligne: https://riptutorial.com/fr/dot-net/topic/6944/expressions-regulieres--system-text-regularexpressions-

Chapitre 24: Fichier d'entrée / sortie

Paramètres

Paramètre	Détails
chemin de chaîne	Chemin du fichier à vérifier (parent ou pleinement qualifié)

Remarques

Renvoie true si le fichier existe, false sinon.

Examples

VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\path\to\file.txt"
File.WriteAllText(filename, "Text to write" & vbCrLf)
```

VB StreamWriter

```
Dim filename As String = "c:\path\to\file.txt"
If System.IO.File.Exists(filename) Then
    Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("Text to write" & vbCrLf) 'Add a newline
    writer.close()
End If
```

C # StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\path\to\file.txt";
//'using' structure allows for proper disposal of stream.
using (StreamWriter writer = new StreamWriter(filename"))
{
    writer.WriteLine("Text to Write\n");
}
```

C # WriteAllText ()

```
using System.IO;
using System.Text;
```

```
string filename = "c:\path\to\file.txt";
File.writeAllText(filename, "Text to write\n");
```

C # File.Exists ()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
            Console.WriteLine("Exists");
        }
        else
        {
            Console.WriteLine("Does not exist");
        }
    }
}
```

Peut également être utilisé dans un opérateur ternaire.

```
Console.WriteLine(File.Exists(pathToFile) ? "Exists" : "Does not exist");
```

Lire Fichier d'entrée / sortie en ligne: https://riptutorial.com/fr/dot-net/topic/1376/fichier-d-entree---sortie

Chapitre 25: Filetage

Examples

Accéder aux contrôles de formulaire à partir d'autres threads

Si vous souhaitez modifier un attribut d'un contrôle tel qu'une zone de texte ou une étiquette d'un autre thread que le thread d'interface graphique qui a créé le contrôle, vous devrez l'appeler ou vous pourriez recevoir un message d'erreur indiquant:

> "L'opération inter-thread n'est pas valide: contrôlez 'nom_contrôle' à partir d'un thread autre que celui sur lequel il a été créé."

Utiliser cet exemple de code sur un formulaire system.windows.forms convertira une exception avec ce message:

```
private void button4_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
    textBox1.Text = "updated"; // Throws exception
}
```

Au lieu de cela, lorsque vous souhaitez modifier le texte d'une zone de texte à partir d'un thread qui ne lui appartient pas, utilisez Control.Invoke ou Control.BeginInvoke. Vous pouvez également utiliser Control.InvokeRequired pour vérifier si l'appel du contrôle est nécessaire.

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
        textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
        textBox1.Text = "updated";
}
```

Si vous devez le faire souvent, vous pouvez écrire une extension pour les objets invocables afin de réduire la quantité de code nécessaire pour effectuer cette vérification:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
```

```
}
```

Et la mise à jour de la zone de texte à partir de n'importe quel thread devient un peu plus simple:

```
private void updatetextbox()
{
    textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```

Sachez que Control.BeginInvoke tel qu'utilisé dans cet exemple est asynchrone, ce qui signifie que le code venant après un appel à Control.BeginInvoke peut être exécuté immédiatement après, que le délégué passé ait été exécuté ou non.

Si vous devez être sûr que textBox1 est mis à jour avant de continuer, utilisez plutôt Control.Invoke, qui bloquera le thread d'appel jusqu'à ce que votre délégué ait été exécuté. Notez que cette approche peut considérablement ralentir votre code si vous lancez de nombreux appels et notez qu'il bloquera votre application si votre thread d'interface graphique attend que le thread appelant termine ou libère une ressource en attente.

Lire Filetage en ligne: https://riptutorial.com/fr/dot-net/topic/3098/filetage

Chapitre 26: Flux de données TPL

Remarques

Bibliothèques utilisées dans les exemples

`System.Threading.Tasks.Dataflow`

`System.Threading.Tasks`

`System.Net.Http`

`System.Net`

Différence entre Post et SendAsync

Pour ajouter des éléments à un bloc, vous pouvez utiliser `Post` ou `SendAsync` .

`Post` essaiera d'ajouter l'élément de manière synchrone et de renvoyer un `bool` indiquant s'il a réussi ou non. Cela peut ne pas réussir lorsque, par exemple, un bloc a atteint son `BoundedCapcity` et qu'il n'a plus de place pour les nouveaux éléments. `SendAsync` , `SendAsync` renverra une `Task<bool>` inachevée que vous pouvez `await` . Cette tâche s'achèvera à l'avenir avec un `true` résultat lorsque le bloc aura effacé sa file d'attente interne et pourra accepter plus d'éléments ou un résultat `false` s'il est en baisse permanente (par exemple suite à une annulation).

Examples

Publier sur un ActionBlock et attendre la fin

```
// Create a block with an asynchronous action
var block = new ActionBlock<string>(async hostName =>
{
    IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // Post items to the block's InputQueue for processing
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // Tell the block to complete and stop accepting new items
await block.Completion; // Asynchronously wait until all items completed processingu
```

Liaison de blocs pour créer un pipeline

```
var httpClient = new HttpClient();
```

```
// Create a block the accepts a uri and returns its contents as a string
var downloaderBlock = new TransformBlock<string, string>(
    async uri => await httpClient.GetStringAsync(uri));

// Create a block that accepts the content and prints it to the console
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// Make the downloaderBlock complete the printerBlock when its completed.
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// Link the block to create a pipeline
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// Post urls to the first block which will pass their contents to the second one.
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // Completion will propagate to printerBlock
await printerBlock.Completion; // Only need to wait for the last block in the pipeline
```

Producteur / consommateur synchrone avec BufferBlock

```
public class Producer
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    //produce the value that will be posted to buffer block
    public double Produce ( )
    {
        var value = random.NextDouble();
        Console.WriteLine($"Producing value: {value}");
        return value;
    }
}

public class Consumer
{
    //consume the value that will be received from buffer block
    public void Consume (double value) => Console.WriteLine($"Consuming value: {value}");
}

class Program
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
        //start a task that will every 1 second post a value from the producer to buffer block
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
                buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
```

```
        {
            var consumer = new Consumer();
            while(true)
            {
                consumer.Consume(buffer.Receive());
            }
        });

        Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```

Producteur Asynchrone Consommateur Avec Un Tampon Limité

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
    BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

Lire Flux de données TPL en ligne: https://riptutorial.com/fr/dot-net/topic/784/flux-de-donnees-tpl

Chapitre 27: Formulaires VB

Examples

Bonjour tout le monde dans les formulaires VB.NET

Pour afficher une boîte de message lorsque le formulaire a été affiché:

```
Public Class Form1
    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown
        MessageBox.Show("Hello, World!")
    End Sub
End Class
To show a message box before the form has been shown:

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        MessageBox.Show("Hello, World!")
    End Sub
End Class
```

Load () sera appelée en premier, et une seule fois, lors du premier chargement du formulaire. Show () sera appelé à chaque fois que l'utilisateur lance le formulaire. Activate () sera appelé chaque fois que l'utilisateur active le formulaire.

Load () s'exécutera avant l'appel de Show (), mais soyez averti: l'appel de msgBox () dans show peut entraîner l'exécution de msgBox () avant la fin de Load (). **C'est généralement une mauvaise idée de dépendre de l'ordre des événements entre Load (), Show () et similaire.**

Pour les débutants

Certaines choses que tous les débutants doivent savoir / faire pour les aider à bien démarrer avec VB .Net:

Définissez les options suivantes:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Utilisez &, pas + pour la concaténation de chaînes. Les chaînes doivent être étudiées en détail car elles sont largement utilisées.

Passez du temps à comprendre les types de valeur et de référence .

N'utilisez jamais Application.DoEvents . Faites attention à la "Attention". Lorsque vous atteignez un point où cela semble être quelque chose que vous devez utiliser, demandez.

La documentation est votre ami.

Minuteur de Formulaires

Le composant Windows.Forms.Timer peut être utilisé pour fournir à l'utilisateur des informations qui **ne** sont **pas** critiques en termes de temps. Créez un formulaire avec un bouton, une étiquette et un composant Timer.

Par exemple, il pourrait être utilisé pour montrer à l'utilisateur l'heure de la journée périodiquement.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

Mais cette minuterie n'est pas adaptée au timing. Un exemple serait de l'utiliser pour un compte à rebours. Dans cet exemple, nous allons simuler un compte à rebours de trois minutes. Cela pourrait très bien être l'un des exemples les plus ennuyeux ici.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
```

```
    End Sub

    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Après avoir cliqué sur button1, environ trois minutes passent et label1 affiche les résultats. Est-ce que label1 montre 180? Probablement pas. Sur ma machine, il y avait 182,5!

La raison de cette différence se trouve dans la documentation, "Le composant Windows Forms Timer est à thread unique et sa précision est limitée à 55 millisecondes." C'est pourquoi il ne faut pas l'utiliser pour le chronométrage.

En utilisant un peu le chronomètre et le chronomètre, nous pouvons obtenir de meilleurs résultats.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Il existe d'autres minuteries qui peuvent être utilisées selon les besoins. Cette recherche devrait aider à cet égard.

Lire Formulaires VB en ligne: https://riptutorial.com/fr/dot-net/topic/2197/formulaires-vb

Chapitre 28: Gestion de la mémoire

Remarques

Les applications à performance critique dans les applications .NET gérées peuvent être gravement affectées par le GC. Lorsque le GC s'exécute, tous les autres threads sont suspendus jusqu'à la fin. Pour cette raison, il est recommandé d'évaluer soigneusement les processus du GC et de déterminer comment les réduire au minimum.

Examples

Ressources non gérées

Quand on parle du GC et du "tas", on parle vraiment de ce qu'on appelle le *tas géré* . Les objets sur le *segment géré* peuvent accéder aux ressources qui ne sont pas sur le segment de mémoire géré, par exemple lors de l'écriture ou de la lecture d'un fichier. Un comportement inattendu peut se produire lorsqu'un fichier est ouvert pour être lu et qu'une exception se produit, empêchant le descripteur de fichier de se fermer normalement. Pour cette raison, .NET exige que les ressources non managées implémentent l'interface `IDisposable` . Cette interface a une seule méthode appelée `Dispose` sans paramètres:

```
public interface IDisposable
{
    Dispose();
}
```

Lorsque vous manipulez des ressources non gérées, vous devez vous assurer qu'elles sont correctement éliminées. Vous pouvez le faire en appelant explicitement `Dispose()` dans un bloc `finally` ou avec une instruction `using` .

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

ou

```
string textFromFile;
string filename = "SomeFile.txt";
```

```
using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

Cette dernière méthode est la méthode préférée et s'étend automatiquement à la précédente lors de la compilation.

Utilisez SafeHandle lorsque vous encapsulez des ressources non managées

Lorsque vous écrivez des wrappers pour des ressources non gérées, vous devez sous- `SafeHandle` plutôt que d'essayer d'implémenter `IDisposable` et un finaliseur vous-même. Votre sous-classe `SafeHandle` doit être aussi petite et simple que possible pour minimiser les risques de fuite. Cela signifie probablement que votre implémentation SafeHandle serait un détail d'implémentation interne d'une classe qui l'enveloppe pour fournir une API utilisable. Cette classe garantit que, même si un programme `SafeHandle` votre instance `SafeHandle` , votre `SafeHandle` non géré est libéré.

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Disclaimer: Cet exemple est une tentative pour montrer comment protéger une ressource gérée avec `SafeHandle` qui implémente `IDisposable` pour vous et qui configure les finaliseurs de manière appropriée. C'est très artificiel et probablement inutile d'allouer un morceau de mémoire de cette manière.

Lire Gestion de la mémoire en ligne: https://riptutorial.com/fr/dot-net/topic/59/gestion-de-la-memoire

Chapitre 29: Globalisation dans ASP.NET MVC en utilisant l'internationalisation intelligente pour ASP.NET

Remarques

Internationalisation intelligente pour la page ASP.NET

L'avantage de cette approche est que vous n'avez pas à encombrer les contrôleurs et les autres classes avec du code pour rechercher des valeurs à partir de fichiers .resx. Vous entourez simplement le texte dans [[[triple crochets.]]]] (Le délimiteur est configurable.) Un `HttpModule` recherche une traduction dans votre fichier .po pour remplacer le texte délimité. Si une traduction est trouvée, le `HttpModule` remplace la traduction. Si aucune traduction n'est trouvée, elle supprime les triples parenthèses et restitue la page avec le texte original non traduit.

Les fichiers .po sont un format standard pour fournir des traductions pour les applications, il existe donc un certain nombre d'applications disponibles pour les éditer. Il est facile d'envoyer un fichier .po à un utilisateur non technique afin qu'il puisse ajouter des traductions.

Examples

Configuration et configuration de base

1. Ajoutez le package nuget I18N à votre projet MVC.
2. Dans web.config, ajoutez le `i18n.LocalizingModule` à votre section `<httpModules>` ou `<modules>` .

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Ajoutez un dossier nommé "paramètres régionaux" à la racine de votre site. Créez un sous-dossier pour chaque culture que vous souhaitez prendre en charge. Par exemple, `/locale/fr/` .
4. Dans chaque dossier spécifique à la culture, créez un fichier texte nommé `messages.po` .
5. À des fins de test, entrez les lignes de texte suivantes dans votre fichier `messages.po` :

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. Ajoutez un contrôleur à votre projet qui renvoie du texte à traduire.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Exécutez votre application MVC et naviguez jusqu'à l'itinéraire correspondant à l'action de votre contrôleur, tel que http: // localhost: [votre numéro de port] / default .
   Notez que l'URL est modifiée pour refléter votre culture par défaut, telle que http: // localhost: [votre numéro de port] / en / default .
8. Remplacez `/en/` dans l'URL par `/fr/` (ou la culture que vous avez sélectionnée). La page doit maintenant afficher la version traduite de votre texte.
9. Modifiez le paramètre de langue de votre navigateur pour préférer votre autre culture et accédez à `/default` nouveau. Notez que l'URL est modifiée pour refléter votre autre culture et que le texte traduit apparaît.
10. Dans web.config, ajoutez des gestionnaires pour que les utilisateurs ne puissent pas accéder à votre dossier de `locale` .

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

Lire Globalisation dans ASP.NET MVC en utilisant l'internationalisation intelligente pour ASP.NET en ligne: https://riptutorial.com/fr/dot-net/topic/5086/globalisation-dans-asp-net-mvc-en-utilisant-l-internationalisation-intelligente-pour-asp-net

Chapitre 30: Glossaire de l'acronyme

Examples

Acronymes liés à .Net

Veuillez noter que certains termes tels que JIT et GC sont suffisamment génériques pour s’appliquer à de nombreux environnements de langage de programmation et d’exécution.

CLR: Common Language Runtime
IL: langue intermédiaire
EE: moteur d'exécution
JIT: compilateur juste à temps
GC: ramasse-miettes
OOM: Pas de mémoire
STA: appartement à filetage unique
MTA: appartement multi-thread

Lire Glossaire de l'acronyme en ligne: https://riptutorial.com/fr/dot-net/topic/10939/glossaire-de-l-acronyme

Chapitre 31: Injection de dépendance

Remarques

Problèmes résolus par injection de dépendance

Si nous n'utilisions pas d'injection de dépendance, la classe `Greeter` pourrait ressembler davantage à ceci:

```
public class ControlFreakGreeter
{
    public void Greet()
    {
        var greetingProvider = new SqlGreetingProvider(
            ConfigurationManager.ConnectionStrings["myConnectionString"].ConnectionString);
        var greeting = greetingProvider.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

C'est un "contrôle freak" car il contrôle la création de la classe qui fournit le message d'accueil, il contrôle la provenance de la chaîne de connexion SQL et contrôle la sortie.

En utilisant l'injection de dépendance, la classe `Greeter` renonce à ces responsabilités en faveur d'une responsabilité unique, en lui écrivant un message d'accueil.

Le principe d'inversion de dépendance suggère que les classes devraient dépendre des abstractions (comme les interfaces) plutôt que d'autres classes concrètes. Les dépendances directes (couplage) entre classes peuvent rendre la maintenance progressivement difficile. Selon les abstractions peuvent réduire ce couplage.

L'injection de dépendance nous aide à réaliser cette inversion de dépendance car elle conduit à écrire des classes qui dépendent des abstractions. La classe `Greeter` "sait" rien des détails d' `IGreetingProvider` de `IGreetingProvider` et `IGreetingWriter` . Il sait seulement que les dépendances injectées implémentent ces interfaces. Cela signifie que les modifications apportées aux classes concrètes qui implémentent `IGreetingProvider` et `IGreetingWriter` n'affecteront pas `Greeter` . Ne les remplaceront pas non plus par des implémentations entièrement différentes. Seules les modifications apportées aux interfaces le seront. `Greeter` est découplé.

`ControlFreakGreeter` est impossible à tester correctement. Nous voulons tester une petite unité de code, mais à la place, notre test comprendra la connexion à SQL et l'exécution d'une procédure stockée. Il faudrait également tester la sortie de la console. Parce que ControlFreakGreeter fait tellement de choses, il est impossible de tester indépendamment des autres classes.

`Greeter` est facile à tester car nous pouvons injecter des implémentations simulées de ses dépendances qui sont plus faciles à exécuter et à vérifier qu'à appeler une procédure stockée ou à lire la sortie de la console. Il ne nécessite pas de chaîne de connexion dans app.config.

Les implémentations concrètes de `IGreetingProvider` et `IGreetingWriter` pourraient devenir plus complexes. Ils peuvent à leur tour avoir leurs propres dépendances qui leur sont injectées. (Par exemple, nous injecterions la chaîne de connexion SQL dans `SqlGreetingProvider` .) Mais cette complexité est "cachée" des autres classes qui ne dépendent que des interfaces. Cela facilite la modification d'une classe sans "effet d'entraînement", ce qui nous oblige à apporter des modifications correspondantes aux autres classes.

Examples

Injection de dépendance - Exemple simple

Cette classe s'appelle `Greeter` . Sa responsabilité est de générer un message d'accueil. Il a deux *dépendances* . Il a besoin de quelque chose qui lui donnera le message d'accueil, et il lui faudra alors un moyen de sortir ce message d'accueil. Ces dépendances sont toutes deux décrites comme des interfaces, `IGreetingProvider` et `IGreetingWriter` . Dans cet exemple, ces deux dépendances sont "injectées" dans `Greeter` . (Plus d'explications suivant l'exemple.)

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

La classe de `IGreetingProvider` `Greeting` dépend à la fois de `IGreetingProvider` et de `IGreetingWriter` , mais elle n'est pas responsable de la création des instances de l'un ou de l'autre. Au lieu de cela, il les requiert dans son constructeur. Tout ce qui crée une instance de `Greeting` d' `Greeting` doit fournir ces deux dépendances. Nous pouvons appeler cela "injecter" les dépendances.

Parce que les dépendances sont fournies à la classe dans son constructeur, cela s'appelle aussi "injection constructeur".

Quelques conventions communes:

- Le constructeur enregistre les dépendances en tant que champs `private` . Dès que la classe est instanciée, ces dépendances sont disponibles pour toutes les autres méthodes non statiques de la classe.
- Les champs `private` sont en `readonly` . Une fois définies dans le constructeur, elles ne peuvent plus être modifiées. Cela indique que ces champs ne doivent pas (et ne peuvent pas) être modifiés en dehors du constructeur. Cela garantit en outre que ces dépendances seront disponibles pour toute la durée de la classe.
- Les dépendances sont des interfaces. Ce n'est pas strictement nécessaire, mais cela est fréquent car cela facilite la substitution d'une implémentation de la dépendance à une autre. Il permet également de fournir une version simulée de l'interface à des fins de test unitaire.

Comment l'injection de dépendance facilite les tests unitaires

Cela se base sur l'exemple précédent de la classe `Greeter` qui a deux dépendances, `IGreetingProvider` et `IGreetingWriter` .

L'implémentation réelle de `IGreetingProvider` peut récupérer une chaîne à partir d'un appel API ou d'une base de données. L'implémentation de `IGreetingWriter` peut afficher le `IGreetingWriter` d'accueil dans la console. Mais comme `Greeter` a ses dépendances injectées dans son constructeur, il est facile d'écrire un test unitaire qui injecte des versions simulées de ces interfaces. Dans la vraie vie, nous pourrions utiliser un framework comme Moq , mais dans ce cas, je vais écrire ces implémentations simulées.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
```

```
    }
}
```

Le comportement de `IGreetingProvider` et `IGreetingWriter` ne concerne pas ce test. Nous voulons tester que `Greeter` reçoit un message et l'écrit. La conception de `Greeter` (en utilisant l'injection de dépendance) nous permet d'injecter des dépendances simulées sans pièces mobiles compliquées. Tout ce que nous testons, c'est que `Greeter` interagit avec ces dépendances comme nous le souhaitons.

Pourquoi nous utilisons des conteneurs d'injection de dépendance (conteneurs IoC)

L'injection de dépendances consiste à écrire des classes pour qu'elles ne contrôlent pas leurs dépendances - au lieu de cela, leurs dépendances leur sont fournies ("injectées").

Ce n'est pas la même chose que d'utiliser un framework d'injection de dépendances (souvent appelé "conteneur DI", "conteneur IoC" ou simplement "conteneur") comme Castle Windsor, Autofac, SimpleInjector, Ninject, Unity ou autres.

Un conteneur facilite l'injection de dépendance. Par exemple, supposons que vous écrivez un certain nombre de classes qui reposent sur l'injection de dépendances. Une classe dépend de plusieurs interfaces, les classes qui implémentent ces interfaces dépendent d'autres interfaces, etc. Certains dépendent de valeurs spécifiques. Et juste pour le plaisir, certaines de ces classes implémentent `IDisposable` et doivent être éliminées.

Chaque cours individuel est bien écrit et facile à tester. Mais maintenant, il y a un problème différent: la création d'une instance d'une classe est devenue beaucoup plus compliquée. Supposons que nous créons une instance d'une classe `CustomerService` . Il a des dépendances et ses dépendances ont des dépendances. Construire une instance peut ressembler à ceci:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint =
ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
}
```

Vous pourriez vous demander, pourquoi ne pas mettre toute la construction géante dans une fonction distincte qui renvoie simplement `CustomerService` ? Une des raisons est que, parce que les dépendances de chaque classe y sont injectées, une classe n'est pas responsable de savoir si ces dépendances sont `IDisposable` ou de les éliminer. Il les utilise juste. Donc , si nous avions un un `GetCustomerService()` fonction qui retourne un entièrement construit `CustomerService` , cette classe peut contenir un certain nombre de ressources disponibles et aucun moyen d'accéder ou de les disposer.

Et mis à part disposer de `IDisposable` , qui veut appeler une série de constructeurs imbriqués comme ça, jamais? C'est un court exemple. Cela pourrait être bien pire. Encore une fois, cela ne signifie pas que nous avons écrit les classes dans le mauvais sens. Les classes peuvent être individuellement parfaites. Le défi consiste à les composer ensemble.

Un conteneur d'injection de dépendance simplifie cela. Cela nous permet de spécifier quelle classe ou valeur doit être utilisée pour remplir chaque dépendance. Cet exemple légèrement simplifié utilise Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

Nous appelons cela "enregistrer des dépendances" ou "configurer le conteneur". Traduit, cela dit à notre `WindsorContainer` :

- Si une classe requiert `ILogWriter` , créez une instance de `LogWriter` . `LogWriter` nécessite un chemin de fichier. Utilisez cette valeur depuis `AppSettings` .
- Si une classe requiert `IAuthorizationRepository` , créez une instance de `SqlAuthorizationRepository` . Il nécessite une chaîne de connexion. Utilisez cette valeur de la section `ConnectionStrings` .
- Si une classe requiert `ICustomerDataProvider` , créez un `CustomerApiClient` et fournissez la chaîne `AppSettings` partir de `AppSettings` .

Lorsque nous demandons une dépendance à partir du conteneur, nous appelons cela "résoudre" une dépendance. C'est une mauvaise pratique de le faire directement en utilisant le conteneur, mais c'est une autre histoire. À des fins de démonstration, nous pouvons maintenant faire ceci:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

Le conteneur sait que `CustomerService` dépend de `IAuthorizationRepository` et `ICustomerDataProvider`

. Il sait quelles classes il doit créer pour répondre à ces exigences. Ces classes, à leur tour, ont plus de dépendances et le conteneur sait comment les remplir. Il créera toutes les classes nécessaires pour pouvoir renvoyer une instance de `CustomerService` .

Si une classe nécessite une dépendance que nous n'avons pas enregistrée, comme `IDoesSomethingElse` , alors, lorsque nous essayons de résoudre `CustomerService` cela `IDoesSomethingElse` une exception claire indiquant que nous n'avons rien enregistré pour répondre à cette exigence.

Chaque structure DI se comporte un peu différemment, mais en général, elles nous permettent de contrôler la manière dont certaines classes sont instanciées. Par exemple, voulons-nous créer une instance de `LogWriter` et la fournir à chaque classe qui dépend de `ILogWriter` , ou voulons-nous en créer une à chaque fois? La plupart des conteneurs ont un moyen de spécifier cela.

Qu'en est-il des classes qui implémentent `IDisposable` ? C'est pourquoi nous appelons `container.Release(customerService);` à la fin. La plupart des conteneurs (y compris Windsor) reculeront dans toutes les dépendances créées et `Dispose` celles qui doivent être éliminées. Si `CustomerService` est `IDisposable` il en disposera également.

L'enregistrement des dépendances, comme vu ci-dessus, peut sembler plus simple à écrire. Mais lorsque nous avons beaucoup de classes avec beaucoup de dépendances, cela porte ses fruits. Et si nous devions écrire ces mêmes classes *sans* utiliser l'injection de dépendance, cette même application avec beaucoup de classes deviendrait difficile à maintenir et à tester.

Cela égratigne la *raison pour laquelle* nous utilisons des conteneurs d'injection de dépendance. *La façon dont* nous configurons notre application pour en utiliser une (et l'utiliser correctement) n'est pas un sujet: il s'agit d'un certain nombre de sujets, car les instructions et les exemples varient d'un conteneur à l'autre.

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/dot-net/topic/5085/injection-de-dependance

Chapitre 32: Invocation de plate-forme

Syntaxe

- [DllImport ("Example.dll")] static extern void SetText (string inString);
- [DllImport ("Example.dll")] static externe void GetText (StringBuilder outString);
- [MarshalAs (UnmanagedType.ByValTStr, SizeConst = 32)] texte de chaîne;
- [MarshalAs (UnmanagedType.ByValArray, SizeConst = 128)] byte [] byteArr;
- [StructLayout (LayoutKind.Sequential)] structure publique PERSON {...}
- [StructLayout (LayoutKind.Explicit)] public struct MarshaledUnion {[FieldOffset (0)] ...}

Examples

Appeler une fonction Win32 dll

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int
options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Déclarez une fonction en tant que fichier `static extern DllImportAttribute` avec sa propriété `Value` définie sur .dll name. N'oubliez pas d'utiliser l'espace de noms `System.Runtime.InteropServices` . Puis appelez-le comme une méthode statique régulière.

Platform Invocation Services se chargera de charger le fichier .dll et de trouver la finition souhaitée. Dans la plupart des cas, le P / Invoke regroupera également les paramètres et renverra la valeur vers et depuis le fichier .dll (c.-à-d. La conversion des types de données .NET en fichiers Win32 et inversement).

Utiliser l'API Windows

Utilisez pinvoke.net .

Avant de déclarer une fonction API Windows `extern` dans votre code, envisagez de la rechercher sur pinvoke.net . Ils ont probablement déjà une déclaration appropriée avec tous les types de support et de bons exemples.

Matrices de Marshalling

Tableaux de type simple

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

Tableaux de chaîne

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

Structures de marshaling

Structure simple

Signature C ++:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

C # définition

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

Struct avec des champs de tableau de taille inconnue. En passant

Signature C ++

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

Lorsqu'elle est passée du code géré au code non géré, cette

Le tableau de `data` doit être défini comme IntPtr et la mémoire doit être explicitement allouée avec `Marshal.AllocHGlobal()` (et libérée avec les `Marshal.FreeHGlobal()`):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

Struct avec des champs de tableau de taille inconnue. Recevoir

Signature C ++:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

Lorsque de telles données sont transmises à partir de code non géré et que la mémoire est allouée par les fonctions non gérées, l'appelant géré doit le recevoir dans une variable `IntPrt` et convertir le tampon en un tableau géré. Dans le cas de chaînes, il existe une `Marshal.PtrToStringAnsi()` pratique `Marshal.PtrToStringAnsi()` :

```
[StructLayout(LayoutKind.Sequential)]
```

```
public struct USER
{
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

Union de marshaling

Champs de type valeur uniquement

Déclaration C ++

```
typedef union
{
    char c;
    int i;
} CharOrInt;
```

Déclaration C #

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

Mélanger les champs de type valeur et référence

Le chevauchement d'une valeur de référence avec un type de valeur un n'est pas autorisé, vous ne pouvez donc pas simplement utiliser le ~~`FieldOffset(0) text; FieldOffset(0) i;`~~ ne compilera pas pour

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

et généralement vous devriez employer le marshaling fait sur commande. Cependant, dans des cas particuliers comme celui-ci, des techniques plus simples peuvent être utilisées:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
```

```
}
```

Lire Invocation de plate-forme en ligne: https://riptutorial.com/fr/dot-net/topic/1643/invocation-de-plate-forme

Chapitre 33: JSON dans .NET avec Newtonsoft.Json

Introduction

Le package `Newtonsoft.Json` est devenu le standard de facto pour utiliser et manipuler du texte et des objets au format JSON dans .NET. C'est un outil robuste, rapide et facile à utiliser.

Examples

Sérialiser l'objet dans JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

Cela se traduit par ce JSON: `{"Name":"Joe Smith","Age":21}`

Désérialiser un objet à partir de texte JSON

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

Cela donne un objet `Person` avec le nom "Joe Smith" et l'âge 21.

Lire JSON dans .NET avec Newtonsoft.Json en ligne: https://riptutorial.com/fr/dot-net/topic/8746/json-dans--net-avec-newtonsoft-json

Chapitre 34: La mise en réseau

Remarques

Voir aussi: Clients HTTP

Examples

Discussion TCP de base (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
```

```
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

Client SNTP de base (UdpClient)

Voir RFC 2030 pour plus de détails sur le protocole SNTP.

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from
UTC");
```

```
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)

        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

Lire La mise en réseau en ligne: https://riptutorial.com/fr/dot-net/topic/35/la-mise-en-reseau

Chapitre 35: Lecture et écriture de fichiers Zip

Introduction

La classe **ZipFile** réside dans l'espace de noms **System.IO.Compression** . Il peut être utilisé pour lire et écrire dans des fichiers Zip.

Remarques

- Vous pouvez également utiliser un MemoryStream au lieu d'un FileStream.
- Des exceptions

Exception	Condition
ArgumentException	Le flux a déjà été fermé ou les capacités du flux ne correspondent pas au mode (par exemple: essayer d'écrire dans un flux en lecture seule)
ArgumentNullException	le *flux d'* entrée est nul
ArgumentOutOfRangeException	*le mode* a une valeur non valide
InvalidDataException	Voir liste ci-dessous

Lorsqu'une **InvalidDataException** est lancée, elle peut avoir trois causes:

- Le contenu du flux ne peut pas être interprété comme une archive zip
- *le mode* est Mise à jour et une entrée est absente de l'archive ou est corrompue et ne peut pas être lue
- *le mode* est Mise à jour et une entrée est trop grande pour tenir dans la mémoire

Toutes les informations ont été extraites de cette page MSDN

Examples

Liste des contenus ZIP

Cet extrait listera tous les noms de fichiers d'une archive zip. Les noms de fichiers sont relatifs à la racine zip.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
```

```
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
    }
}
```

Extraction de fichiers à partir de fichiers ZIP

Extraire tous les fichiers dans un répertoire est très simple:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

Lorsque le fichier existe déjà, une **exception System.IO.IOException** sera lancée.

Extraire des fichiers spécifiques:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Chacune de ces méthodes produira le même résultat.

Mise à jour d'un fichier ZIP

Pour mettre à jour un fichier ZIP, le fichier doit être ouvert avec ZipArchiveMode.Update à la place.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

Il y a aussi la possibilité d'écrire directement dans un fichier dans l'archive:

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Lire Lecture et écriture de fichiers Zip en ligne: https://riptutorial.com/fr/dot-net/topic/9943/lecture-et-ecriture-de-fichiers-zip

Chapitre 36: LINQ

Introduction

LINQ (Language Integrated Query) est une expression qui extrait des données d'une source de données. LINQ simplifie cette situation en proposant un modèle cohérent pour travailler avec des données sur différents types de sources de données et de formats. Dans une requête LINQ, vous travaillez toujours avec des objets. Vous utilisez les mêmes modèles de codage de base pour interroger et transformer des données dans des documents XML, des bases de données SQL, des ensembles de données ADO.NET, des collections .NET et tout autre format pour lequel un fournisseur est disponible. LINQ peut être utilisé en C # et VB.

Syntaxe

- statique statique TSource Aggregate <TSource> (cette source IEnumerable <TSource>, Func <TSource, TSource, TSource> func)
- statique statique TAccumulate Aggregate <TSource, TAccumulate> (cette source IEnumerable <TSource>, graine TAccumulate, Func <TAccumulate, TSource, TAccumulate> func)
- statique statique TResult Aggregate <TSource, TAccumulate, TResult> (cette source IEnumerable <TSource>, graine TAccumulate, Func <TAccumulate, TSource, TAccumulate> func, Func <TAccumulate, TResult> resultSelector)
- public statique Booléen Tous <TSource> (cet attribut IEnumerable <TSource> source, Func <TSource, Boolean>)
- Booléen statique public Tout <TSource> (cette source IEnumerable <TSource>)
- booléen statique public Tout <TSource> (cet attribut IEnumerable <TSource> source, Func <TSource, Boolean>)
- public statique IEnumerable <TSource> AsEnumerable <TSource> (cette source IEnumerable <TSource>)
- Moyenne décimale statique publique (cette source IEnumerable <Decimal>)
- Double statique statique publique (cette source IEnumerable <Double>)
- Double statique statique publique (cette source IEnumerable <Int32>)
- public static Double Average (cette source IEnumerable <Int64>)
- public statique Nullable Moyenne <Decimal> (cette source IEnumerable <Nullable <Decimal >>)
- public statique Nullable <Double> Average (cette source IEnumerable <Nullable <Double >>)
- public statique Nullable <Double> Average (cette source IEnumerable <Nullable <Int32 >>)
- public statique Nullable <Double> Average (cette source IEnumerable <Nullable <Int64 >>)
- public statique Nullable <Single> Average (cette source IEnumerable <Nullable <Single >>)
- Public Single Single Average (cette source IEnumerable <Single>)
- Moyenne décimale statique publique <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Decimal>)
- statique publique Double Moyenne <TSource> (cette source IEnumerable <TSource>, Func

<TSource, Double> sélecteur)
- statique publique Double Moyenne <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int32> sélecteur)
- double statique public <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int64> sélecteur)
- public statique Nullable <Decimal> Average <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Nullable <Decimal >>)
- public statique Nullable <Double> Moyenne <TSource> (cette source IEnumerable <TSource>, Func <TSource, Sélecteur <Double >>)
- public statique Nullable <Double> Moyenne <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int32 >>)
- public statique Nullable <Double> Moyenne <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Average <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- public statique Single Average <TSource> (cette source IEnumerable <TSource>, Func <TSource, Single> sélecteur)
- public statique IEnumerable <TResult> Cast <TResult> (cette source IEnumerable)
- public statique IEnumerable <TSource> Concat <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second)
- Booléen statique public Contient <TSource> (cette source IEnumerable <TSource>, valeur TSource)
- Boolean statique public Contient <TSource> (cette source IEnumerable <TSource>, valeur TSource, IEqualityComparer <TSource> comparer)
- statique statique Int32 Count <TSource> (cette source IEnumerable <TSource>)
- statique public Int32 Count <TSource> (cet attribut IEnumerable <TSource> source, Func <TSource, Boolean>)
- public statique IEnumerable <TSource> DefaultIfEmpty <TSource> (cette source IEnumerable <TSource>)
- public statique IEnumerable <TSource> DefaultIfEmpty <TSource> (cette source IEnumerable <TSource>, TSource defaultValue)
- statique publique IEnumerable <TSource> Distinct <TSource> (cette source IEnumerable <TSource>)
- public statique IEnumerable <TSource> Distinct <TSource> (cette source IEnumerable <TSource>, IEqualityComparer <TSource> comparer)
- Élément TSource statique public <TSource> (cette source IEnumerable <TSource>, index Int32)
- statique publique TSource ElementAtOrDefault <TSource> (cette source IEnumerable <TSource>, index Int32)
- public static IEnumerable <TResult> Vide <TResult> ()
- public static IEnumerable <TSource> Sauf <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second)
- public static IEnumerable <TSource> Sauf <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second, IEqualityComparer <TSource> comparer)
- statique publique TSource First <TSource> (cette source IEnumerable <TSource>)
- public statique TSource First <TSource> (ce prédicat IEnumerable <TSource>, Func

<TSource, Boolean>)
- statique publique TSource FirstOrDefault <TSource> (cette source IEnumerable <TSource>)
- statique publique TSource FirstOrDefault <TSource> (ce prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> cléSelector, IEqualityComparer <TKey> comparer
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> sélecteur de clés, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer
- public statique IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector)
- public statique IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (cette source IEnumerable <TSource>, Func <TSource, TKey> KeySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector, IEqualityComparer <TKey> comparer
- public statique IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (cette source IEnumerable <TSource>, Func <TSource, TKey> sélecteur de clés, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector)
- public statique IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (cette source IEnumerable <TSource>, Func <TSource, TKey> sélecteur de clés, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector, IEqualityComparer <TKey> comparer
- public statique IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (this externe IEnumerable <TOuter>, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector)
- public statique IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (this externe IEnumerable <TOuter>, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector, IEqualityComparer <TKey> comparer
- public statique IEnumerable <TSource> Intersect <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second)
- public statique IEnumerable <TSource> Intersect <TSource> (ce IEnumerable <TSource> en premier, IEnumerable <TSource> second, IEqualityComparer <TSource> comparer)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (this externe IEnumerable <TOuter>, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (this externe

IEnumerable <TOuter>, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector, IEqualityComparer <TKey> comparer

- statique publique TSource Last <TSource> (cette source IEnumerable <TSource>)
- TSource statique public Last <TSource> (cet attribut IEnumerable <TSource> source, Func <TSource, Boolean>)
- statique publique TSource LastOrDefault <TSource> (cette source IEnumerable <TSource>)
- statique publique TSource LastOrDefault <TSource> (ce prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public statique Int64 LongCount <TSource> (cette source IEnumerable <TSource>)
- public statique Int64 LongCount <TSource> (ce prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public decimal max statique (cette source IEnumerable <Decimal>)
- statique publique Double Max (cette source IEnumerable <Double>)
- public statique Int32 Max (cette source IEnumerable <Int32>)
- public statique Int64 Max (cette source IEnumerable <Int64>)
- public static Nullable <Decimal> Max (cette source IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Max (cette source IEnumerable <Nullable <Double >>)
- public static Nullable <Int32> Max (cette source IEnumerable <Nullable <Int32 >>)
- public static Nullable <Int64> Max (cette source IEnumerable <Nullable <Int64 >>)
- public static Nullable <Single> Max (cette source IEnumerable <Nullable <Single >>)
- statique publique Single Max (cette source IEnumerable <Single>)
- statique publique TSource Max <TSource> (cette source IEnumerable <TSource>)
- décimal statique public Max <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Decimal>)
- statique publique Double Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, Double> sélecteur)
- public static Int32 Max <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Int32>)
- public statique Int64 Max <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Int64>)
- public statique Nullable <Décimal> Max <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Nullable <Decimal >>)
- public statique Nullable <Double> Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Double >> sélecteur)
- public statique Nullable <Int32> Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int32 >> sélecteur)
- public statique Nullable <Int64> Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int64 >>)
- public statique Nullable <Single> Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, sélecteur <Single >>)
- statique publique Single Max <TSource> (cette source IEnumerable <TSource>, Func <TSource, Single> sélecteur)
- statique publique TResult Max <TSource, TResult> (ce sélecteur IEnumerable <TSource>, Func <TSource, TResult>)
- public decimal min decimal (cette source IEnumerable <Decimal>)

- public static Double Min (cette source IEnumerable <Double>)
- public static Int32 Min (cette source IEnumerable <Int32>)
- public static Int64 Min (cette source IEnumerable <Int64>)
- public static Nullable <Decimal> Min (cette source IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Min (cette source IEnumerable <Nullable <Double >>)
- public static Nullable <Int32> Min (cette source IEnumerable <Nullable <Int32 >>)
- public static Nullable <Int64> Min (cette source IEnumerable <Nullable <Int64 >>)
- public static Nullable <Single> Min (cette source IEnumerable <Nullable <Single >>)
- public static Single Min (cette source IEnumerable <Single>)
- statique publique TSource Min <TSource> (cette source IEnumerable <TSource>)
- statique publique décimale min <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Decimal>)
- statique publique Double Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, Double> sélecteur)
- public static Int32 Min <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Int32>)
- public static Int64 Min <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Int64>)
- public static Nullable <Decimal> Min <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Nullable <Decimal >>)
- public statique Nullable <Double> Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Double >> sélecteur)
- public statique Nullable <Int32> Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, sélecteur Nullable <Int32 >>)
- public statique Nullable <Int64> Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, sélecteur <Single >>)
- statique publique Single Min <TSource> (cette source IEnumerable <TSource>, Func <TSource, Single> sélecteur)
- statique publique TResult Min <TSource, TResult> (ce sélecteur IEnumerable <TSource>, Func <TSource, TResult>)
- public statique IEnumerable <TResult> OfType <TResult> (cette source IEnumerable)
- public statique IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public statique IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> cléSelector, IComparer <TKey> comparer
- public statique IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public statique IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> cléSelector, IComparer <TKey> comparer
- public static IEnumerable <Int32> Range (démarrage Int32, nombre Int32)
- public static IEnumerable <TResult> Répéter <TResult> (élément TResult, nombre Int32)
- statique publique IEnumerable <TSource> Reverse <TSource> (cette source IEnumerable <TSource>)

- public static IEnumerable <TResult> Sélectionnez <TSource, TResult> (ce sélecteur IEnumerable <TSource>, Func <TSource, TResult>)
- public static IEnumerable <TResult> Sélectionnez <TSource, TResult> (ce sélecteur IEnumerable <TSource>, Func <TSource, Int32, TResult>)
- public statique IEnumerable <TResult> SelectMany <TSource, TResult> (cette source IEnumerable <TSource>, Func <TSource, IEnumerable <TResult >>)
- public statique IEnumerable <TResult> SelectMany <TSource, TResult> (cette source IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TResult >> sélecteur)
- public statique IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (cette source IEnumerable <TSource>, Func <TSource, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public statique IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (cette source IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public statique Boolean SequenceEqual <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second)
- public statique Boolean SequenceEqual <TSource> (ceci IEnumerable <TSource> en premier, IEnumerable <TSource> second, IEqualityComparer <TSource> comparer)
- statique publique TSource Single <TSource> (cette source IEnumerable <TSource>)
- statique publique TSource Single <TSource> (cet prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- statique publique TSource SingleOrDefault <TSource> (cette source IEnumerable <TSource>)
- statique publique TSource SingleOrDefault <TSource> (ce prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public static IEnumerable <TSource> Ignorer <TSource> (cette source IEnumerable <TSource>, nombre Int32)
- public statique IEnumerable <TSource> SkipWhile <TSource> (cet prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public statique IEnumerable <TSource> SkipWhile <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int32, Boolean> prédicat)
- Sum décimal statique public (cette source IEnumerable <Decimal>)
- statique publique Double Sum (cette source IEnumerable <Double>)
- statique statique Int32 Sum (cette source IEnumerable <Int32>)
- public statique Int64 Sum (cette source IEnumerable <Int64>)
- public statique Nullable <Decimal> Sum (cette source IEnumerable <Nullable <Decimal >>)
- public statique Nullable <Double> Sum (cette source IEnumerable <Nullable <Double >>)
- public statique Nullable <Int32> Sum (cette source IEnumerable <Nullable <Int32 >>)
- public static Nullable <Int64> Sum (cette source IEnumerable <Nullable <Int64 >>)
- public statique Nullable <Single> Sum (cette source IEnumerable <Nullable <Single >>)
- statique publique Single Sum (cette source IEnumerable <Single>)
- Somme décimale statique publique <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Decimal>)
- statique publique Double Sum <TSource> (cette source IEnumerable <TSource>, Func <TSource, Double> sélecteur)
- statique publique Int32 Sum <TSource> (ce sélecteur IEnumerable <TSource>, Func

<TSource, Int32>)
- statique publique Int64 Somme <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int64> sélecteur)
- public statique Nullable <Decimal> Sum <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Nullable <Decimal >>)
- statique publique Nullable <Double> Sum <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Double >>)
- statique publique Nullable <Int32> Somme <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int32 >> sélecteur)
- public statique Nullable <Int64> Sum <TSource> (cette source IEnumerable <TSource>, Func <TSource, Nullable <Int64 >>)
- statique publique Nullable <Single> Sum <TSource> (ce sélecteur IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- statique publique Single Sum <TSource> (cette source IEnumerable <TSource>, Func <TSource, Single> sélecteur)
- public static IEnumerable <TSource> Take <TSource> (cette source IEnumerable <TSource>, nombre Int32)
- public statique IEnumerable <TSource> TakeWhile <TSource> (ce prédicat IEnumerable <TSource>, Func <TSource, Boolean>)
- public statique IEnumerable <TSource> TakeWhile <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int32, Boolean> prédicat)
- public statique IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (cette source IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public statique IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (cette source IOrderedEnumerable <TSource>, Func <TSource, TKey> cléSelector, IComparer <TKey> comparer
- public statique IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (cette source IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public statique IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (cette source IOrderedEnumerable <TSource>, Func <TSource, TKey> cléSelector, IComparer <TKey> comparer
- statique publique TSource [] ToArray <TSource> (cette source IEnumerable <TSource>)
- Dictionnaire statique public <TKey, TSource> ToDictionary <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- Dictionnaire statique public <TKey, TSource> ToDictionary <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> KeySelector, IEqualityComparer <TKey> comparer
- Dictionnaire statique public <TKey, TElement> ToDictionary <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> KeySelector, Func <TSource, TElement> elementSelector)
- Dictionnaire statique public <TKey, TElement> ToDictionary <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> KeySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer
- Liste statique publique <TSource> ToList <TSource> (cette source IEnumerable <TSource>)
- ILookup statique public <TKey, TSource> ToLookup <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector)

- ILookup statique public <TKey, TSource> ToLookup <TSource, TKey> (cette source IEnumerable <TSource>, Func <TSource, TKey> KeySelector, IEqualityComparer <TKey> comparer)
- ILookup statique public <TKey, TElement> ToLookup <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- ILookup statique public <TKey, TElement> ToLookup <TSource, TKey, TElement> (cette source IEnumerable <TSource>, Func <TSource, TKey> sélecteur de clés, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer
- public statique IEnumerable <TSource> Union <TSource> (ceci IEnumerable <TSource> d'abord, IEnumerable <TSource> second)
- public statique IEnumerable <TSource> Union <TSource> (ceci IEnumerable <TSource> en premier, IEnumerable <TSource> second, IEqualityComparer <TSource> comparer)
- public statique IEnumerable <TSource> Où <TSource> (cette source IEnumerable <TSource>, Func <TSource, Boolean> prédicat)
- public statique IEnumerable <TSource> Où <TSource> (cette source IEnumerable <TSource>, Func <TSource, Int32, Boolean> prédicat)
- public statique IEnumerable <TResult> Zip <TFirst, TSecond, TResult> (cet IEnumerable <TFirst> en premier, IEnumerable <TSecond> second, Func <TFirst, TSecond, TResult> resultSelector)

Remarques

- Voir aussi LINQ .

Les méthodes intégrées LINQ sont des méthodes d'extension pour l'interface `IEnumerable<T>` qui vivent dans la classe `System.Linq.Enumerable` de l'assembly `System.Core` . Ils sont disponibles dans .NET Framework 3.5 et versions ultérieures.

LINQ permet la modification, la transformation et la combinaison simples de divers `IEnumerable` utilisant une syntaxe de type requête ou fonctionnelle.

Alors que les méthodes LINQ standard peuvent fonctionner sur tout `IEnumerable<T>` , y compris les tableaux simples et `List<T>` , elles peuvent également être utilisées sur des objets de base de données, où l'ensemble des expressions LINQ peut être transformé en SQL si le objet de données le prend en charge. Voir LINQ to SQL .

Pour les méthodes qui comparent des objets (tels que `Contains` et `Except`), `IEquatable<T>.Equals` est utilisé si le type T de la collection implémente cette interface. Sinon, les `Equals` standard et `GetHashCode` du type (éventuellement remplacées par les implémentations d' `Object` par défaut) sont utilisées. Il existe également des surcharges pour ces méthodes qui permettent de spécifier un `IEqualityComparer<T>` .

Pour les méthodes `...OrDefault` , la `default(T)` est utilisée pour générer des valeurs par défaut.

Référence officielle: Classe énumérable

Évaluation paresseuse

Pratiquement chaque requête qui renvoie un `IEnumerable<T>` n'est pas évaluée immédiatement; au lieu de cela, la logique est retardée jusqu'à ce que la requête soit répétée. Une implication est que chaque fois que quelqu'un parcourt un `IEnumerable<T>` créé à partir de l'une de ces requêtes, par exemple `.Where()` , la logique de requête complète est répétée. Si le prédicat est de longue durée, cela peut entraîner des problèmes de performances.

Une solution simple (lorsque vous connaissez ou pouvez contrôler la taille approximative de la séquence résultante) consiste à mettre les résultats en mémoire tampon en utilisant `.ToArray()` ou `.ToList()` . `.ToDictionary()` ou `.ToLookup()` peuvent remplir le même rôle. Bien entendu, on peut également parcourir toute la séquence et mettre les éléments en mémoire tampon selon une autre logique personnalisée.

`ToArray()` ou `ToList()` ?

Les deux `.ToArray()` et `.ToList()` parcourent tous les éléments d'une séquence `IEnumerable<T>` et enregistrent les résultats dans une collection stockée en mémoire. Utilisez les directives suivantes pour déterminer laquelle choisir:

- Certaines API peuvent nécessiter un `T[]` ou une `List<T>` .
- `.ToList()` s'exécute généralement plus rapidement et génère moins de déchets que `.ToArray()` , car ce dernier doit copier tous les éléments dans une nouvelle collection de taille fixe une fois de plus, dans presque tous les cas.
- Les éléments peuvent être ajoutés ou supprimés de la `List<T>` renvoyée par `.ToList()` , tandis que le `T[]` renvoyé par `.ToArray()` reste une taille fixe tout au long de sa durée de vie. En d'autres termes, `List<T>` est modifiable et `T[]` est immuable.
- Le `T[]` renvoyé par `.ToArray()` utilise moins de mémoire que le `List<T>` renvoyé par `.ToList()` , donc si le résultat doit être stocké pendant longtemps, préférez `.ToArray()` . La `List<T>.TrimExcess()` appels `List<T>.TrimExcess()` rendrait la différence de mémoire strictement académique, au détriment de l'avantage relatif de la vitesse de `.ToList()` .

Examples

Sélectionnez (carte)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

Ce type de fonction est généralement appelé `map` dans les langages de programmation fonctionnels.

Où (filtre)

Cette méthode retourne un IEnumerable avec tous les éléments correspondant à l'expression lambda

Exemple

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Sortie:

Foo, Fizz

Voir la démo

Commandé par

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p =>
p.Id).ToArray()));
```

```
//1,3,4,2
```

Contient

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

Sauf

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

Couper

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

Concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Notez que les doublons sont conservés dans le résultat. Si cela n'est pas souhaitable, utilisez `Union` place.

Premier (trouver)

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1
```

```
var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

L'exemple suivant lance `InvalidOperationException` avec le message "La séquence ne contient aucun élément correspondant":

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

Unique

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

La suite lance `InvalidOperationException` car il y a plus d'un élément dans la séquence:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Dernier

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

L'exemple suivant lance `InvalidOperationException` :

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0
```

```
var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
Console.WriteLine(lastMissingWord); // null
```

SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

L'exemple suivant lance `InvalidOperationException` :

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

Tout

Renvoie `true` si la collection contient des éléments répondant à la condition de l'expression lambda:

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

Tout

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Notez que la méthode `All` fonctionne en vérifiant que le premier élément est évalué comme `false` selon le prédicat. Par conséquent, la méthode retournera `true` pour *tout* prédicat dans le cas où l'ensemble est vide:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

SelectMany (carte plate)

`Enumerable.Select` renvoie un élément de sortie pour chaque élément d'entrée. Alors que `Enumerable.SelectMany` produit un nombre variable d'éléments de sortie pour chaque élément d'entrée. Cela signifie que la séquence de sortie peut contenir plus ou moins d'éléments que ceux de la séquence d'entrée.

`Lambda expressions` transmises à `Enumerable.Select` doivent renvoyer un seul élément. Les expressions lambda transmises à `Enumerable.SelectMany` doivent produire une séquence enfant. Cette séquence enfant peut contenir un nombre variable d'éléments pour chaque élément de la séquence d'entrée.

Exemple

```
class Invoice
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

Sortie:

1,2,3,4,5,6

Voir la démo

`Enumerable.SelectMany` peut également être obtenue avec une requête basée syntaxe en utilisant deux consécutifs `from` clauses:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

Somme

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10
```

```
var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

Sauter

Skip énumère les N premiers éléments sans les retourner. Une fois le numéro d'article N + 1 atteint, Skip commence à renvoyer tous les éléments énumérés:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

Prendre

Cette méthode prend les `n` premiers éléments d'un énumérable.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

SéquenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

Sens inverse

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

De type

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

Min

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

Moyenne

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

Cette méthode calcule la moyenne des nombres énumérables.

```
var cities = new[] {
    new {Population = 1000},
```

```
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

Cette méthode calcule la moyenne des énumérables à l'aide de la fonction déléguée.

Zip *: français

.NET 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

Distinct

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

Par groupe

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Regrouper les factures par pays, générer un nouvel objet avec le numéro d'enregistrement, le

total payé et la moyenne payée

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Si nous voulons seulement les totaux, aucun groupe

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Si nous avons besoin de plusieurs comptes

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

ToDictionary

Renvoie un nouveau dictionnaire à partir de la source `IEnumerable` à l'aide de la fonction keySelector fournie pour déterminer les clés. Lance une exception `ArgumentException` si keySelector n'est pas injectif (renvoie une valeur unique pour chaque membre de la collection source). Des surcharges permettent de spécifier la valeur à stocker ainsi que la clé.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

La spécification d'une simple fonction de sélection de clé créera un `Dictionary<TKey,TVal>` avec `TKey` le type de retour du sélecteur à clé, `TVal` le type d'objet d'origine et l'objet d'origine comme valeur stockée.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

La spécification d'une fonction de sélecteur de valeur créera également un `Dictionary<TKey,TVal>` avec `TKey` le type de retour du sélecteur de clé, mais `TVal` maintenant le type de retour de la fonction de sélection de valeur et la valeur renvoyée comme valeur stockée.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
```

```
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

Comme indiqué ci-dessus, les clés renvoyées par le sélecteur de clé doivent être uniques. Le suivant lancera une exception.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

Si une clé unique ne peut pas être donnée pour la collection source, envisagez d'utiliser ToLookup à la place. En apparence, ToLookup se comporte de la même manière que ToDictionary, mais dans la recherche résultante, chaque clé est associée à une collection de valeurs avec des clés correspondantes.

syndicat

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Notez que les doublons sont supprimés du résultat. Si cela n'est pas souhaitable, utilisez `Concat` place.

ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

Lister

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

Compter

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);
```

```
Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

Agrégat (pli)

Générer un nouvel objet à chaque étape:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

En utilisant le même objet dans toutes les étapes:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

En utilisant un sélecteur de résultat:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

Si une graine est omise, le premier élément devient la graine:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

Pour rechercher

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Joindre

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
```

```
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

GroupJoin

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
```

```
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}

//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

Jeter

`Cast` est différent des autres méthodes de `Enumerable` en ce qu'il s'agit d'une méthode d'extension pour `IEnumerable` , pas pour `IEnumerable<T>` . Ainsi, il peut être utilisé pour convertir des instances de la première en instances de la dernière.

Cela ne compile pas car `ArrayList` pas `IEnumerable<T>` :

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

Cela fonctionne comme prévu:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` n'effectue **pas de** conversion. Ce qui suit compile mais lève `InvalidCastException` à l'exécution:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

La méthode appropriée pour effectuer une conversion de conversion en une collection est la suivante:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

Vide

Pour créer un IEnumerable vide de int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

Ce IEnumerable vide est mis en cache pour chaque type T, de sorte que:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

Puis par

`ThenBy` ne peut être utilisé qu'après une clause `OrderBy` permettant de commander en utilisant plusieurs critères

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

Gamme

Les deux paramètres de la `Range` sont le *premier* nombre et le *nombre* d'éléments à produire (pas le dernier nombre).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

Jointure externe gauche

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };

    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

Répéter

`Enumerable.Repeat` génère une séquence d'une valeur répétée. Dans cet exemple, il génère 4 fois "Bonjour".

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
```

```
    Console.WriteLine(item);
}

/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

Lire LINQ en ligne: https://riptutorial.com/fr/dot-net/topic/34/linq

Chapitre 37: Paramètres

Examples

AppSettings from ConfigurationSettings dans .NET 1.x

Usage déconseillé

La classe ConfigurationSettings était le moyen original de récupérer les paramètres d'un assembly dans .NET 1.0 et 1.1. Il a été remplacé par la classe ConfigurationManager et la classe WebConfigurationManager .

Si vous avez deux clés portant le même nom dans la section `appSettings` du fichier de configuration, la dernière est utilisée.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Lire les AppSettings à partir de ConfigurationManager dans .NET 2.0 et versions ultérieures

La classe ConfigurationManager prend en charge la propriété `AppSettings` , qui vous permet de continuer à lire les paramètres de la section `appSettings` d'un fichier de configuration de la même manière que celle prise en charge par .NET 1.x.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Introduction à la prise en charge d'applications et de paramètres utilisateur fortement typés depuis Visual Studio

Visual Studio aide à gérer les paramètres des utilisateurs et des applications. L'utilisation de cette approche présente des avantages par rapport à la section `appSettings` du fichier de configuration.

1. Les paramètres peuvent être fortement typés Tout type pouvant être sérialisé peut être utilisé pour une valeur de paramètre.

2. Les paramètres d'application peuvent être facilement séparés des paramètres utilisateur. Les paramètres d'application sont stockés dans un seul fichier de configuration: `web.config` pour les sites Web et les applications Web et app.config, renommé en tant *qu'assembly* .exe.config, où *assembly* est le nom de l'exécutable. Les paramètres utilisateur (non utilisés par les projets Web) sont stockés dans un fichier `user.config` dans le dossier Application Data de l'utilisateur (qui varie en fonction de la version du système d'exploitation).

3. Les paramètres d'application des bibliothèques de classes peuvent être combinés en un seul fichier de configuration sans risque de collision de noms, car chaque bibliothèque de classes peut avoir sa propre section de paramètres personnalisés.

Dans la plupart des types de projet, le concepteur de propriétés de projet comporte un onglet Paramètres, qui constitue le point de départ de la création d'applications utilisateur et de paramètres personnalisés. Au départ, l'onglet Paramètres sera vide, avec un seul lien pour créer un fichier de paramètres par défaut. En cliquant sur le lien, vous obtenez ces modifications:

1. Si un fichier de configuration (`app.config` ou `web.config`) n'existe pas pour le projet, un fichier sera créé.

2. L'onglet Paramètres sera remplacé par un contrôle de grille qui vous permet de créer, modifier et supprimer des entrées de paramètres individuels.

3. Dans l'Explorateur de solutions, un élément `Settings.settings` est ajouté sous le dossier spécial Propriétés. L'ouverture de cet élément ouvre l'onglet Paramètres.

4. Un nouveau fichier avec une nouvelle classe partielle est ajouté sous le dossier `Properties` dans le dossier du projet. Ce nouveau fichier s'appelle `Settings.Designer.__` (.cs, .vb, etc.) et la classe s'appelle `Settings` . La classe est générée par le code et ne doit donc pas être modifiée, mais la classe est une classe partielle. Vous pouvez donc étendre la classe en ajoutant des membres supplémentaires dans un fichier distinct. En outre, la classe est implémentée à l'aide du modèle Singleton, exposant l'instance singleton avec la propriété nommée `Default` .

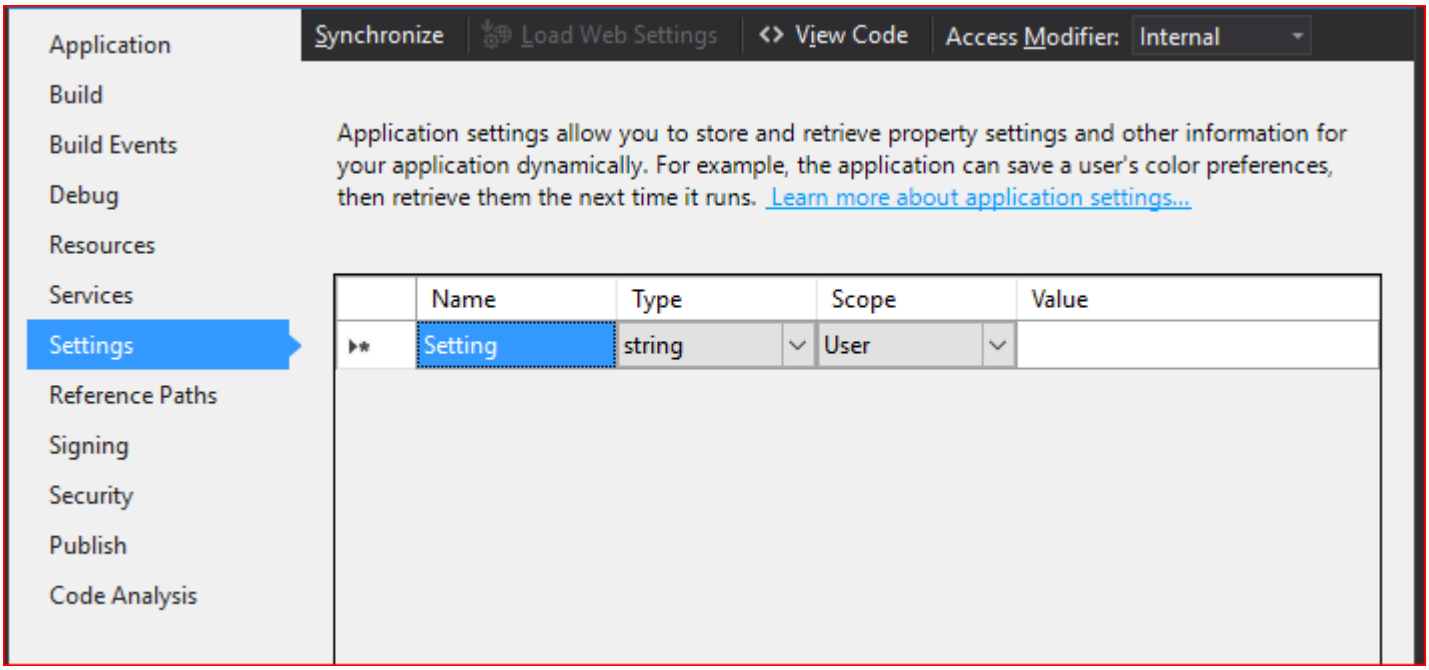
Lorsque vous ajoutez chaque nouvelle entrée à l'onglet Paramètres, Visual Studio effectue ces deux opérations:

1. Enregistre le paramètre dans le fichier de configuration, dans une section de configuration personnalisée conçue pour être gérée par la classe Settings.

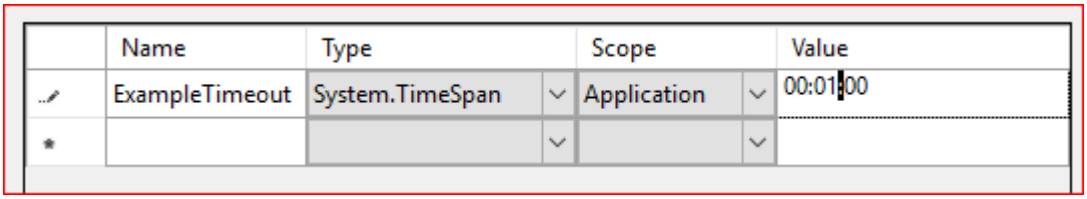
2. Crée un nouveau membre dans la classe Paramètres pour lire, écrire et présenter le paramètre dans le type spécifique sélectionné dans l'onglet Paramètres.

Lecture de paramètres fortement typés à partir de la section personnalisée du fichier de configuration

À partir d'une nouvelle classe Settings et d'une section de configuration personnalisée:



Ajoutez un paramètre d'application nommé ExampleTimeout, en utilisant l'heure System.Timespan et définissez la valeur sur 1 minute:



Enregistrez les propriétés du projet, qui enregistre les entrées de l'onglet Paramètres, puis recrée la classe Paramètres personnalisée et met à jour le fichier de configuration du projet.

Utilisez le paramètre du code (C #):

Program.cs

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

Sous les couvertures

Regardez dans le fichier de configuration du projet pour voir comment l'entrée de paramétrage de l'application a été créée:

app.config (Visual Studio met à jour cela automatiquement)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Notez que la section `appSettings` n'est pas utilisée. La section `applicationSettings` contient une section personnalisée qualifiée par un espace de noms qui contient un élément de `setting` pour chaque entrée. Le type de la valeur n'est pas stocké dans le fichier de configuration. il n'est connu que par la classe `Settings` .

Regardez dans la classe `Settings` pour voir comment il utilise la classe `ConfigurationManager` pour lire cette section personnalisée.

Settings.designer.cs (pour les projets C #)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
        }
    }
...
```

Notez qu'un objet `DefaultSettingValueAttribute` été créé pour stocker la valeur entrée dans l'onglet Paramètres du Concepteur de propriétés de projet. Si l'entrée est manquante dans le fichier de configuration, cette valeur par défaut est utilisée à la place.

Lire Paramètres en ligne: https://riptutorial.com/fr/dot-net/topic/54/parametres

Chapitre 38: Pile et tas

Remarques

Il convient de noter que lors de la déclaration d'un type de référence, sa valeur initiale sera `null` .
En effet, il ne pointe pas encore vers un emplacement en mémoire et constitue un état parfaitement valide.
Cependant, à l'exception des types nullables, les types de valeur doivent généralement toujours avoir une valeur.

Examples

Types de valeur utilisés

Les types de valeur contiennent simplement une ***valeur*** .

Tous les types de valeur sont dérivés de la classe System.ValueType , et cela inclut la plupart des types intégrés.

Lors de la création d'un nouveau type de valeur, la zone de mémoire appelée ***la pile*** est utilisée. La pile augmentera en conséquence, de la taille du type déclaré. Ainsi, par exemple, un int contient toujours 32 bits de mémoire sur la pile. Lorsque le type de valeur n'est plus dans la portée, l'espace sur la pile sera libéré.

Le code ci-dessous illustre un type de valeur affecté à une nouvelle variable. Une structure est utilisée comme un moyen pratique de créer un type de valeur personnalisé (la classe System.ValueType ne peut pas être étendue autrement).

La chose importante à comprendre est que lors de l'attribution d'un type valeur, la valeur elle-même est ***copiée*** dans la nouvelle variable, ce qui signifie que nous avons deux instances distinctes de l'objet, qui ne peuvent pas s'affecter.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";
```

```
        Console.WriteLine(                // Outputs 'False' - because
            object.ReferenceEquals(       // personA and personB are referencing
                personA,                  // different areas of memory
                personB));

        Console.WriteLine(personA.Name);  // Outputs 'Linda'
        Console.WriteLine(personB.Name);  // Outputs 'Bob'
    }
}
```

Types de référence utilisés

Les types de référence comprennent à la fois une ***référence*** à une zone de mémoire et une ***valeur*** stockée dans cette zone.
Ceci est analogue aux pointeurs en C / C ++.

Tous les types de référence sont stockés sur ce que l'on appelle ***le tas*** .
Le tas est simplement une zone de mémoire gérée où les objets sont stockés. Lorsqu'un nouvel objet est instancié, une partie du tas sera allouée pour être utilisée par cet objet et une référence à cet emplacement du segment sera renvoyée. Le tas est géré et géré par le *garbage collector* et ne permet pas une intervention manuelle.

Outre l'espace mémoire requis pour l'instance elle-même, de l'espace supplémentaire est nécessaire pour stocker la référence elle-même, ainsi que des informations temporaires supplémentaires requises par le CLR .NET.

Le code ci-dessous illustre un type de référence affecté à une nouvelle variable. Dans ce cas, nous utilisons une classe, toutes les classes sont des types de référence (même si elles sont statiques).

Lorsqu'un type de référence est affecté à une autre variable, il s'agit de la ***référence*** à l'objet copié et **non de** la valeur elle-même. Ceci est une distinction importante entre les types de valeur et les types de référence.

Cela implique que nous avons maintenant *deux* références au même objet.
Toute modification des valeurs dans cet objet sera reflétée par les deux variables.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

        personA.Name = "Linda";
```

```
        Console.WriteLine(               // Outputs 'True' - because
            object.ReferenceEquals(      // personA and personB are referencing
                personA,                 // the *same* memory location
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

Lire Pile et tas en ligne: https://riptutorial.com/fr/dot-net/topic/9358/pile-et-tas

Chapitre 39: Ports série

Examples

Opération de base

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

Liste les noms de ports disponibles

```
string[] portNames = SerialPort.GetPortNames();
```

Lecture asynchrone

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

Service d'écho de texte synchrone

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
            serialPort.Close();
        }
    }
}
```

Récepteur de message asynchrone

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
```

```
                        var port = (SerialPort)sender;
                        int bytesToRead = port.BytesToRead;
                        if (bytesToRead > buffer.Length)
                            Array.Resize(ref buffer, bytesToRead);
                        int bytesRead = port.Read(buffer, 0, bytesToRead);
                        ProcessBuffer(buffer, bytesRead);
                        break;
                    case SerialData.Eof:
                        terminateService.Set();
                        break;
                }
            }
        }
        static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
        {
            lock (eventLock)
                if (e.EventType == SerialError.TXFull)
                {
                    Console.WriteLine("Error: TXFull. Can't handle this!");
                    terminateService.Set();
                }
                else
                {
                    Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                    var port = (SerialPort)sender;
                    port.DiscardInBuffer();
                    port.DiscardOutBuffer();
                    unprocessedBuffer = null;
                    port.Write(new byte[] { NAK }, 0, 1);
                }
        }

        static void ProcessBuffer(byte[] buffer, int length)
        {
            List<byte> message = unprocessedBuffer;
            for (int i = 0; i < length; i++)
                if (buffer[i] == ETX)
                {
                    if (message != null)
                    {
                        Console.WriteLine("MessageReceived: {0}",
                            Encoding.ASCII.GetString(message.ToArray()));
                        message = null;
                    }
                }
                else if (buffer[i] == STX)
                    message = null;
                else if (message != null)
                    message.Add(buffer[i]);
            unprocessedBuffer = message;
        }
    }
}
```

Ce programme attend les messages inclus dans les octets `STX` et `ETX` et `ETX` le texte entre eux. Tout le reste est rejeté. En cas de dépassement du tampon d'écriture, il s'arrête. Sur d'autres erreurs, il réinitialise les tampons d'entrée et de sortie et attend d'autres messages.

Le code illustre:

- Lecture de port série asynchrone (voir Utilisation de `SerialPort.DataReceived`).
- Traitement des erreurs du port série (voir Utilisation de `SerialPort.ErrorReceived`).
- Implémentation de protocole sans message textuel.
- Lecture de message partielle.
    - L'événement `SerialPort.DataReceived` peut se produire plus tôt que la totalité du message (jusqu'à `ETX`). Le message entier peut également ne pas être disponible dans le tampon d'entrée (SerialPort.Read (..., ..., port.BytesToRead) ne lit qu'une partie du message). Dans ce cas, nous stockons la partie reçue (`unprocessedBuffer`) et attendons d'autres données.
- Traiter plusieurs messages à la fois.
    - L'événement `SerialPort.DataReceived` peut se produire uniquement après l'envoi de plusieurs messages par l'autre extrémité.

Lire Ports série en ligne: https://riptutorial.com/fr/dot-net/topic/5366/ports-serie

Chapitre 40: Pour chaque

Remarques

Utilisez-le du tout?

Vous pourriez faire valoir que l'intention du framework .NET est que les requêtes n'ont aucun effet secondaire et que la méthode `ForEach` provoque par définition un effet secondaire. Vous pourriez trouver votre code plus facile à entretenir et plus facile à tester si vous utilisez plutôt un `foreach` simple.

Examples

Appeler une méthode sur un objet dans une liste

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

Méthode d'extension pour IEnumerable

`ForEach()` est défini sur la classe `List<T>` , mais pas sur `IQueryable<T>` ou `IEnumerable<T>` . Vous avez deux choix dans ces cas:

ToList en premier

L'énumération (ou la requête) sera évaluée, en copiant les résultats dans une nouvelle liste ou en appelant la base de données. La méthode est ensuite appelée sur chaque élément.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

Cette méthode a une surcharge de mémoire évidente, car une liste intermédiaire est créée.

Méthode d'extension

Écrivez une méthode d'extension:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Utilisation:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Attention: Les méthodes LINQ du Framework ont été conçues avec l'intention d'être *pures* , ce qui signifie qu'elles ne produisent pas d'effets secondaires. Le seul but de la méthode `ForEach` est de produire des effets secondaires et de s'écarter des autres méthodes de cet aspect. Vous pouvez envisager d'utiliser simplement une boucle `foreach` à la place.

Lire Pour chaque en ligne: https://riptutorial.com/fr/dot-net/topic/2225/pour-chaque

Chapitre 41: ReadOnlyCollections

Remarques

Un `ReadOnlyCollection` fournit une vue en lecture seule à une collection existante (la «collection source»).

Les éléments ne sont pas directement ajoutés ou supprimés d'un `ReadOnlyCollection` . Au lieu de cela, ils sont ajoutés et supprimés de la collection source et `ReadOnlyCollection` reflétera ces modifications dans la source.

Le nombre et l'ordre des éléments à l'intérieur d'une `ReadOnlyCollection` ne peuvent pas être modifiés, mais les propriétés des éléments peuvent l'être et les méthodes peuvent être appelées, en supposant qu'elles soient dans la portée.

Utilisez un `ReadOnlyCollection` lorsque vous souhaitez autoriser le code externe à afficher votre collection sans pouvoir le modifier, tout en étant capable de modifier la collection vous-même.

Voir également

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

ReadOnlyCollections vs ImmutableCollection

Un `ReadOnlyCollection` diffère d'un `ImmutableCollection` en ce sens que vous ne pouvez pas éditer un `ImmutableCollection` une fois que vous l'avez créé - il contiendra toujours `n` éléments, et ils ne peuvent pas être remplacés ou réordonnés. Un `ReadOnlyCollection` , d'autre part, ne peut pas être modifié directement, mais des éléments peuvent toujours être ajoutés / supprimés / réordonnés à l'aide de la collection source.

Examples

Créer un ReadOnlyCollection

Utiliser le constructeur

Un `ReadOnlyCollection` est créé en transmettant un objet `IList` existant au constructeur:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Utiliser LINQ

De plus, LINQ fournit une méthode d'extension `AsReadOnly()` pour les objets `IList` :

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Remarque

En règle générale, vous souhaitez conserver la collection source en privé et autoriser un accès public à `ReadOnlyCollection` . Bien que vous puissiez créer un `ReadOnlyCollection` partir d'une liste en ligne, vous ne pourrez pas modifier la collection après l'avoir créée.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

Si vous le faites, vous pouvez envisager d'utiliser une autre structure de données, telle qu'une `ImmutableCollection` .

Mise à jour d'un ReadOnlyCollection

Un `ReadOnlyCollection` ne peut pas être modifié directement. Au lieu de cela, la collection source est mise à jour et le `ReadOnlyCollection` reflétera ces modifications. C'est la fonctionnalité clé de `ReadOnlyCollection` .

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                // ..but they can be added to the original
collection

itemCount = readOnlyGroceryList.Count;      // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now
"Vitamins"
```

Voir la démo

Avertissement: les éléments d'un ReadOnlyCollection ne sont pas intrinsèquement en lecture seule

Si la collection source est d'un type qui n'est pas immuable, les éléments accessibles via un `ReadOnlyCollection` peuvent être modifiés.

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
```

```
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

Voir la démo

Lire ReadOnlyCollections en ligne: https://riptutorial.com/fr/dot-net/topic/6906/readonlycollections

Chapitre 42: Réflexion

Examples

Qu'est-ce qu'une assemblée?

Les assemblages constituent le composant de base de toute application Common Language Runtime (CLR) . Chaque type que vous définissez, ainsi que ses méthodes, ses propriétés et son bytecode, sont compilés et empaquetés dans un assemblage.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Les assemblages sont auto-documentés: ils contiennent non seulement les types, les méthodes et leur code IL, mais également les métadonnées nécessaires pour les inspecter et les consommer, à la compilation et à l'exécution:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Les assemblées ont des noms qui décrivent leur identité complète et unique:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Si ce nom inclut un `PublicKeyToken` , il s'appelle un *nom fort* . Le nom fort d'un assembly est le processus de création d'une signature à l'aide de la clé privée qui correspond à la clé publique distribuée avec l'assembly. Cette signature est ajoutée au manifeste d'assembly, qui contient les noms et les hachages de tous les fichiers qui constituent l'assembly, et son `PublicKeyToken` devient partie intégrante du nom. Les assemblages ayant le même nom fort doivent être identiques; Les noms forts sont utilisés dans la gestion des versions et pour éviter les conflits d'assemblage.

Comment créer un objet de T en utilisant la réflexion

Utiliser le constructeur par défaut

```
T variable = Activator.CreateInstance(typeof(T));
```

Utiliser un constructeur paramétré

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Création d'objets et définition des propriétés à l'aide de la réflexion

Disons que nous avons une classe `Classy` qui a la propriété Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

définir `Propertua` utilisant la réflexion:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Obtenir un attribut d'un enum avec réflexion (et le mettre en cache)

Les attributs peuvent être utiles pour indiquer les métadonnées sur les énumérations. Obtenir cette valeur peut être lent, il est donc important de mettre en cache les résultats.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object,
object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
    }
```

Comparer deux objets avec réflexion

```
public class Equatable
{
    public string field1;

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }

    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^
field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

Remarque: cet exemple fait une comparaison basée sur les champs (ignorez les champs et propriétés statiques) pour plus de simplicité

Lire Réflexion en ligne: https://riptutorial.com/fr/dot-net/topic/44/reflexion

Chapitre 43: Réglage d'affinité des processus et des threads

Paramètres

Paramètre	Détails
affinité	entier décrivant l'ensemble des processeurs sur lesquels le processus est autorisé à s'exécuter. Par exemple, sur un système à 8 processeurs, si vous souhaitez que votre processus ne soit exécuté que sur les processeurs 3 et 4, choisissez l'affinité suivante: 00001100 qui équivaut à 12

Remarques

L'affinité du processeur d'un thread est l'ensemble des processeurs avec lesquels il a une relation. En d'autres termes, ceux sur lesquels il peut être programmé de s'exécuter.

L'affinité du processeur représente chaque processeur en tant que bit. Le bit 0 représente le processeur un, le bit 1 représente le processeur deux, etc.

Examples

Obtenir un masque d'affinité de processus

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
```

```
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Exemple d'utilisation:

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Définir le masque d'affinité du processus

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Exemple d'utilisation:

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

Lire Réglage d'affinité des processus et des threads en ligne: https://riptutorial.com/fr/dot-net/topic/4431/reglage-d-affinite-des-processus-et-des-threads

Chapitre 44: Sérialisation JSON

Remarques

JavaScriptSerializer vs Json.NET

La classe `JavaScriptSerializer` été introduite dans .NET 3.5 et est utilisée en interne par la couche de communication asynchrone de .NET pour les applications compatibles AJAX. Il peut être utilisé pour travailler avec JSON dans le code managé.

Malgré l'existence de la classe `JavaScriptSerializer` , Microsoft recommande d'utiliser la bibliothèque open source Json.NET pour la sérialisation et la désérialisation. Json.NET offre de meilleures performances et une interface plus conviviale pour mapper JSON sur des classes personnalisées (un objet `JavaScriptConverter` personnalisé serait nécessaire pour accomplir la même chose avec `JavaScriptSerializer`).

Examples

Désérialisation à l'aide de System.Web.Script.Serialization.JavaScriptSerializer

La méthode `JavaScriptSerializer.Deserialize<T>(input)` tente de désérialiser une chaîne de JSON valide dans un objet du type spécifié `<T>` , en utilisant les mappages par défaut pris en charge de manière native par `JavaScriptSerializer` .

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Remarque: L'objet `JavaScriptSerializer` été introduit dans .NET version 3.5

Désérialisation à l'aide de Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...
```

```
string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

Pour plus d'informations, consultez le site officiel Json.NET .

Remarque: Json.NET prend en charge .NET version 2 et supérieure.

Sérialisation à l'aide de Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address"
};
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Remarquez comment les propriétés (et les classes) peuvent être décorées avec des attributs pour modifier leur apparence dans la chaîne json résultante ou pour les supprimer de la chaîne json (JsonIgnore).

Vous trouverez plus d'informations sur les attributs de sérialisation Json.NET ici .

En C #, les identifiants publics sont écrits en *PascalCase* par convention. En JSON, la convention est d'utiliser *camelCase* pour tous les noms. Vous pouvez utiliser un résolveur de contrat pour convertir entre les deux.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);
```

```
    Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

Serialization-Deserialization en utilisant Newtonsoft.Json

Contrairement aux autres assistants, celui-ci utilise des helpers de classes statiques pour les sérialiser et les désérialiser, ce qui facilite son utilisation.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

Liaison dynamique

Json.NET de Newtonsoft vous permet de lier dynamiquement json (en utilisant des objets ExpandoObject / Dynamic) sans avoir besoin de créer le type explicitement.

La sérialisation

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

Désérialisation

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Notez que les clés de l'objet rawJson ont été transformées en variables membres dans l'objet dynamique.

Ceci est utile dans les cas où une application peut accepter / produire différents formats de JSON. Il est toutefois suggéré d'utiliser un niveau de validation supplémentaire pour la chaîne Json ou l'objet dynamique généré à la suite de la sérialisation / désérialisation.

Sérialisation à l'aide de Json.NET avec JsonSerializerSettings

Ce sérialiseur a quelques fonctionnalités intéressantes que le sérialiseur par défaut de .net json n'a pas, comme la gestion des valeurs Null, il vous suffit de créer les `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore});
   return result;
```

```
}
```

Un autre problème sérieux de sérialiseur dans .net est la boucle d'auto-référencement. Dans le cas d'un étudiant inscrit à un cours, son instance possède une propriété de cours et un cours comporte une collection d'étudiants, ce qui signifie une `List<Student>` qui créera une boucle de référence. Vous pouvez gérer cela avec `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Vous pouvez mettre diverses options de sérialisation comme ceci:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore, ReferenceLoopHandling =
ReferenceLoopHandling.Ignore});
   return result;
}
```

Lire Sérialisation JSON en ligne: https://riptutorial.com/fr/dot-net/topic/183/serialisation-json

Chapitre 45: Serveurs HTTP

Examples

Serveur de fichiers HTTP de base en lecture seule (HttpListener)

Remarques:

Cet exemple doit être exécuté en mode administratif.

Un seul client simultané est pris en charge.

Pour plus de simplicité, les noms de fichiers sont supposés être tous ASCII (pour la partie *nom de fichier* dans l'en - tête *Content-Disposition*) et les erreurs d'accès aux fichiers ne sont pas gérées.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);

        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
            ServerLoop();
```

```
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }

    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
```

```
    }
}
```

Serveur de fichiers HTTP de base en lecture seule (ASP.NET Core)

1 - Créez un dossier vide, il contiendra les fichiers créés dans les étapes suivantes.

2 - Créez un fichier nommé `project.json` avec le contenu suivant (ajustez le numéro de port et `rootDirectory` selon le cas):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Créez un fichier nommé `Startup.cs` avec le code suivant:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
            context.Context.Response.Headers.Add(
```

```
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };

        app.UseStaticFiles(options);
    }
}
```

4 - Ouvrez une invite de commande, accédez au dossier et exécutez:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Remarque: ces commandes ne doivent être exécutées qu'une seule fois. Utilisez la `dnvm list` pour vérifier le nombre réel de la dernière version installée du CLR principal.

5 - Démarrer le serveur avec: `dnx web` . Les fichiers peuvent maintenant être demandés à l' `http://localhost:60000/path/to/file.ext` .

Pour plus de simplicité, les noms de fichiers sont supposés être tous ASCII (pour la partie nom de fichier dans l'en-tête Content-Disposition) et les erreurs d'accès aux fichiers ne sont pas gérées.

Lire Serveurs HTTP en ligne: https://riptutorial.com/fr/dot-net/topic/53/serveurs-http

Chapitre 46: System.IO

Examples

Lecture d'un fichier texte à l'aide de StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Notez que cette surcharge du constructeur `StreamReader` détecte automatiquement l' encodage , qui peut ou non être conforme à l'encodage réel utilisé dans le fichier.

Notez que certaines méthodes pratiques permettent de lire tout le texte d'un fichier disponible dans la classe `System.IO.File` , à savoir `File.ReadAllText(path)` et `File.ReadAllLines(path)` .

Lecture / écriture de données à l'aide de System.IO.File

Voyons d'abord trois manières différentes d'extraire des données d'un fichier.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- Sur la première ligne, nous lisons toutes les données du fichier sous forme de chaîne.
- Sur la deuxième ligne, nous lisons les données du fichier dans un tableau de chaînes. Chaque ligne du fichier devient un élément du tableau.
- Le troisième, nous lisons les octets du fichier.

Ensuite, voyons trois méthodes différentes pour **ajouter des** données à un fichier. Si le fichier que vous spécifiez n'existe pas, chaque méthode créera automatiquement le fichier avant de tenter d'y ajouter les données.

```
 File.AppendAllText(file, "Here is some data that is\nappended to the file.");
 File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the
file." });
 using (StreamWriter stream = File.AppendText(file))
 {
     stream.WriteLine("Here is some data that is");
     stream.Write("appended to the file.");
 }
```

- Sur la première ligne, nous ajoutons simplement une chaîne à la fin du fichier spécifié.

- Sur la deuxième ligne, nous ajoutons chaque élément du tableau sur une nouvelle ligne du fichier.
- Enfin, à la troisième ligne, nous utilisons `File.AppendText` pour ouvrir un rédacteur qui ajoutera les données qui y sont écrites.

Et enfin, voyons trois méthodes différentes pour **écrire des** données dans un fichier. La différence entre l' *ajout* et l' *écriture* étant que l'écriture **écrase** les données dans le fichier pendant que l'ajout **ajoute** aux données du fichier. Si le fichier que vous spécifiez n'existe pas, chaque méthode créera automatiquement le fichier avant de tenter d'y écrire les données.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- La première ligne écrit une chaîne dans le fichier.
- La deuxième ligne écrit chaque chaîne du tableau sur sa propre ligne dans le fichier.
- Et la troisième ligne vous permet d'écrire un tableau d'octets dans le fichier.

Ports série utilisant System.IO.SerialPorts

Itérer sur les ports série connectés

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Instanciation d'un objet System.IO.SerialPort

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

REMARQUE : ce ne sont que trois des sept surcharges du constructeur pour le type SerialPort.

Lecture / écriture de données sur le port série

La méthode la plus simple consiste à utiliser les méthodes `SerialPort.Read` et `SerialPort.Write` . Cependant, vous pouvez également récupérer un objet `System.IO.Stream` que vous pouvez utiliser pour diffuser des données sur le port série. Pour ce faire, utilisez `SerialPort.BaseStream` .

En train de lire

```
int length = port.BytesToRead;
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

Vous pouvez également lire toutes les données disponibles:

```
string curData = port.ReadExisting();
```

Ou simplement lire la première ligne rencontrée dans les données entrantes:

```
string line = port.ReadLine();
```

L'écriture

La manière la plus simple d'écrire des données sur le port série est la suivante:

```
port.Write("here is some text to be sent over the serial port.");
```

Cependant, vous pouvez également envoyer des données de cette manière si nécessaire:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

Lire System.IO en ligne: https://riptutorial.com/fr/dot-net/topic/5259/system-io

Chapitre 47: System.Net.Mail

Remarques

Il est important de supprimer un System.Net.MailMessage car chaque pièce jointe contient un flux et ces flux doivent être libérés dès que possible. L'instruction using garantit que l'objet jetable est également disposé en cas d'exceptions

Examples

MailMessage

Voici l'exemple de la création d'un message électronique avec des pièces jointes. Après la création, nous envoyons ce message à l'aide de la classe `SmtpClient` . Le port par défaut 25 est utilisé ici.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string>
Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;
```

```
                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
            }
        }
        catch
        {
            return false;
        }
    }
}
```

Mail avec pièce jointe

`MailMessage` représente un message électronique qui peut être envoyé en utilisant la classe `SmtpClient` . Plusieurs pièces jointes (fichiers) peuvent être ajoutées à un message électronique.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

Lire System.Net.Mail en ligne: https://riptutorial.com/fr/dot-net/topic/7440/system-net-mail

Chapitre 48: System.Reflection.Emit espace de noms

Examples

Création dynamique d'un assembly

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
```

```
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);

        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance
method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as
the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);
```

```
        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another
constructor in the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save
the assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can
use e.g. ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

Lire System.Reflection.Emit espace de noms en ligne: https://riptutorial.com/fr/dot-net/topic/74/system-reflection-emit-espace-de-noms

Chapitre 49: System.Runtime.Caching.MemoryCache (ObjectCache)

Examples

Ajouter un élément au cache (Set)

La fonction Set insère une entrée de cache dans le cache en utilisant une instance CacheItem pour fournir la clé et la valeur de l'entrée de cache.

Cette fonction remplace `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

System.Runtime.Caching.MemoryCache (ObjectCache)

Cette fonction obtient le cache de formulaire d'élément existant et, si l'élément n'existe pas dans le cache, il récupère l'élément en fonction de la fonction valueFetchFactory.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

            //returns existing item form cache or add the new value if it does not exist.
```

```
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
            return default(TValue);
        }
    }
```

Lire System.Runtime.Caching.MemoryCache (ObjectCache) en ligne: https://riptutorial.com/fr/dot-net/topic/76/system-runtime-caching-memorycache--objectcache-

Chapitre 50: Système d'emballage NuGet

Remarques

NuGet.org :

> NuGet est le gestionnaire de paquets pour la plate-forme de développement Microsoft, y compris .NET. Les outils clients NuGet permettent de produire et de consommer des packages. La galerie NuGet est le référentiel de paquets central utilisé par tous les auteurs et consommateurs de paquets.

Images dans des exemples fournis par NuGet.org .

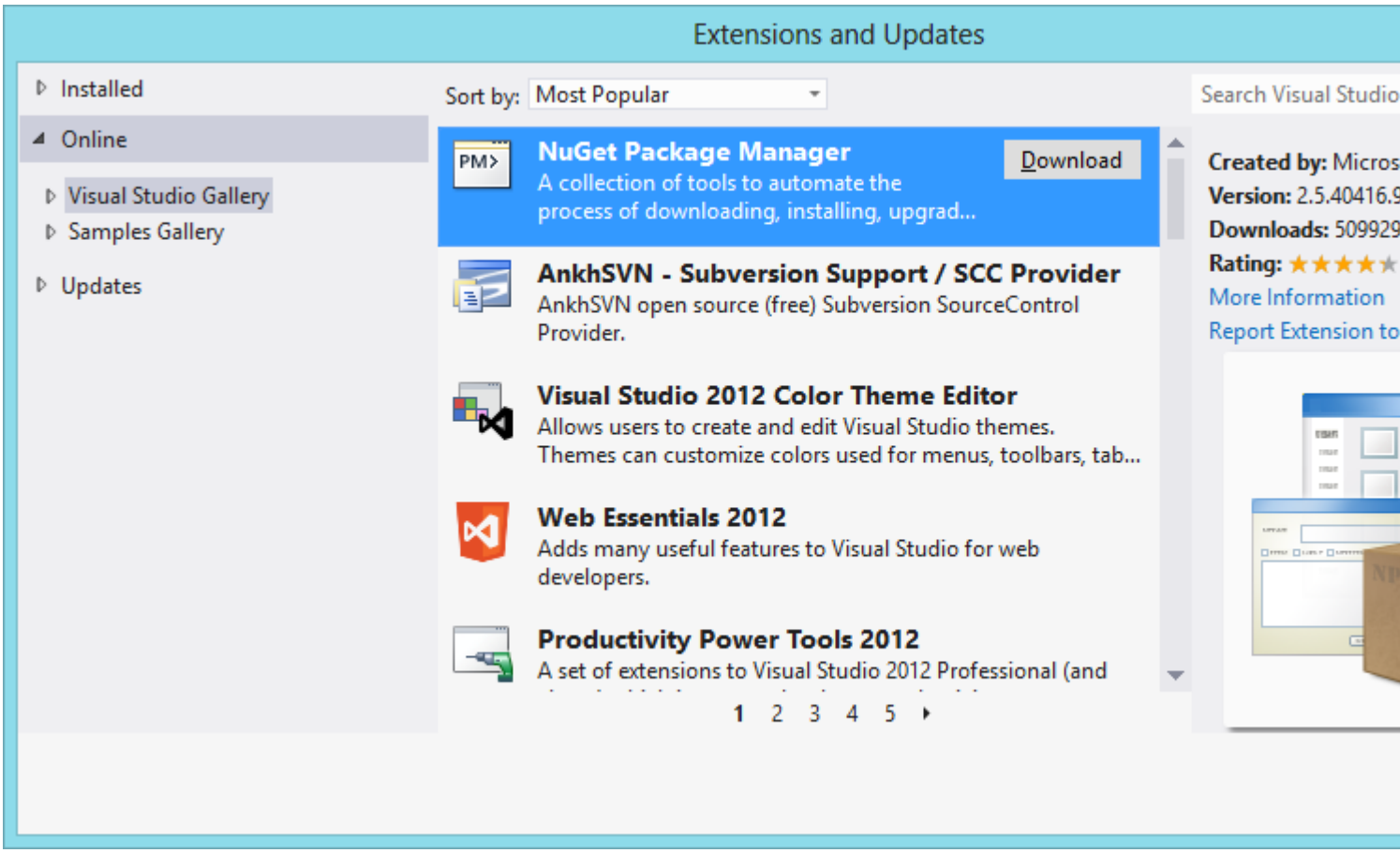
Examples

Installation du gestionnaire de paquets NuGet

Pour pouvoir gérer les packages de vos projets, vous avez besoin du gestionnaire de paquets NuGet. Ceci est une extension de Visual Studio, expliquée dans les documents officiels: Installation et mise à jour du client NuGet .

À partir de Visual Studio 2012, NuGet est inclus dans chaque édition et peut être utilisé à partir de: Outils -> Gestionnaire de packages NuGet -> Console du gestionnaire de packages.

Vous le faites via le menu Outils de Visual Studio, en cliquant sur Extensions et mises à jour:



Cela installe à la fois l'interface graphique:

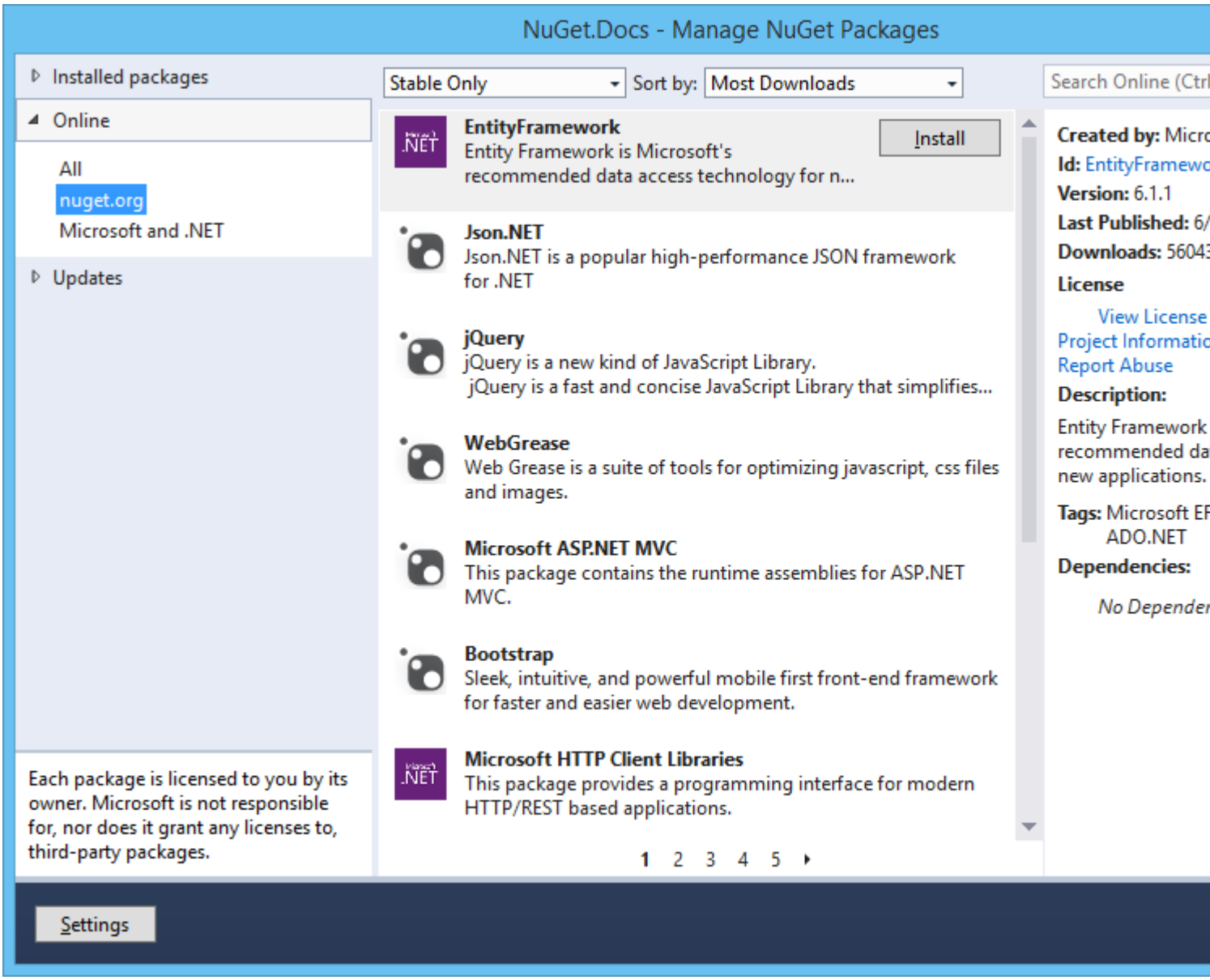
- Disponible en cliquant sur "Gérer les packages NuGet ..." sur un projet ou son dossier Références

Et la console du gestionnaire de paquets:

- Outils -> Gestionnaire de packages NuGet -> Console du gestionnaire de packages.

Gestion des packages via l'interface utilisateur

Lorsque vous cliquez avec le bouton droit sur un projet (ou son dossier Références), vous pouvez cliquer sur l'option "Gérer les packages NuGet ...". Cela montre la boîte de dialogue Gestionnaire de packages .



Gestion des packages via la console

Cliquez sur les menus Outils -> Gestionnaire de package NuGet -> Console Gestionnaire de packages pour afficher la console dans votre IDE. Documentation officielle ici .

Ici, vous pouvez, entre autres, lancer des commandes `install-package` qui installent le paquet saisi dans le "Projet par défaut" actuellement sélectionné:

```
Install-Package Elmah
```

Vous pouvez également fournir le projet dans lequel installer le package, en remplaçant le projet sélectionné dans le menu déroulant "Projet par défaut":

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Mise à jour d'un package

Pour mettre à jour un package, utilisez la commande suivante:

```
PM> Update-Package EntityFramework
```

où EntityFramework est le nom du package à mettre à jour. Notez que la mise à jour s'exécutera pour tous les projets, et est donc différente de celle de `Install-Package EntityFramework` qui s'installera uniquement dans "Projet par défaut".

Vous pouvez également spécifier un seul projet explicitement:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Désinstaller un paquet

```
PM> Uninstall-Package EntityFramework
```

Désinstallation d'un package d'un projet dans une solution

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Installation d'une version spécifique d'un package

```
PM> Install-Package EntityFramework -Version 6.1.2
```

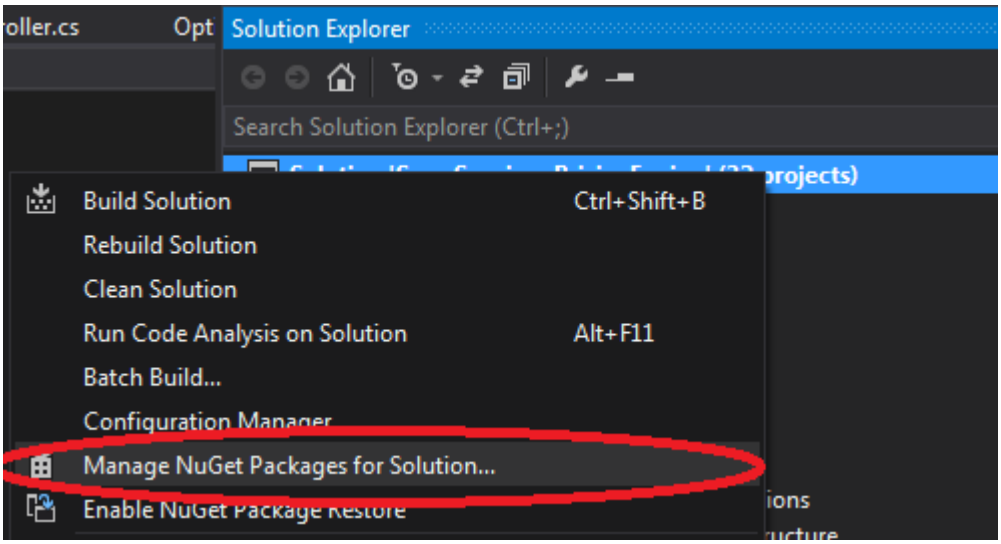
Ajout d'un flux source de package (MyGet, Klondike, ect)

```
nuget sources add -name feedname -source http://sourcefeedurl
```

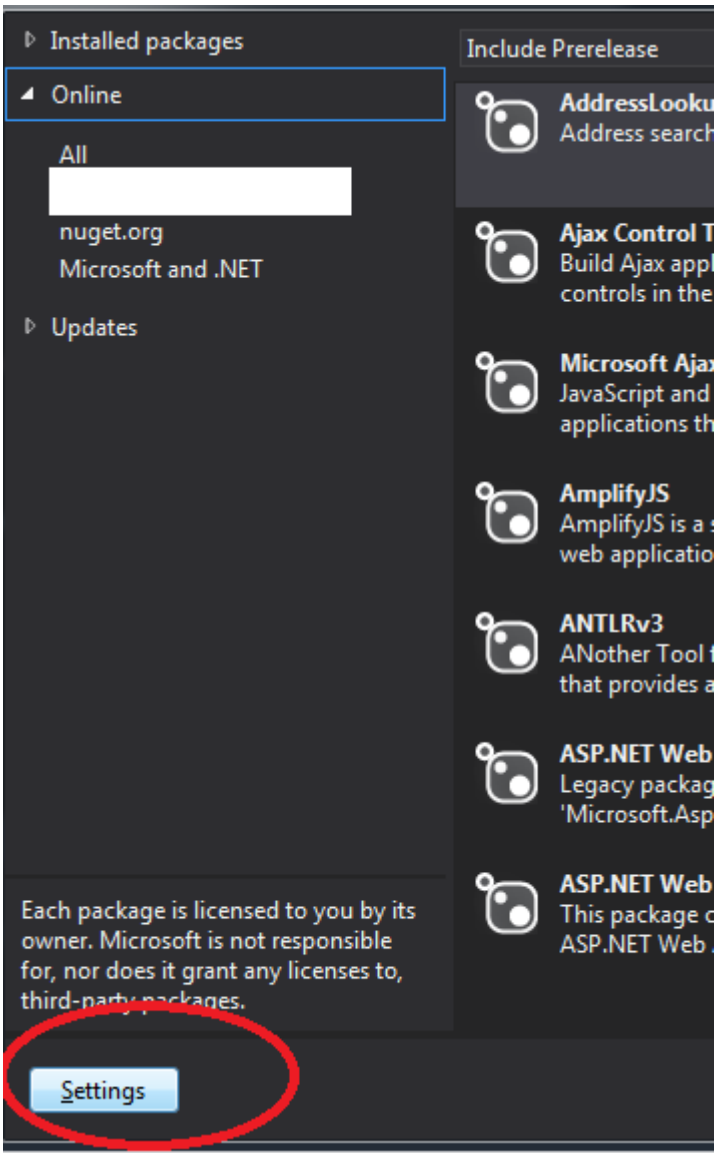
Utilisation de sources de package Nuget différentes (locales) à l'aide de l'interface utilisateur

Il est courant que l'entreprise configure son propre serveur nuget pour la distribution de packages entre différentes équipes.

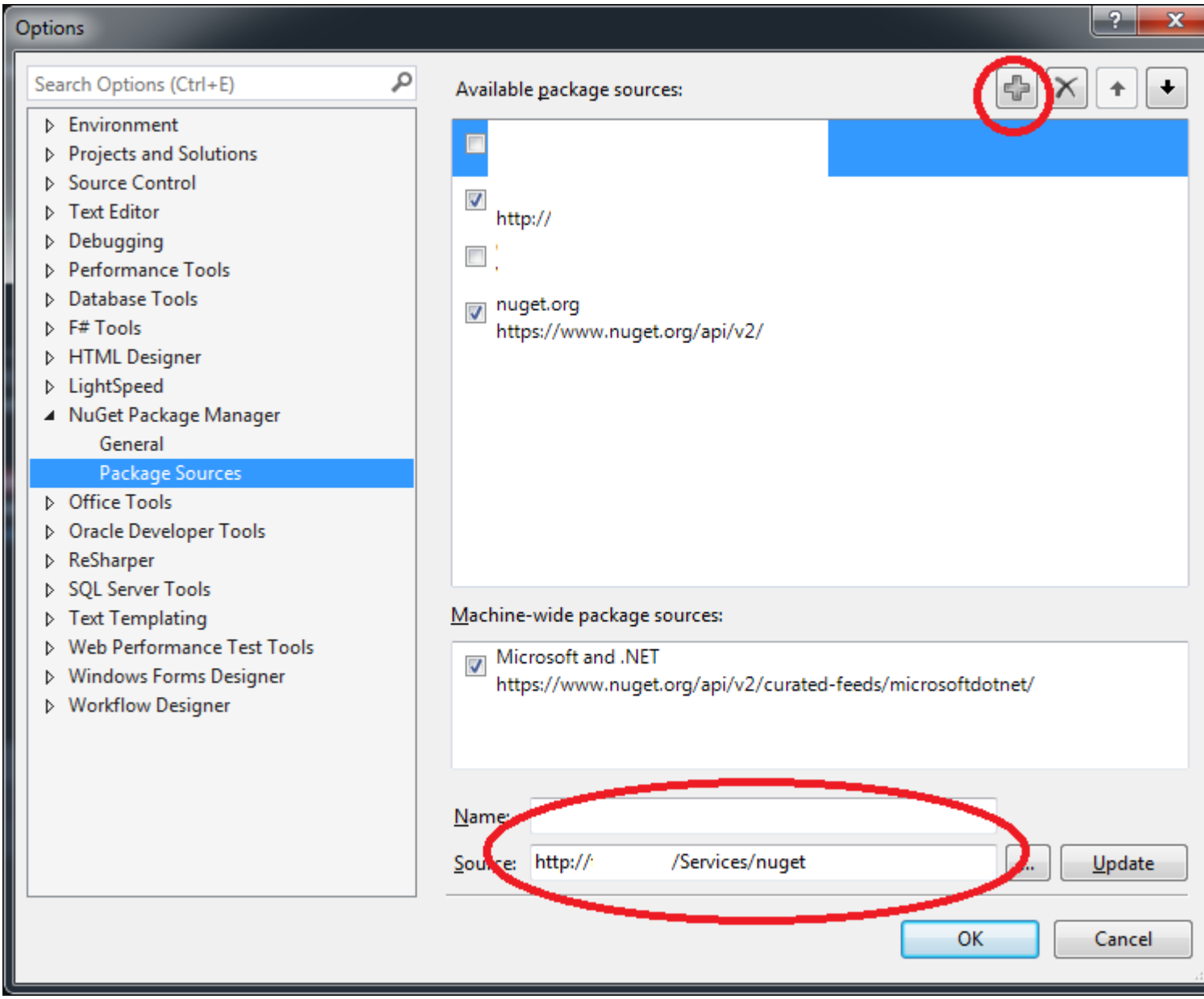
1. Accédez à l'Explorateur de solutions, cliquez sur le bouton `droit de la souris`, puis sélectionnez `Manage NuGet Packages for Solution`



2. Dans la fenêtre qui s'ouvre, cliquez sur `Settings`



3. Cliquez sur `+` dans le coin supérieur droit, puis ajoutez le nom et l'URL qui pointe vers votre serveur nuget local.



désinstaller une version spécifique du package

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

Lire Système d'emballage NuGet en ligne: https://riptutorial.com/fr/dot-net/topic/43/systeme-d-emballage-nuget

Chapitre 51: Télécharger le fichier et les données POST sur le serveur Web

Examples

Télécharger le fichier avec WebRequest

Pour envoyer un fichier et des données de formulaire en une seule requête, le contenu doit avoir le type multipart / form-data .

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Usage:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
    });
```

Lire Télécharger le fichier et les données POST sur le serveur Web en ligne: https://riptutorial.com/fr/dot-net/topic/10845/telecharger-le-fichier-et-les-donnees-post-sur-le-serveur-web

Chapitre 52: Tests unitaires

Examples

Ajout d'un projet de test d'unité MSTest à une solution existante

- Clic droit sur la solution, Ajouter un nouveau projet
- Dans la section Test, sélectionnez un projet de test unitaire
- Choisissez un nom pour l'assembly - si vous testez le projet `Foo` , le nom peut être `Foo.Tests`
- Ajouter une référence au projet testé dans les références du projet de test unitaire

Créer un exemple de méthode de test

MSTest (la structure de test par défaut) exige que vos classes de test soient décorées par un attribut `[TestClass]` et les méthodes de test avec un attribut `[TestMethod]` , et `[TestMethod]` soient publiques.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

Lire Tests unitaires en ligne: https://riptutorial.com/fr/dot-net/topic/5365/tests-unitaires

Chapitre 53: Traitement parallèle à l'aide du framework .Net

Introduction

Cette rubrique concerne la programmation multi-cœur en utilisant la bibliothèque parallèle de tâches avec le framework .NET. La bibliothèque parallèle de tâches vous permet d'écrire du code lisible par l'homme et s'adapte au nombre de cœurs disponibles. Ainsi, vous pouvez être sûr que votre logiciel se mettra à jour automatiquement avec l'environnement de mise à niveau.

Examples

Extensions parallèles

Des extensions parallèles ont été introduites avec la bibliothèque parallèle de tâches pour atteindre le parallélisme des données. Le parallélisme de données fait référence aux scénarios dans lesquels la même opération est effectuée simultanément (c'est-à-dire en parallèle) sur des éléments d'une collection ou d'un tableau source. Le .NET fournit de nouvelles constructions pour réaliser le parallélisme des données en utilisant les constructions Parallel.For et Parallel.Foreach.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

La construction Parallel.ForEach mentionnée ci-dessus utilise les cœurs multiples et améliore ainsi les performances de la même manière.

Lire Traitement parallèle à l'aide du framework .Net en ligne: https://riptutorial.com/fr/dot-net/topic/8085/traitement-parallele-a-l-aide-du-framework--net

Chapitre 54: Travailler avec SHA1 en C

Introduction

Dans ce projet, vous voyez comment travailler avec la fonction de hachage cryptographique SHA1. par exemple obtenir le hachage de la chaîne et comment craquer le hachage SHA1. source sur hub git: https://github.com/mahdiabasi/SHA1Tool

Examples

Générer la somme de contrôle SHA1 d'une fonction de fichier

D'abord, vous ajoutez System.Security.Cryptography et System.IO à votre projet.

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Lire Travailler avec SHA1 en C # en ligne: https://riptutorial.com/fr/dot-net/topic/9457/travailler-avec-sha1-en-c-sharp

Chapitre 55: Travailler avec SHA1 en C

Introduction

Dans ce projet, vous voyez comment travailler avec la fonction de hachage cryptographique SHA1. par exemple obtenir le hachage de la chaîne et comment craquer le hachage SHA1.

source compelete sur github: https://github.com/mahdiabasi/SHA1Tool

Examples

#Générer la somme de contrôle SHA1 d'un fichier

D'abord, vous ajoutez l'espace de noms System.Security.Cryptography à votre projet.

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

#Générer le hachage d'un texte

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

Lire Travailler avec SHA1 en C # en ligne: https://riptutorial.com/fr/dot-net/topic/9458/travailler-avec-sha1-en-c-sharp

Chapitre 56: Types personnalisés

Remarques

Généralement, une `struct` est utilisée uniquement lorsque les performances sont très importantes. Étant donné que les types de valeur vivent sur la pile, ils peuvent être accédés beaucoup plus rapidement que les classes. Cependant, la pile a beaucoup moins de place que le tas, donc les structures doivent rester petites (Microsoft recommande la `struct` prend pas plus de 16 octets).

Une `class` est le type le plus utilisé (parmi ces trois) en C # et correspond généralement à ce que vous devez faire en premier.

Une `enum` est utilisée chaque fois que vous pouvez avoir une liste distincte et clairement définie d'éléments à définir une seule fois (au moment de la compilation). Les énumérations sont utiles aux programmeurs en tant que références légères à certaines valeurs: au lieu de définir une liste de variables `constant` à comparer, vous pouvez utiliser un enum et obtenir un support Intellisense pour vous assurer de ne pas utiliser accidentellement une mauvaise valeur.

Examples

Définition de structure

Les structures héritent de System.ValueType, sont des types de valeur et vivent dans la pile. Lorsque les types de valeur sont passés en paramètre, ils sont transmis par valeur.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Passé par valeur signifie que la valeur du paramètre est *copiée* pour la méthode et que toute modification apportée au paramètre dans la méthode n'est pas répercutée en dehors de la méthode. Par exemple, considérez le code suivant, qui appelle une méthode nommée `AddNumbers` , en transmettant les variables `a` et `b` , de type `int` , qui est un type Value.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
```

```
    z = x + y; // now z becomes 16
}
```

Même si nous avons ajouté 5 à `x` dans la méthode, la valeur d' `a` reste inchangé, parce qu'il est un type de valeur, et cela signifie que `x` était une *copie* d' `a` « valeur s, mais pas vraiment `a` .

Rappelez-vous que les types de valeurs vivent dans la pile et sont transmis par valeur.

Définition de classe

Les classes héritent de System.Object, sont des types de référence et vivent sur le tas. Lorsque les types de référence sont passés en paramètre, ils sont transmis par référence.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Passé par référence signifie qu'une *référence* au paramètre est transmise à la méthode et que toute modification apportée au paramètre sera répercutée en dehors de la méthode lors de son retour, car la référence est *exactement le même objet en mémoire* . Utilisons le même exemple qu'auparavant, mais nous allons "envelopper" les `int` dans une classe en premier.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

Cette fois, lorsque nous avons modifié `sample.a` à `10` , la valeur de `instanceOfMyClass.a` change *également* , car elle a été *transmise par référence* . Passé par référence signifie qu'une *référence* (parfois aussi appelée *pointeur*) à l'objet a été transmise à la méthode, au lieu d'une copie de l'objet lui-même.

Rappelez-vous que les types de référence résident sur le tas et sont transmis par référence.

Enum Définition

Un enum est un type spécial de classe. Le mot clé `enum`

indique au compilateur que cette classe hérite de la classe System.Enum abstraite. Les énumérations sont utilisées pour des listes distinctes d'éléments.

```
public enum MyEnum
{
    Monday = 1,
    Tuesday,
    Wednesday,
    //...
}
```

Vous pouvez considérer une énumération comme un moyen pratique de mapper des constantes à une valeur sous-jacente. L'énumération définie ci-dessus déclare des valeurs pour chaque jour de la semaine et commence par `1` . `Tuesday` deviendrait alors automatiquement mappé sur `2` , `Wednesday` à `3` , etc.

Par défaut, les énumérations utilisent `int` comme type sous-jacent et commencent à 0, mais vous pouvez utiliser l'un des *types intégraux* suivants: `byte, sbyte, short, ushort, int, uint, long, or ulong` , et pouvez spécifier des valeurs explicites pour tout article. Si certains éléments sont explicitement spécifiés, mais que d'autres ne le sont pas, chaque élément après le dernier défini sera incrémenté de 1.

Nous utiliserons cet exemple en *lançant* une autre valeur à un *MyEnum* comme ceci:

```
MyEnum instance = (MyEnum)3; // the variable named 'instance' gets a
                             //value of MyEnum.Wednesday, which maps to 3.

int x = 2;
instance = (MyEnum)x; // now 'instance' has a value of MyEnum.Tuesday
```

Un autre type d'énumération, bien que plus complexe, s'appelle `Flags` . En *décorant* un enum avec l'attribut `Flags` , vous pouvez affecter une variable à plusieurs valeurs à la fois. Notez que pour ce faire, vous *devez* définir des valeurs explicitement dans la représentation de base 2.

```
[Flags]
public enum MyEnum
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    Saturday = 32,
    Sunday = 64
}
```

Vous pouvez maintenant comparer plusieurs valeurs à la fois, en utilisant *des comparaisons* par *bit* ou, si vous utilisez .NET 4.0 ou une version ultérieure, la méthode `Enum.HasFlag` .

```
MyEnum instance = MyEnum.Monday | MyEnum.Thursday; // instance now has a value of
                                                   // *both* Monday and Thursday,
                                                   // represented by (in binary) 0100.

if (instance.HasFlag(MyEnum.Wednesday))
{
    // it doesn't, so this block is skipped
}
else if (instance.HasFlag(MyEnum.Thursday))
{
    // it does, so this block is executed
}
```

Étant donné que la classe Enum est sous- `System.ValueType` de `System.ValueType` , elle est traitée comme un type de valeur et transmise par valeur et non par référence. L'objet de base est créé sur le tas, mais lorsque vous transmettez une valeur enum à un appel de fonction, une copie de la valeur utilisant le type de valeur sous-jacent de Enum (généralement System.Int32) est insérée dans la pile. Le compilateur suit l'association entre cette valeur et l'objet de base créé sur la pile. Voir Classe ValueType (System) (MSDN) pour plus d'informations.

Lire Types personnalisés en ligne: https://riptutorial.com/fr/dot-net/topic/57/types-personnalises

Chapitre 57: Utiliser le progrès et IProgress

Examples

Rapport de progression simple

`IProgress<T>` peut être utilisé pour signaler la progression de certaines procédures à une autre procédure. Cet exemple montre comment créer une méthode de base indiquant ses progrès.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Sortie:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Notez que lorsque vous exécutez ce code, vous pouvez voir des numéros être sortis dans le désordre. En effet, la `IProgress<T>.Report()` est exécutée de manière asynchrone et ne convient donc pas aux situations dans lesquelles la progression doit être signalée dans l'ordre.

Utiliser IProgress

Il est important de noter que la `System.Progress<T>` ne dispose pas de la méthode `Report()` . Cette méthode a été implémentée explicitement à partir de l' `IProgress<T>` et doit donc être appelée sur un `Progress<T>` lorsqu'elle est `IProgress<T>` en `IProgress<T>` .

```
var p1 = new Progress<int>();
```

```
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works

var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

Lire Utiliser le progrès et IProgress en ligne: https://riptutorial.com/fr/dot-net/topic/5628/utiliser-le-progres--t--et-iprogress--t-

Chapitre 58: Vue d'ensemble de l'API TPL (Task Parallel Library)

Remarques

La bibliothèque parallèle de tâches est un ensemble de types publics et d'API qui simplifient considérablement l'ajout de parallélisme et de simultanéité à une application. .Net. TPL a été introduit dans .Net 4 et constitue la méthode recommandée pour écrire du code multi-threadé et parallèle.

TPL prend en charge la planification du travail, l'affinité des threads, la prise en charge de l'annulation, la gestion des états et l'équilibrage de charge pour que le programmeur puisse se concentrer sur la résolution de problèmes plutôt que sur des détails communs de bas niveau.

Examples

Effectuer le travail en réponse à un clic de bouton et mettre à jour l'interface utilisateur

Cet exemple montre comment vous pouvez répondre à un clic de bouton en effectuant un travail sur un thread de travail, puis mettre à jour l'interface utilisateur pour indiquer l'achèvement.

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the
UI thread.

}
```

Lire Vue d'ensemble de l'API TPL (Task Parallel Library) en ligne: https://riptutorial.com/fr/dot-net/topic/5164/vue-d-ensemble-de-l-api-tpl--task-parallel-library-

Chapitre 59: XmlSerializer

Remarques

N'utilisez pas le `XmlSerializer` pour analyser le `HTML` . Pour cela, des bibliothèques spéciales sont disponibles comme le HTML Agility Pack

Examples

Sérialiser l'objet

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

Désérialiser l'objet

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Comportement: Mapper le nom de l'élément à la propriété

```
<Foo>
    <Dog/>
</Foo>
```

-

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

Comportement: Mappez le nom du tableau sur la propriété (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

-

```
public class Store
{
    [XmlArray("Articles")]
    public List<Product> Products {get; set; }
}
```

Formatage: format DateTime personnalisé

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat,
CultureInfo.InvariantCulture); }
    }
}
```

Construire efficacement plusieurs sérialiseurs avec des types dérivés spécifiés dynamiquement

D'où nous venons

Parfois, nous ne pouvons pas fournir toutes les métadonnées requises pour le framework XmlSerializer dans l'attribut. Supposons que nous ayons une classe de base d'objets sérialisés et que certaines des classes dérivées soient inconnues de la classe de base. Nous ne pouvons pas placer d'attribut pour toutes les classes qui ne sont pas connues au moment de la conception du type de base. Nous pourrions avoir une autre équipe développant certaines des classes dérivées.

Que pouvons-nous faire

Nous pouvons utiliser un `new XmlSerializer(type, knownTypes)` , mais ce serait une opération O (N ^ 2) pour N sérialiseurs, du moins pour découvrir tous les types fournis dans les arguments:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
```

```
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

Dans cet exemple, le type Base ne connaît pas ses types dérivés, ce qui est normal dans la POO.

Le faire efficacement

Heureusement, il existe une méthode qui résout ce problème particulier: fournir efficacement des types connus pour plusieurs sérialiseurs:

System.Xml.Serialization.XmlSerializer.FromTypes, méthode (type [])

> La méthode FromTypes vous permet de créer efficacement un tableau d'objets XmlSerializer pour traiter un tableau d'objets Type.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Voici un exemple de code complet:

```
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Linq;
using System.Linq;

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
            Serialize(sampleObject);
        }
```

```
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
}
```

Sortie:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
This error was anticipated,
we have not supplied a derived class.
System.InvalidOperationException: There was an error generating the XML document. --->
System.InvalidOperationException: The type Program+Derived was not expected. Use the
XmlInclude or SoapInclude attribute to specify types that are not known statically.
   at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String
n, String ns, Base o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   --- End of inner exception stack trace ---
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
   at Program.Serialize[T](T o)
   at Program.Main()
Now trying to serialize with all of the type information:
```

```
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Base xsi:type="Derived">
    <JustSomePropInBase>0</JustSomePropInBase>
    <JustSomePropInDerived>0</JustSomePropInDerived>
  </Base>
</Container>
Slides down well this time!
```

Qu'est-ce que dans la sortie

Ce message d'erreur recommande ce que nous avons essayé d'éviter (ou ce que nous ne pouvons pas faire dans certains scénarios): référencer les types dérivés de la classe de base:

```
Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.
```

Voici comment nous obtenons notre classe dérivée dans le XML:

```
<Base xsi:type="Derived">
```

`Base` correspond au type de propriété déclaré dans le type `Container` , et `Derived` étant le type d'instance réellement fourni.

Voici un exemple de violon de travail

Lire XmlSerializer en ligne: https://riptutorial.com/fr/dot-net/topic/31/xmlserializer

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec .NET Framework	Adriano Repetti, Alan McBee, ale10ander, Andrew Jens, Andrew Morton, Andrey Shchekin, Community, Daniel A. White, Ehsan Sajjad, harriyott, hillary.fraley, Ian, James Thorpe, Jamie Rees, Joel Martinez, Kevin Montrose, Lirrik, MarcinJuraszek, matteeyah, naveen, Nicholas Sizer, Paweł Izdebski, Peter, Peter Gordon, Peter Hommel, PSN, Richard Lander, Rion Williams, Robert Columbia, RubberDuck, SeeuD1, Serg Rogovtsev, Squidward, Stephen Leppik, Steven Doggart, svick, ɬolɐəz əuɬ qoq
2	.NET Core	Mihail Stancescu
3	ADO.NET	Akshay Anand, Andrew Morton, Daniel A. White, DavidG, Drew, elmer007, Hamid, Harjot, Heinzi, Igor, user2321864
4	Analyse de date et heure	GalacticCowboy, John
5	Arbres d'expression	Akshay Anand, George Polevoy, Jim, n.podbielski, Pavel Mayorov, RamenChef, Stephen Leppik, Stilgar, wangengzheng
6	Bibliothèque parallèle de tâches (TPL)	Adi Lester, Aman Sharma, Andrew, i3arnon, Jacobr365, JamyRyals, Konamiman, Mathias Müller, Mert Gülsoy, Mikhail Filimonov, Pavel Mayorov, Pavel Voronin, RamenChef, Thomas Bledsoe, TorbenJ
7	Cadre d'extensibilité gérée	Joe Amenta, Kirk Broadhurst, RamenChef
8	Chiffrement / Cryptographie	Alexander Mandt, Daniel A. White, demonplus, Jagadisha B S, lokusking, Matt
9	Classe SpeechRecognitionEngine pour reconnaître la parole	ProgramFOX, RamenChef
10	Classe System.IO.File	Adriano Repetti, delete me

11	Clients HTTP	CodeCaster, Konamiman, MuiBienCarlota
12	CLR	Gajendra, starbeamrainbowlabs, Theodoros Chatzigiannakis
13	Collecte des ordures	avat
14	Collections	Alan McBee, Aman Sharma, Anik Saha, Daniel A. White, demonplus, Felipe Oriani, harriyott, Ian, Mark C., Ravi A., Virtlink
15	Compilateur JIT	Krikor Ailanjian
16	Contextes de synchronisation	DLeh, Gusdor
17	Contrats de code	JJS, Matthew Whited, RamenChef
18	Cordes	Adriano Repetti, Alexander Mandt, Matt, Pavel Voronin, RamenChef
19	Des exceptions	Adi Lester, Akshay Anand, Alan McBee, Alfred Myers, Arvin Baccay, BananaSft, CodeCaster, Dave R., Kritner, Mafii, Matt, Rob, Sean, starbeamrainbowlabs, STW, Yousef Al-Mulla
20	Diagnostique du systeme	Adi Lester, Bassie, Fredou, Ogglas, Ondřej Štorc, RamenChef
21	Dictionnaires	Adriano Repetti, Bjørn-Roger Kringsjå, Daniel Plaisted, Darrel Lee, Felipe Oriani, George Duckett, George Polevoy, hatchet, Hogan, Ian, LegionMammal978, Luke Bearl, Olivier Jacot-Descombes, RamenChef, Ringil, Robert Columbia, Stephen Byrne, the berserker, Tomáš Hübelbauer
22	Ecrire et lire le flux StdErr	Aleks Andreev
23	Expressions régulières (System.Text.RegularExpressions)	BrunoLM, Denuath, Matt dc, tehDorf
24	Fichier d'entrée / sortie	ale10ander, Alexander Mandt, Ingenioushax, Nitram
25	Filetage	Behzad, Martijn Pieters, Mellow
26	Flux de données TPL	i3arnon, Jacobr365, Nikola.Lukovic, RamenChef

27	Formulaires VB	ale10ander, dbasnett
28	Gestion de la mémoire	Big Fan, binki, DrewJordan
29	Globalisation dans ASP.NET MVC en utilisant l'internationalisation intelligente pour ASP.NET	Scott Hannen
30	Glossaire de l'acronyme	Tanveer Badar
31	Injection de dépendance	Phil Thomas, Scott Hannen
32	Invocation de plate-forme	Dmitry Egorov, Imran Ali Khan
33	JSON dans .NET avec Newtonsoft.Json	DLeh
34	La mise en réseau	Konamiman
35	Lecture et écriture de fichiers Zip	Arxae
36	LINQ	A. Raza, Adil Mammadov, Akshay Anand, Alexander V., Benjamin Hodgson, Blachshma, Bradley Grainger, Bruno Garcia, Carlos Muñoz, CodeCaster, dbasnett, DoNot , dotctor, Eduardo Molteni, Ehsan Sajjad, GalacticCowboy, H. Pauwelyn, Haney, J3soon, jbtule, jnovo, Joe Amenta, Kilazur, Konamiman, MarcinJuraszek, Mark Hurd, McKay, Mellow, Mert Gülsoy, Mike Stortz, Mr.Mindor, Nate Barbettini, Pavel Voronin, Ruben Steins, Salvador Rubio Martinez, Sammi, Sergio Domínguez, Sidewinder94
37	Paramètres	Alan McBee
38	Pile et tas	Hywel Rees
39	Ports série	Dmitry Egorov
40	Pour chaque	Dr Rob Lang, just.ru, Lucas Trzesniewski
41	ReadOnlyCollections	tehDorf
42	Réflexion	Aleks Andreev, Bjørn-Roger Kringsjå, demonplus, Jean-Baptiste Noblot, Jigar, JJP , Kirk Broadhurst, Lorenzo Dematté, Matas Vaitkevicius, NetSquirrel, Pavel Mayorov, Peter, smdrager, Terry, user1304444, void
43	Réglage d'affinité des processus et des	MSE, RamenChef

	threads	
44	Sérialisation JSON	Akshay Anand, Andrius, Eric, hasan, M22an, PedroSouki, Thriggle, Tolga Evcimen
45	Serveurs HTTP	Devon Burriss, Konamiman
46	System.IO	CodeCaster, Daniel A. White, demonplus, Filip Frącz, RoyalPotato
47	System.Net.Mail	demonplus, Steve, vicky
48	System.Reflection.Emit espace de noms	Luaan, NikolayKondratyev, RamenChef, toddmo
49	System.Runtime.Caching.MemoryCache (ObjectCache)	Guanxi, RamenChef
50	Système d'emballage NuGet	Andrey Shchekin, Anik Saha, Ashtonian, CodeCaster, Daniel A. White, Matas Vaitkevicius, Ozair Kafray
51	Télécharger le fichier et les données POST sur le serveur Web	Aleks Andreev
52	Tests unitaires	Axarydax
53	Traitement parallèle à l'aide du framework .Net	Yahfoufi
54	Travailler avec SHA1 en C #	mahdi abasi
55	Types personnalisés	Alan McBee, DrewJordan, matteeyah
56	Utiliser le progrès et IProgress	DLeh
57	Vue d'ensemble de l'API TPL (Task Parallel Library)	Gusdor, Jacobr365
58	XmlSerializer	Aphelion, George Polevoy, RamenChef, Rowland Shaw, Thomas Levesque, void, Yogi