無料電子ブック

学習

C# Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c#

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

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

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

1: Cをいめる

Cは、マルチパラダイムのMicrosoftのCプログラミングです。 Cは、Windows、Mac OS XおよびLinuxでなバイトコードであるCILにコンパイルされたです。

バージョン1.0,2.0,5.0はECMA ECMA-334としてによってされ、のCのがである。

バージョン

バージョン	
1.0	2002-01-01
1.2	2003-04-01
2.0	2005-09-01
3.0	2007-08-01
4.0	2010-04-01
5.0	2013-06-01
6.0	2015-07-01
7.0	2017-03-07

Examples

しいコンソールアプリケーションをするVisual Studio

1. Visual Studioをく
2. ツールバーの[ファイル] → [プロジェクト]
3. コンソールアプリケーションのプロジェクトタイプをします。
4. ソリューションエクスプローラで`Program.cs`ファイルをきます。
5. `Main()`のコードをします。

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
```

```
            and disappearing before the programmer can see the contents
            of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. ツールバーで、[デバッグ] - > [デバッグ]をクリックするか、 **F5**キーまたは**Ctrl + F5**キーでデバッガなしでをしてプログラムをします。

のライブデモ

- `class Program`はクラスです。クラス`Program`は、 `Program`がするデータとメソッドのがまれています。クラスには、のメソッドがまれています。メソッドは、クラスのをします。ただし、 `Program`クラスには`Main`メソッドしかありません。
- `static void Main()`はすべてのCプログラムのエントリポイントである`Main`メソッドをします。 `Main`メソッドは、クラスののをします。クラスごとに1つの`Main`メソッドしかされません。
- `System.Console.WriteLine("Hello, world!");`メソッドは、コンソールウィンドウのとして、えられたデータこのでは`Hello, world!` をします。
- `System.Console.ReadKey()`は、メッセージをしたにプログラムがしないようにします。これは、ユーザーがキーボードのキーをすのをつことによっています。ユーザーからのキーをすとプログラムがします。 `main()`メソッドののコードがすると、プログラムはします。

コマンドラインの

コマンドラインでコンパイルするには、 MSビルドツールパッケージのの`MSBuild`または`csc.exe` C コンパイラをします。

このをコンパイルするには、 `HelloWorld.cs`があるディレクトリとじディレクトリでのコマンドをします。

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

また、1つのアプリケーションに2つのなメソッドがあるもあります。この、 コンソールにのコマンドをしてするメインメソッドをコンパイラにえるがありますクラス`ClassA`もHelloWorldのじ`HelloWorld.cs`ファイルにmainメソッドがあるとします

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

HelloWorldはです

これは、 **.NET**フレームワーク**v4.0**がにするパスです。 .NETバージョンにってパスをします。また、32ビット.NET Frameworkをしているは、 **framework64**ではなくディレクトリがフレームワークになるがあります。 Windowsコマンドプロンプトから、のコマンド32ビットフレームワークののコマンドをして、csc.exeのすべてのフレームワークパスをできます。

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

```
C:\Users\Main\Documents\helloworld>%WINDIR%\Microsoft.NET\Framework\v4.0.30319\c
sc.exe HelloWorld.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.


C:\Users\Main\Documents\helloworld>dir
 Volume in drive C is System


 Directory of C:\Users\Main\Documents\helloworld

07/26/2016  03:43 PM    <DIR>          .
07/26/2016  03:43 PM    <DIR>          ..
07/26/2016  03:41 PM               258 HelloWorld.cs
07/26/2016  03:43 PM             3,584 HelloWorld.exe
               2 File(s)          3,842 bytes
               2 Dir(s)  99,699,073,024 bytes free

C:\Users\Main\Documents\helloworld>
```

HelloWorld.exeというのファイルがじディレクトリにするはずです。コマンドプロンプトからプログラムをするには、にファイルのを□□し 、のようにEnterキーをします。

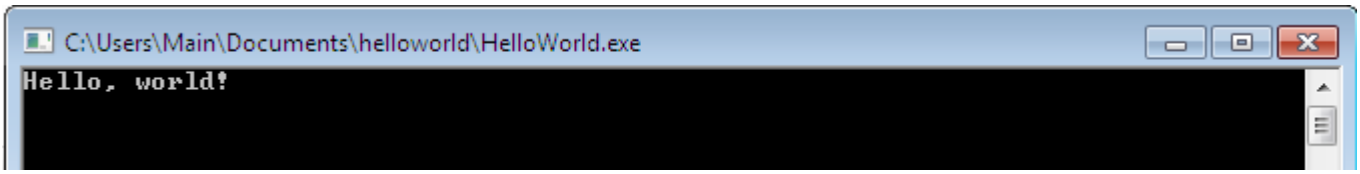
```
HelloWorld.exe
```

これにより、

    こんにちは

```
C:\Users\Main\Documents\helloworld>HelloWorld
Hello, world!
```

また、ファイルをダブルクリックし、 " **Hello**、world "というメッセージがされたしいコンソールウィンドウをすることもできます。

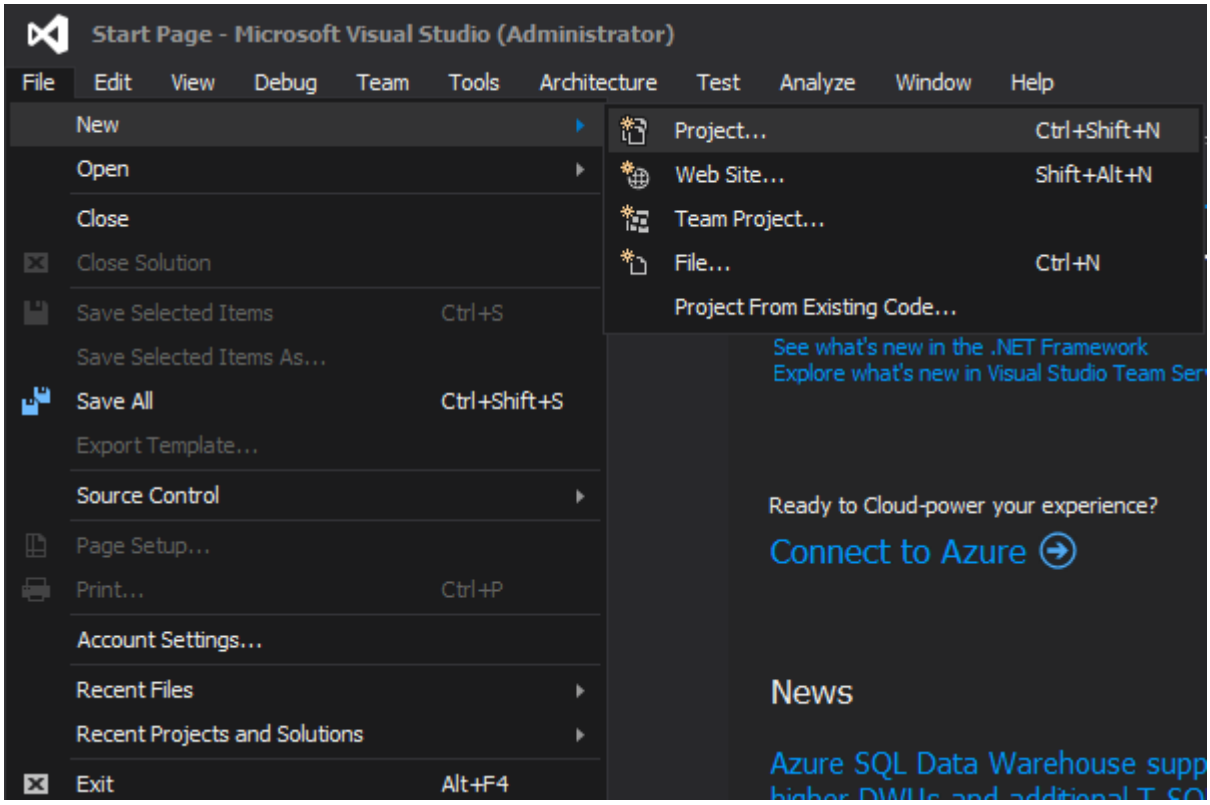


Visual Studioコンソールアプリケーションでしいプロジェクトをし、デバッグモードでする

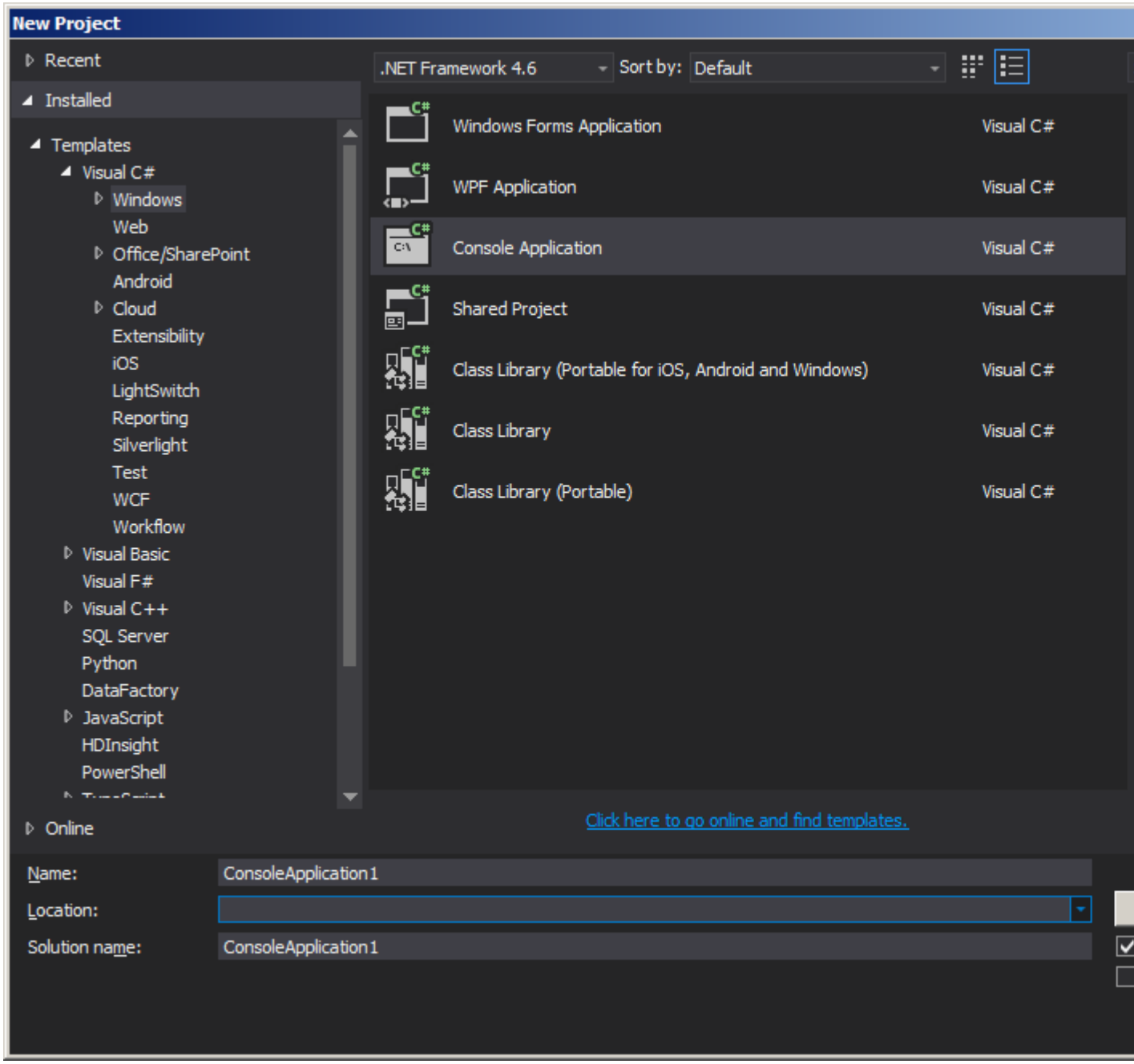
1. **Visual Studio**をダウンロードしてインストールします。 Visual StudioはVisualStudio.com からダウンロードできます 。 Community Editionは、はであることがされています.2のなが

まれており、さらにすることができるためです。

2. **Visual Studio**をきます。
3. ようこそ。 ファイル→ →プロジェクトにみます。

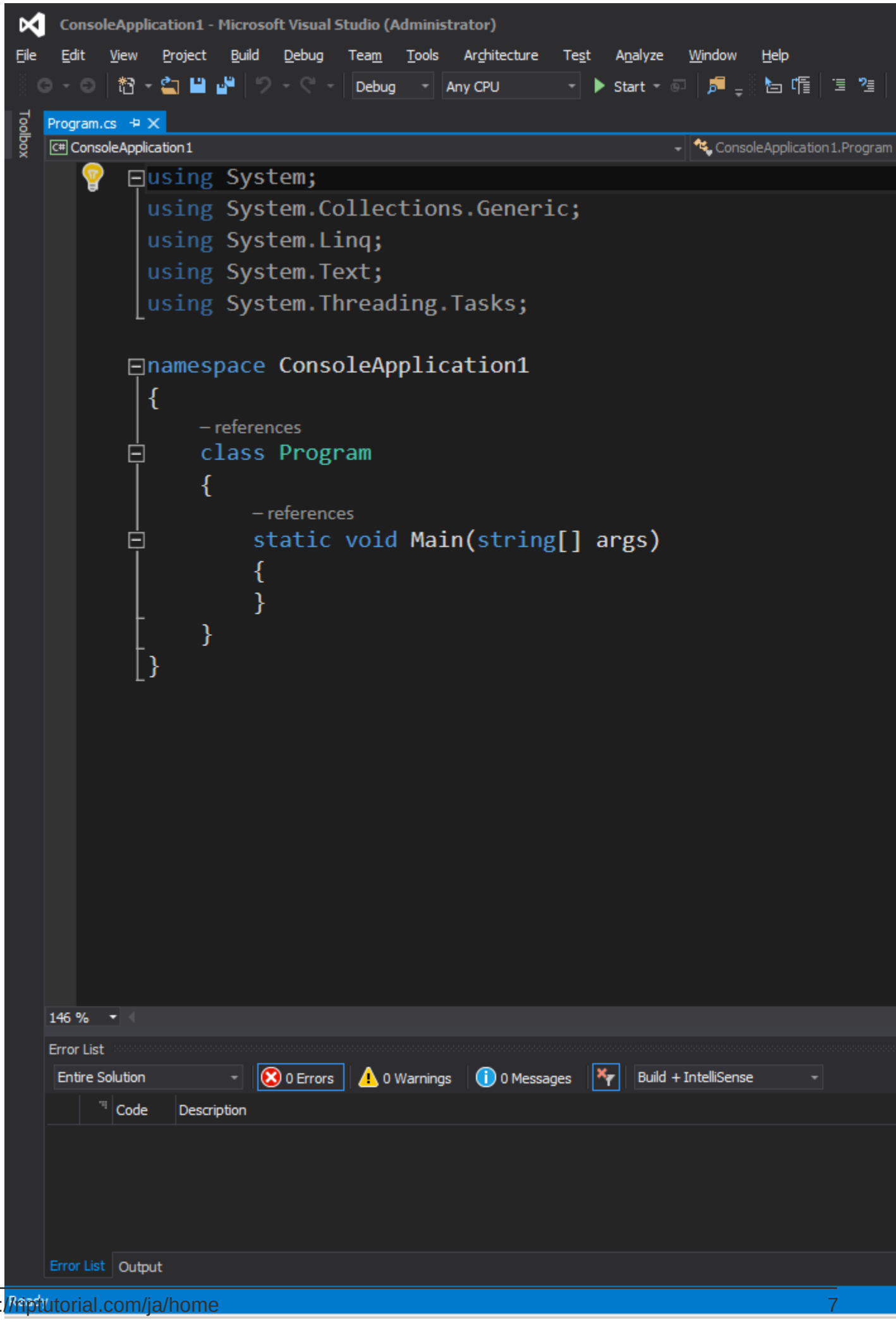


4. テンプレート → **Visual C** → コンソールアプリケーションをクリックします。



5. コンソールアプリケーションをした、プロジェクトのとするをし、OKをします。のをしないでください。

6. プロジェクトがされました。しくされたプロジェクトは、のようになります。


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

のプロジェクトとにできるように、プロジェクトにはわかりやすいをしてください。プロジェクトやクラスにはスペースをしないことをおめします。

プロジェクトやクラスにはスペースをしないことをおめします。

7. コードをく。 `Program.cs`をして「Hello world」をできるようになりました。ユーザにする。

```
using System;

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

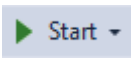
`Program.cs`の`public static void Main(string[] args)`オブジェクトにの2をしますのにあることをしてください

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

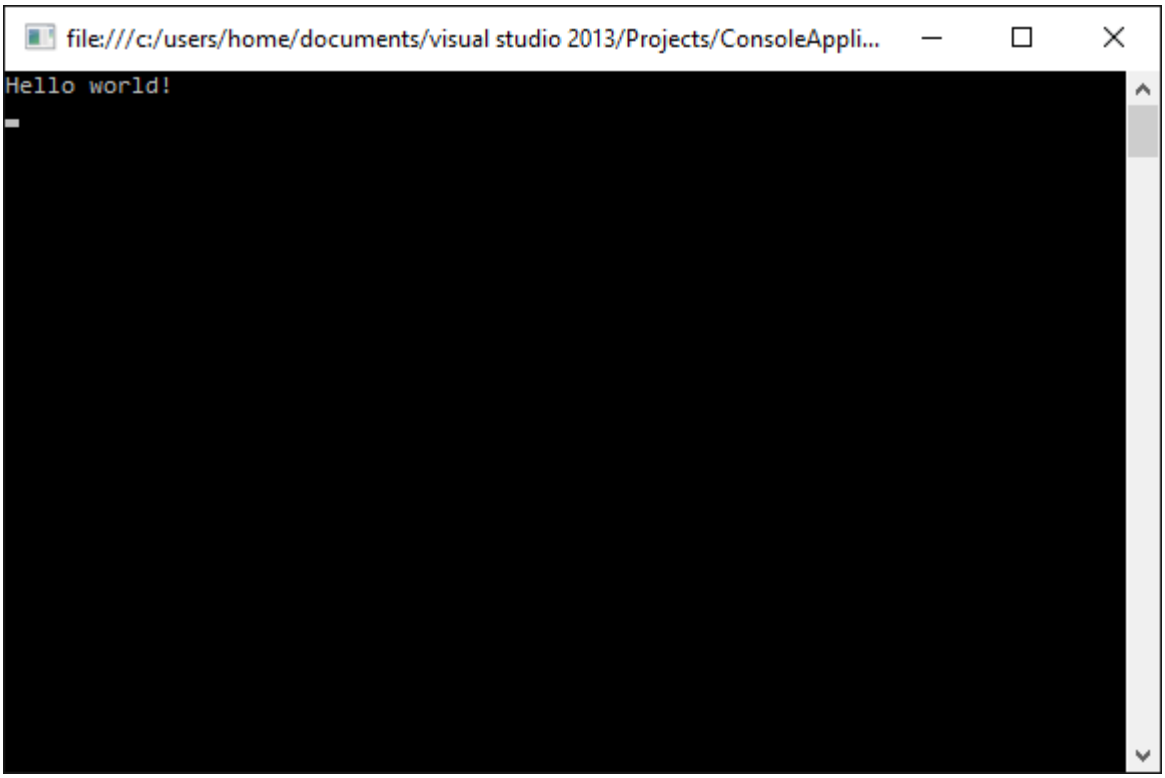
なぜ `Console.Read()` ですかのは "Hello world"というテキストをします。 2は1がされるのをちます。には、プログラムのをし、デバッグにをることができます。 `Console.Read();`なしで`Console.Read();`アプリケーションのデバッグをすると、に「Hello world」とされます。コンソールにし、すぐにじます。コードウィンドウはのようになります。

```
using System;

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

8. プログラムをデバッグします。ウィンドウのにあるツールバーのスタートボタンをします



キーボードで`F5`キーをしてアプリケーションをします。ボタンがないは、トップメニューからデバッグ→デバッグのをしてプログラムをできます。プログラムはコンパイルしてコンソールウィンドウをきます。のスクリーンショットのようにされます。



9. プログラムをします。プログラムをじるには、キーボードののキーをしてください。々がしたConsole.Read()は、このじのためのものでした。プログラムをじるもう1つのは、スタートボタンがあったメニューにき、▯▯ボタンをクリックすることです。

Monoをってしいプログラムをする

に、インストールセクションでしたように、したプラットフォームのインストールをして、Monoをインストールします。

MonoはMac OS X、Windows、Linuxでできます。

インストールがしたら、テキストファイルをし、HelloWorld.csというをけてのをコピーします。

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

Windowsをしているは、Monoインストールにまれているmonoコマンドプロンプトをし、ながされていることをします。 MacまたはLinuxのは、しいをきます。

しくしたファイルをコンパイルするには、HelloWorld.csをむディレクトリでのコマンドをします。

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

された`HelloWorld.exe`は、の`HelloWorld.exe`でできます。

```
mono HelloWorld.exe
```

これはをします

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

.NETコアをしてしいプロ グラムをする

に、したプラットフォームのインストールをして、 **.NET Core SDK**をインストールします 。

- Windows
- OSX
- Linux
- ドッカー

インストールがしたら、コマンドプロンプトまたはターミナルウィンドウをきます。

1. `mkdir hello_world`をしてしいディレクトリをし、 `mkdir hello_world`をしてしくしたディレクトリにし`cd hello_world`。

2. `dotnet new console` new consoleをしてしいコンソールアプリケーションをします。
   これは2つのファイルをします

   - **hello_world.csproj**

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

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

     </Project>
     ```

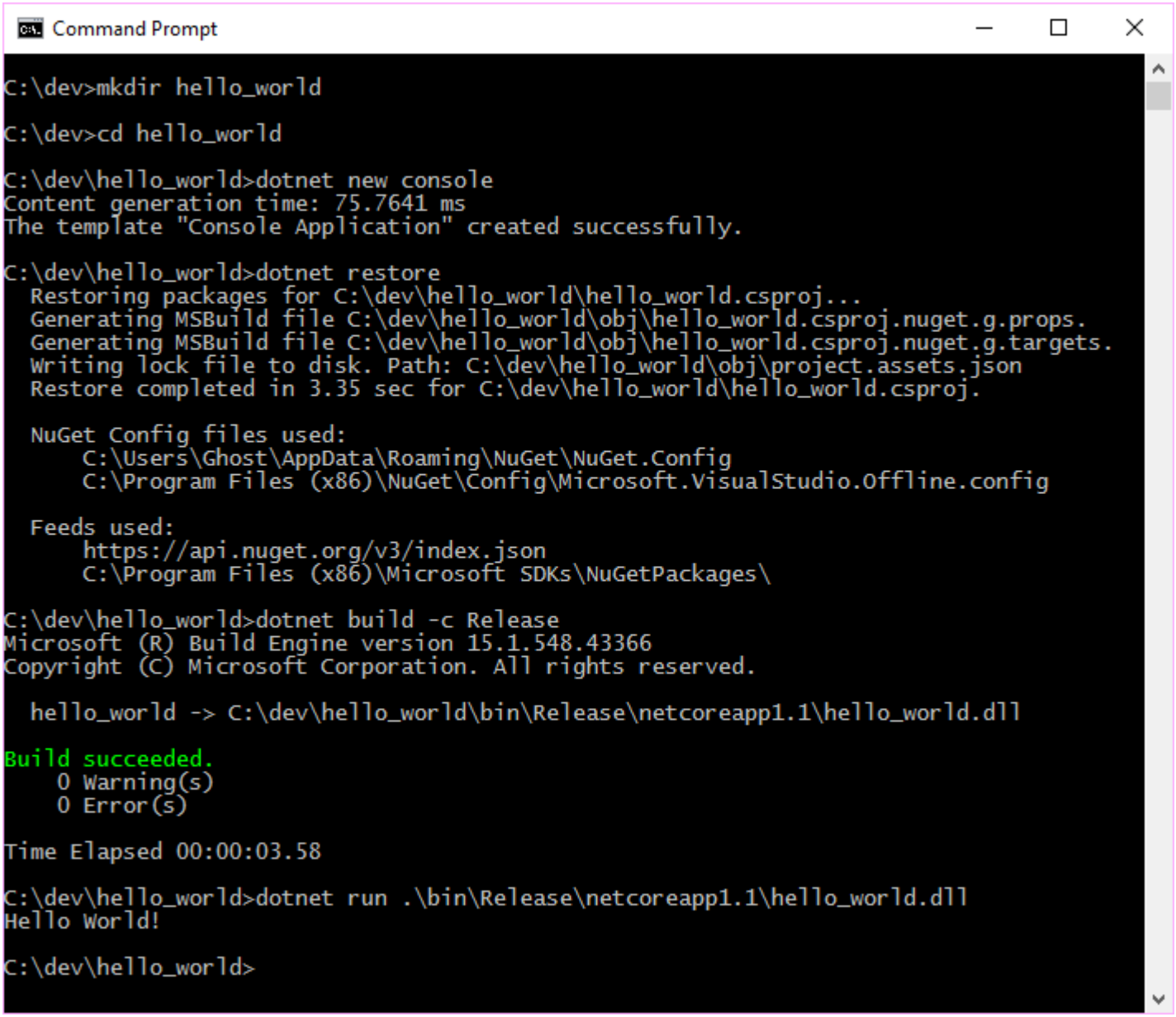
   - **Program.cs**

     ```
     using System;

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

```
        }
    }
}
```

3. `dotnet restore`してなパッケージを`dotnet restore`ます。
4. オプションでアプリケーションをビルド`dotnet build`、デバッグまたは`dotnet build -c Release`リリースのために。 `dotnet run`はコンパイラをし、ビルドエラーがあればそれをスローします。
5. デバッグまたは`dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` `dotnet run`ために`dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll`してアプリケーションをします`dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` for Release。

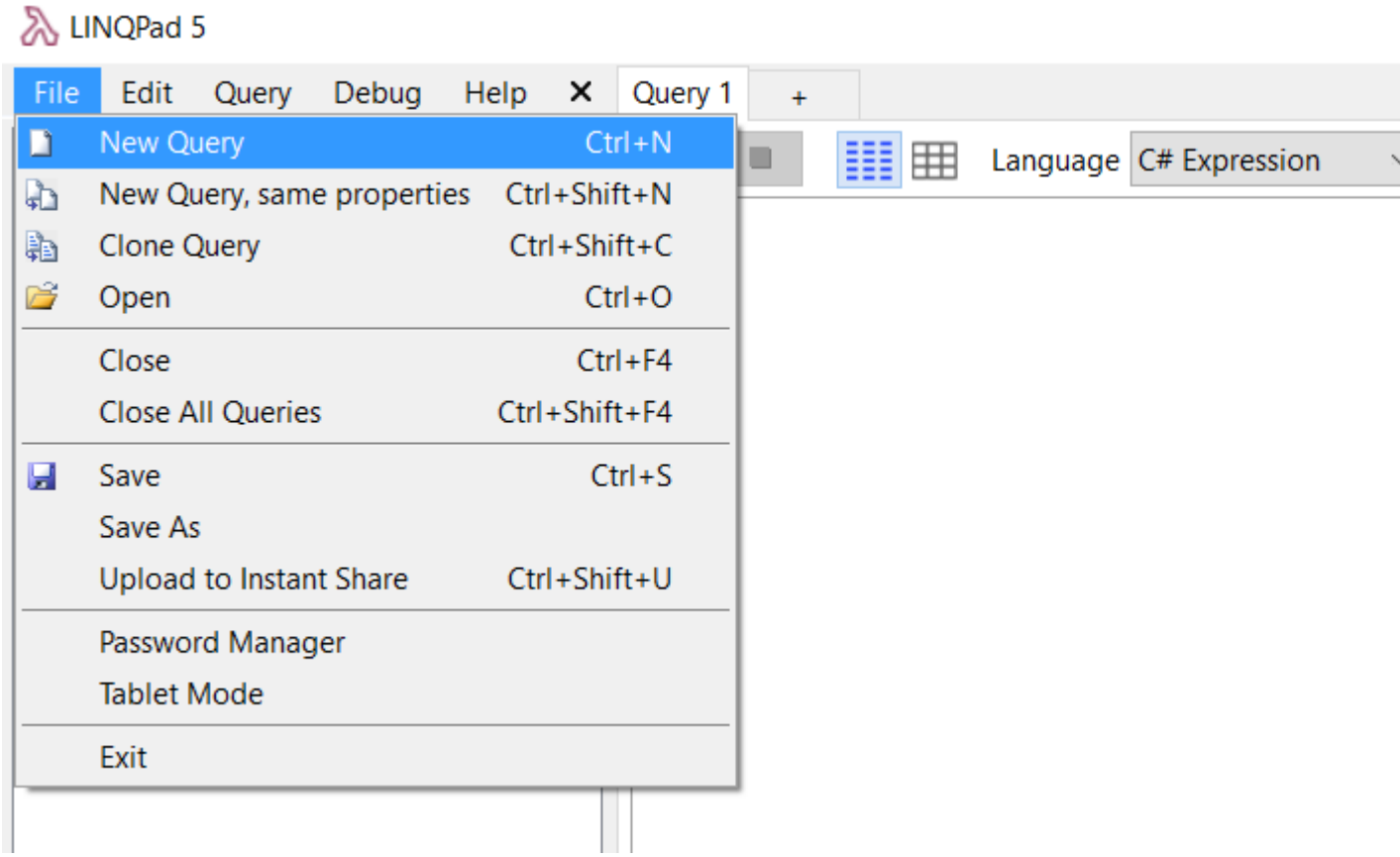
コマンドプロンプト



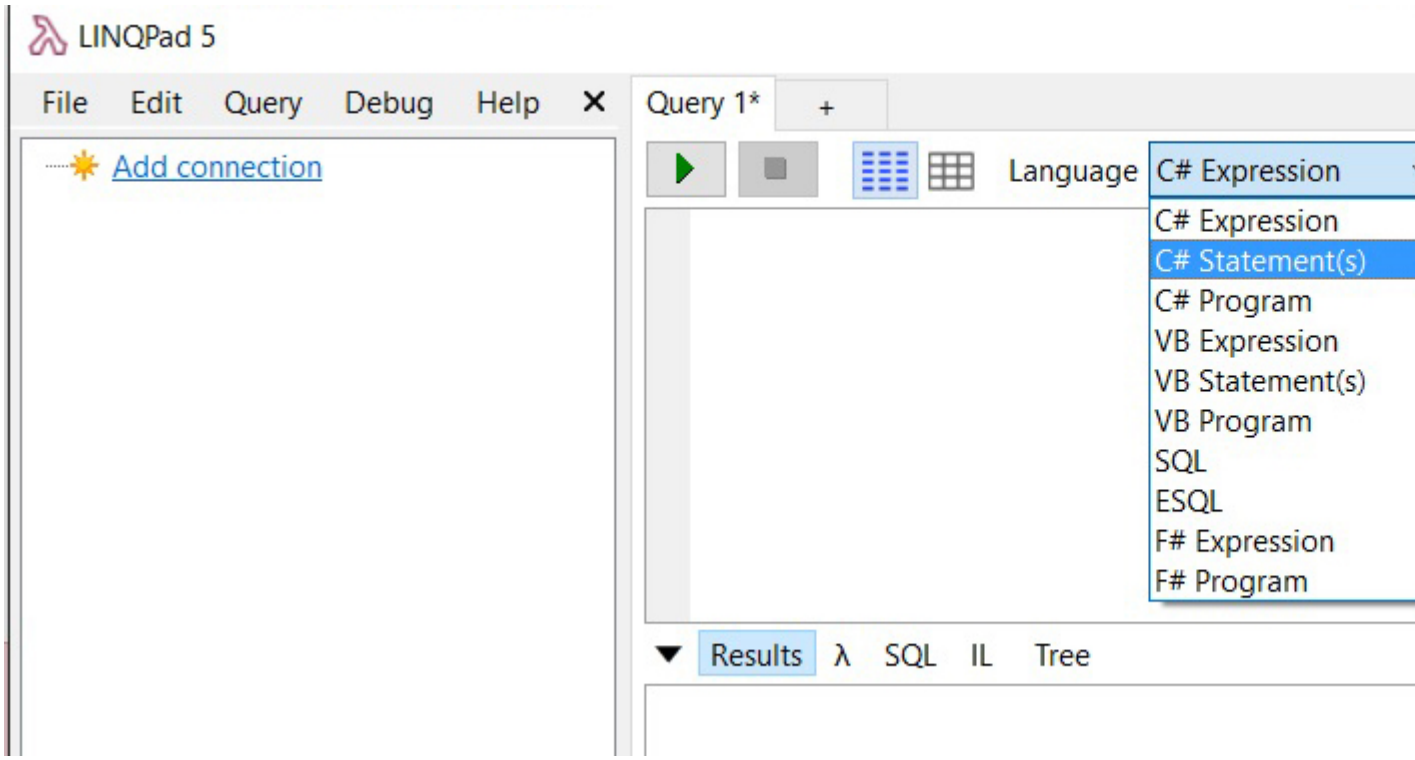
LinqPadをしてしいクエリをする

LinqPadは、.NetのC、F、VB.Netをびテストすることができるらしいツーールです。

1. LinqPadをインストーールする
2. しいクエリをする `Ctrl` + `N`

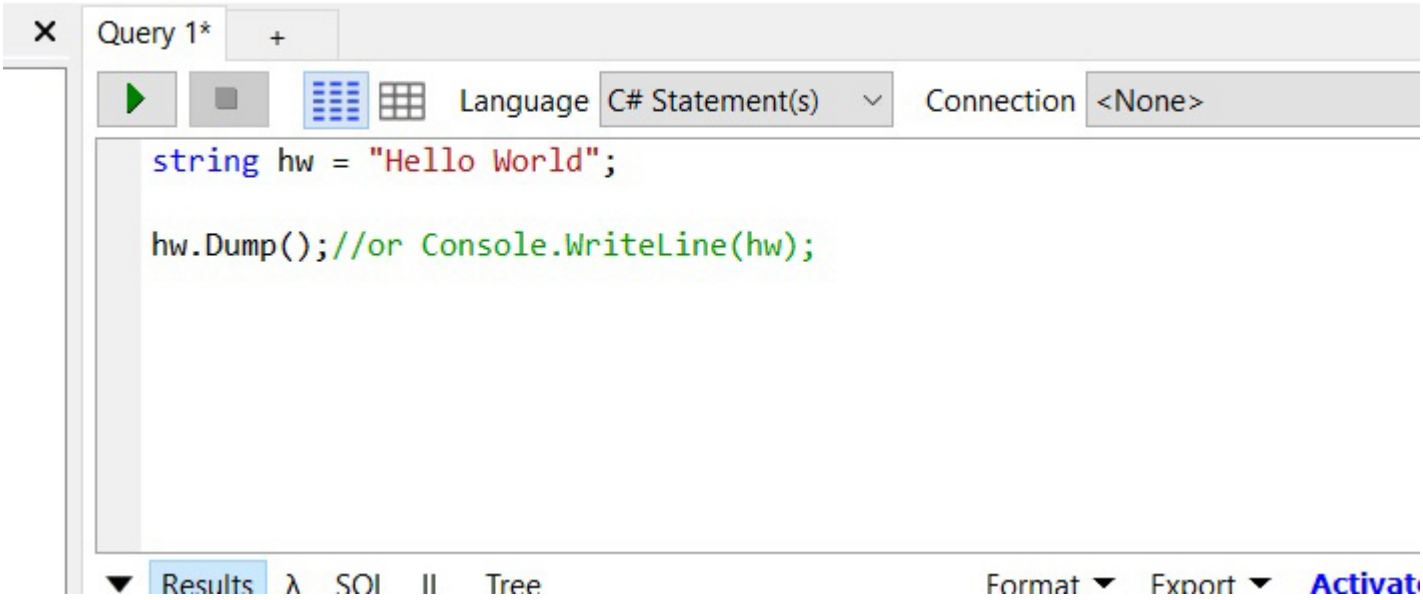


3. ので、「Cステーートメント」をします。

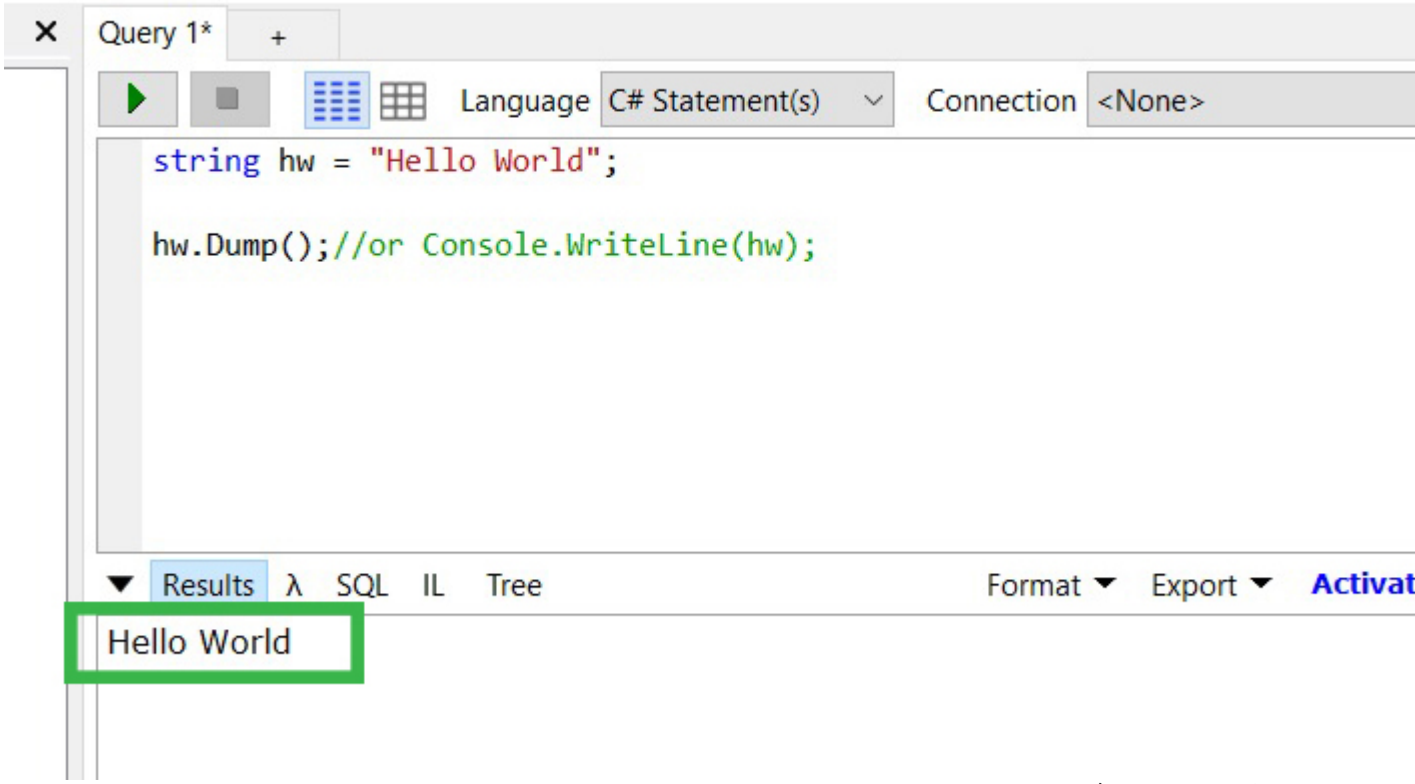


4. のコードをし、 F5 を□します。

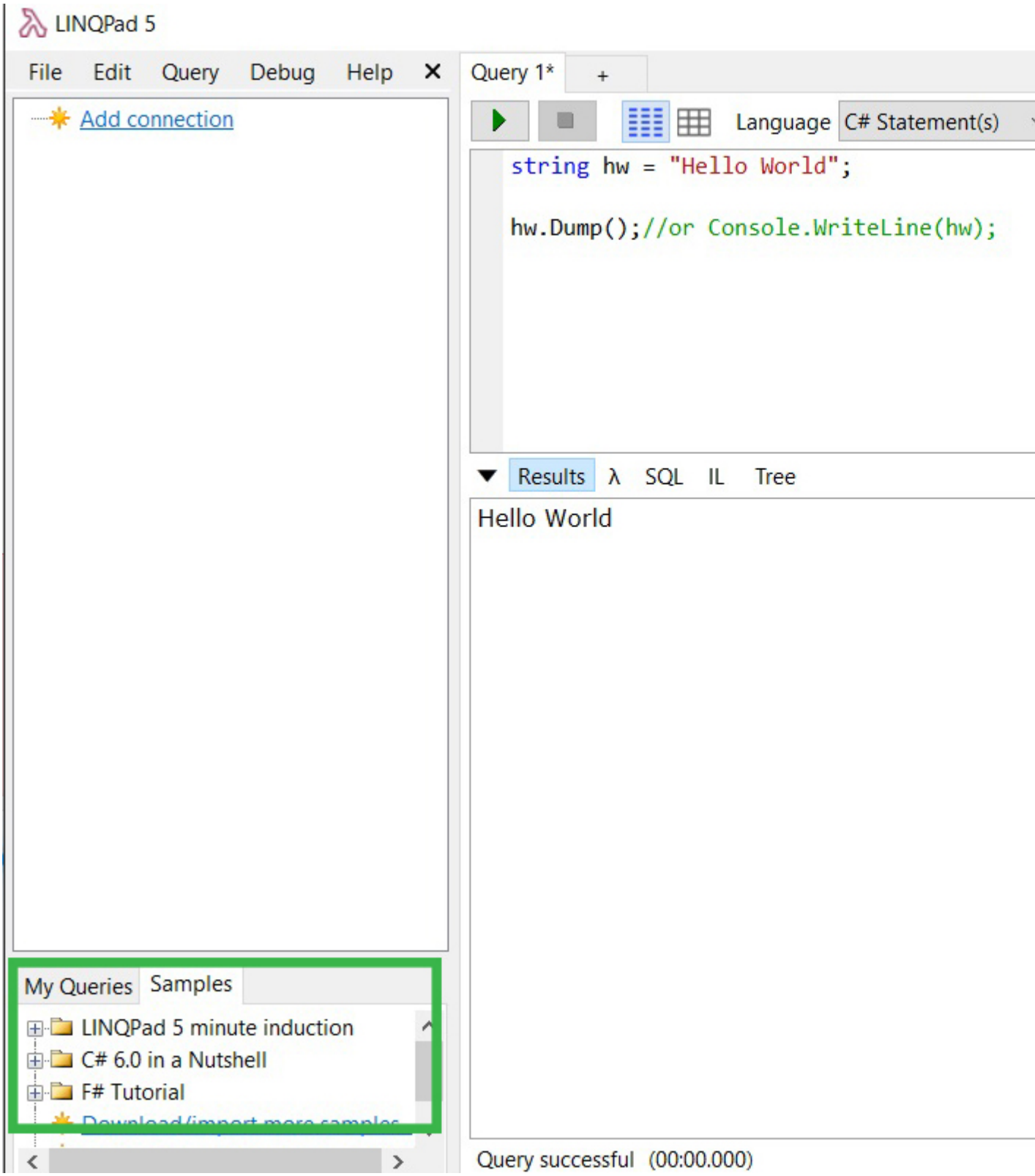
```
string hw = "Hello World";

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



5. に「Hello World」とされます。

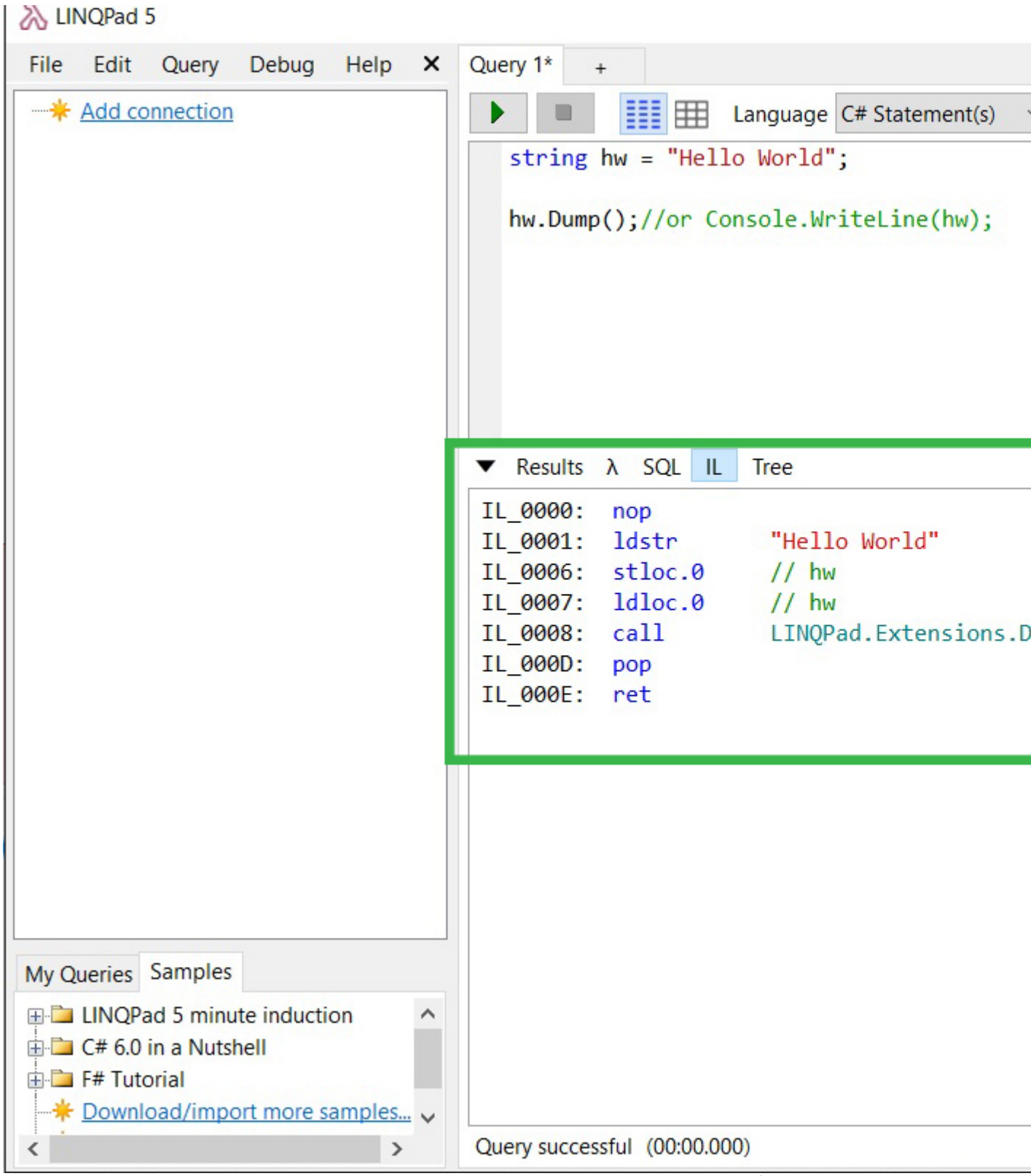


6. の.Netプログラムをしたので、LinqPadにまれているサンプルを "Samples"ブラウザでチェックアウトしてください。 .Netのさまざまなをすらしいがたくさんあります。



ノート

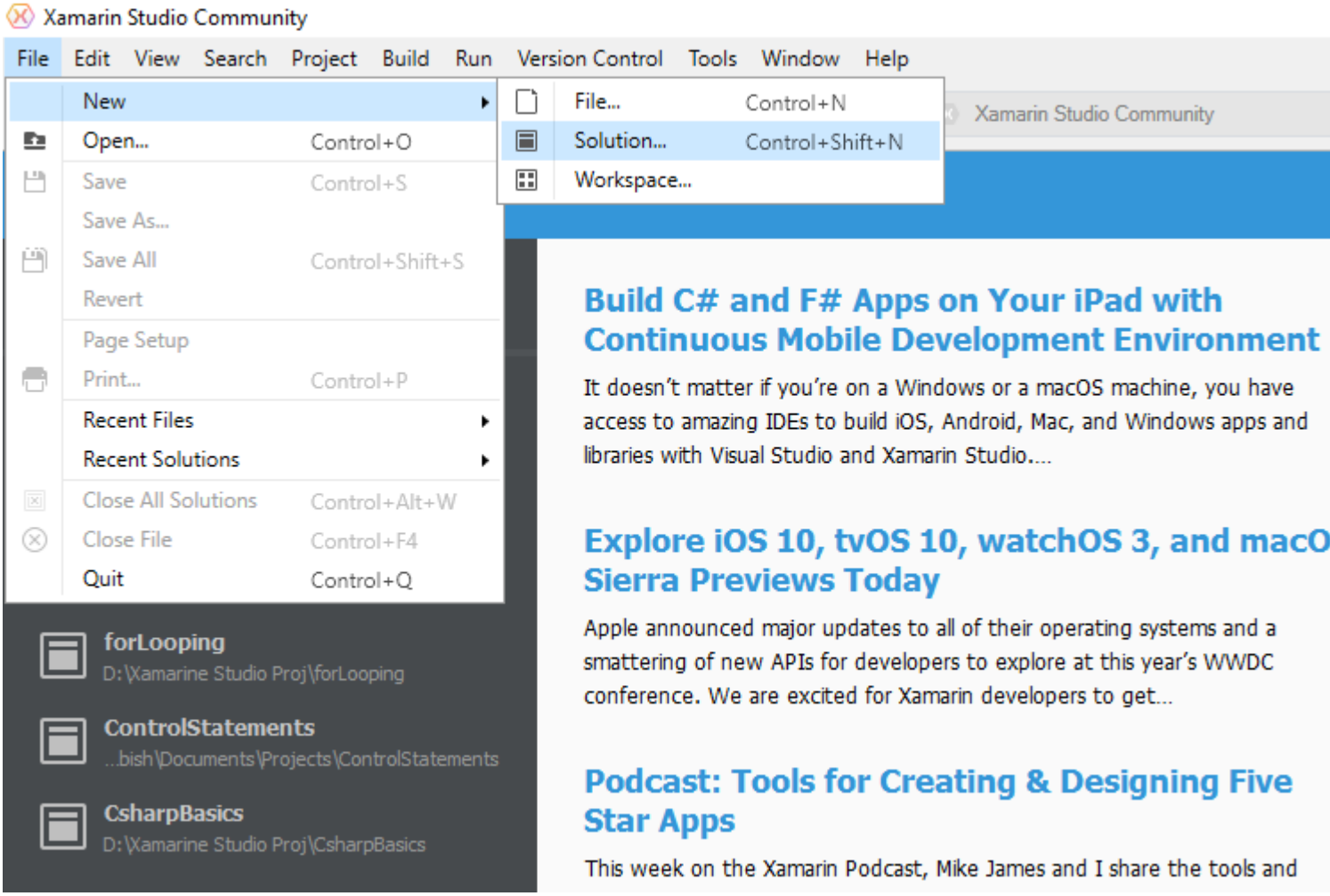
1. "IL"をクリックすると、.netコードがするILコードをべることができます。これはらしいツールです。



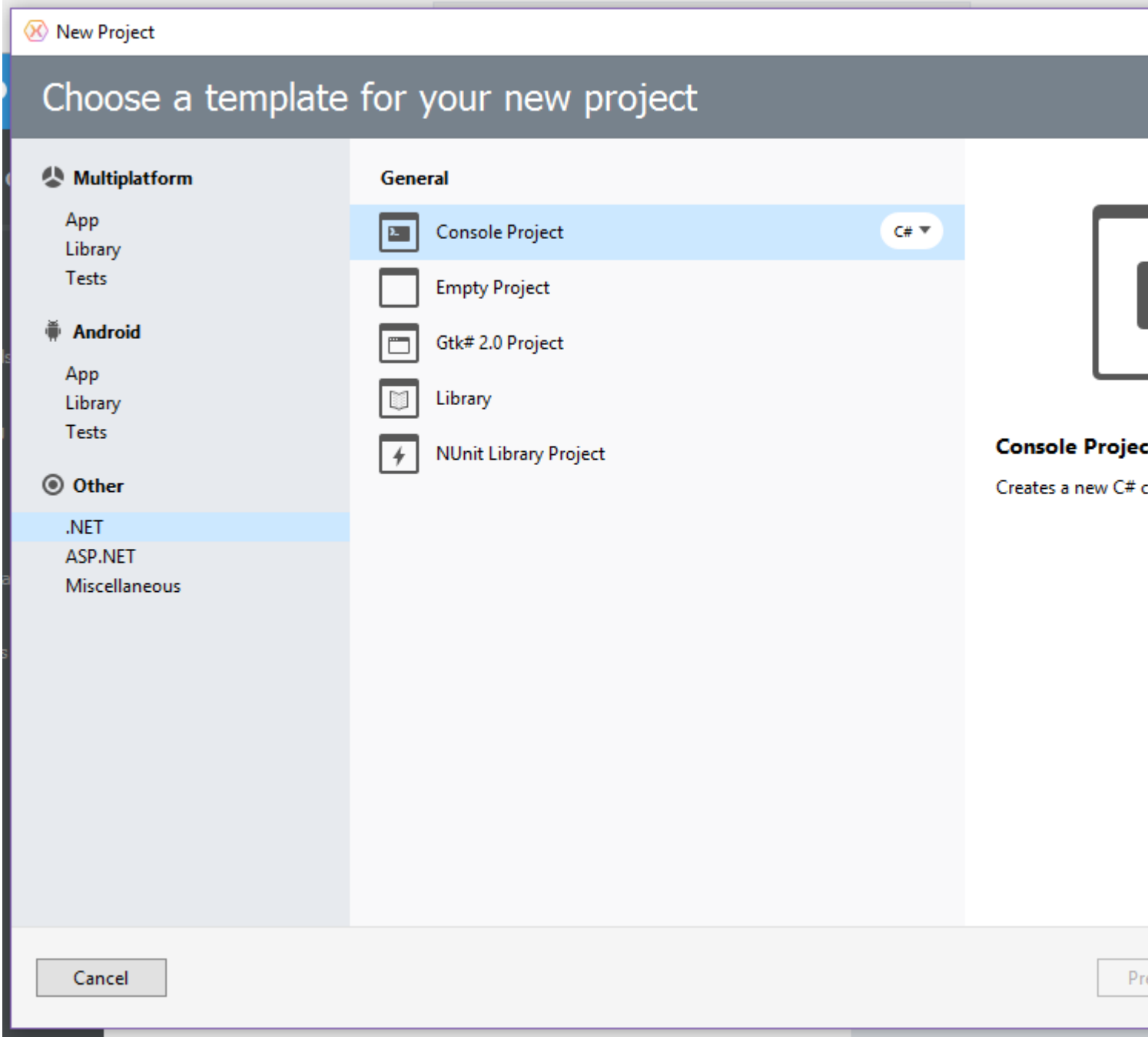
2. `LINQ to SQL`または`Linq to Entities`をすると、されているSQLをすることができます。これは、LINQについてふもう1つのれたです。

Xamarin Studioをしてしいプロジェクトをする

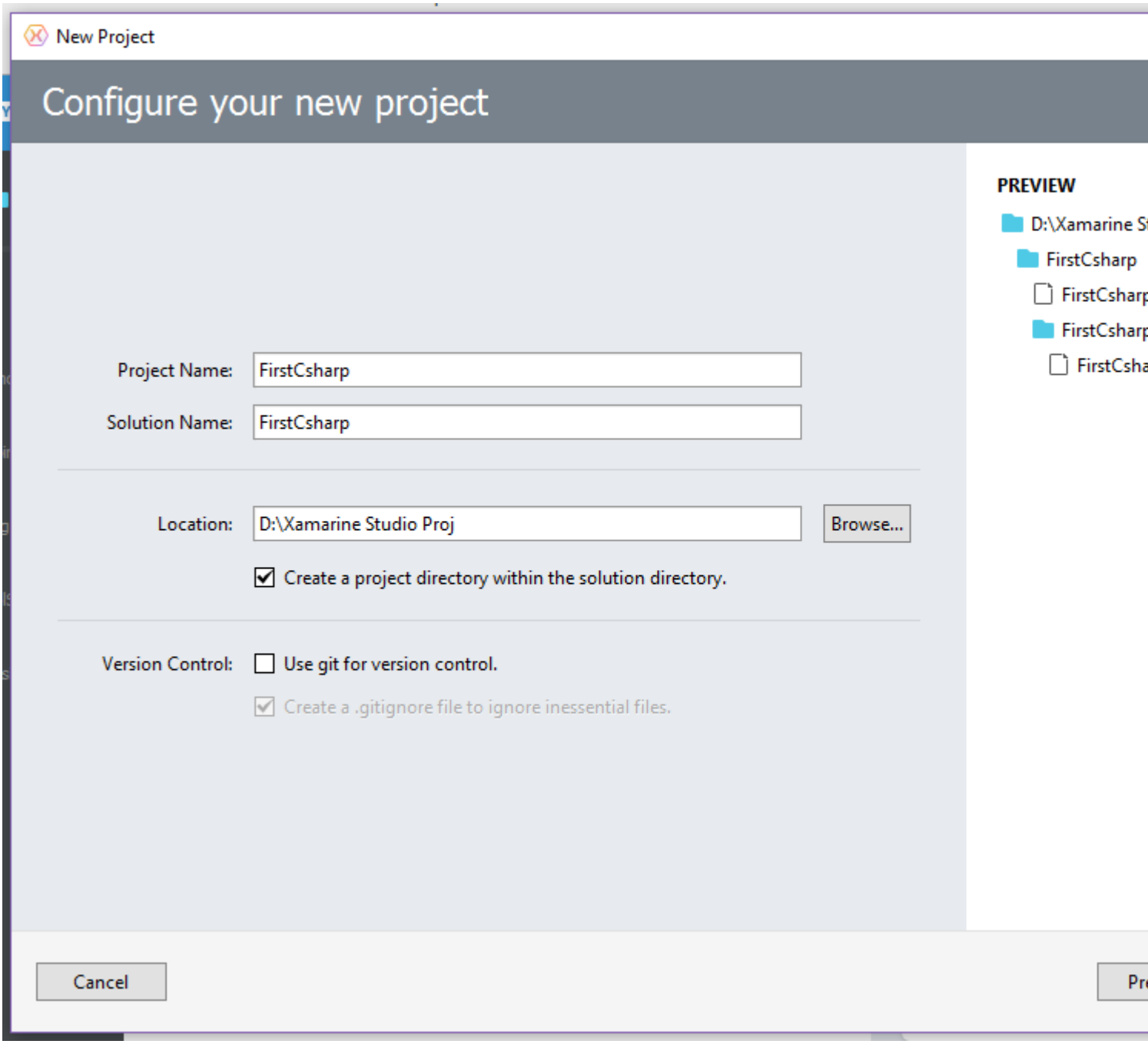
1. Xamarin Studio Communityをダウンロードしてインストールします。
2. Xamarin Studioをきます。
3. 「ファイル」→「」→「ソリューション」をクリックします。



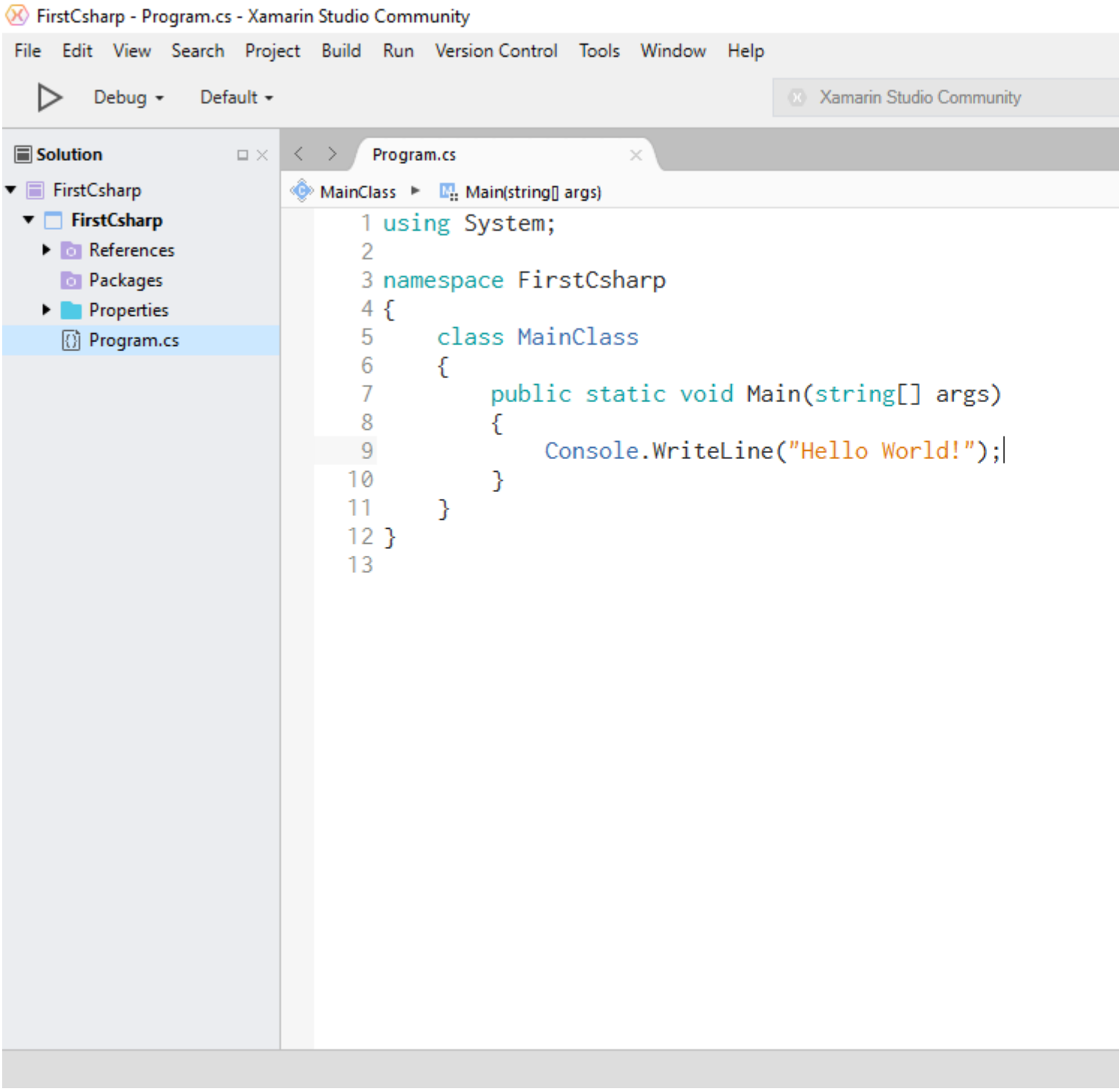
4. **.NET** → **Console Project**をクリックし、 **C**をします。
5. [へ]をクリックしてします



6. プロジェクトをし、するを...ブラウズし、[▯▯]をクリックします。



7. しくされたプロジェクトは、のようになります。

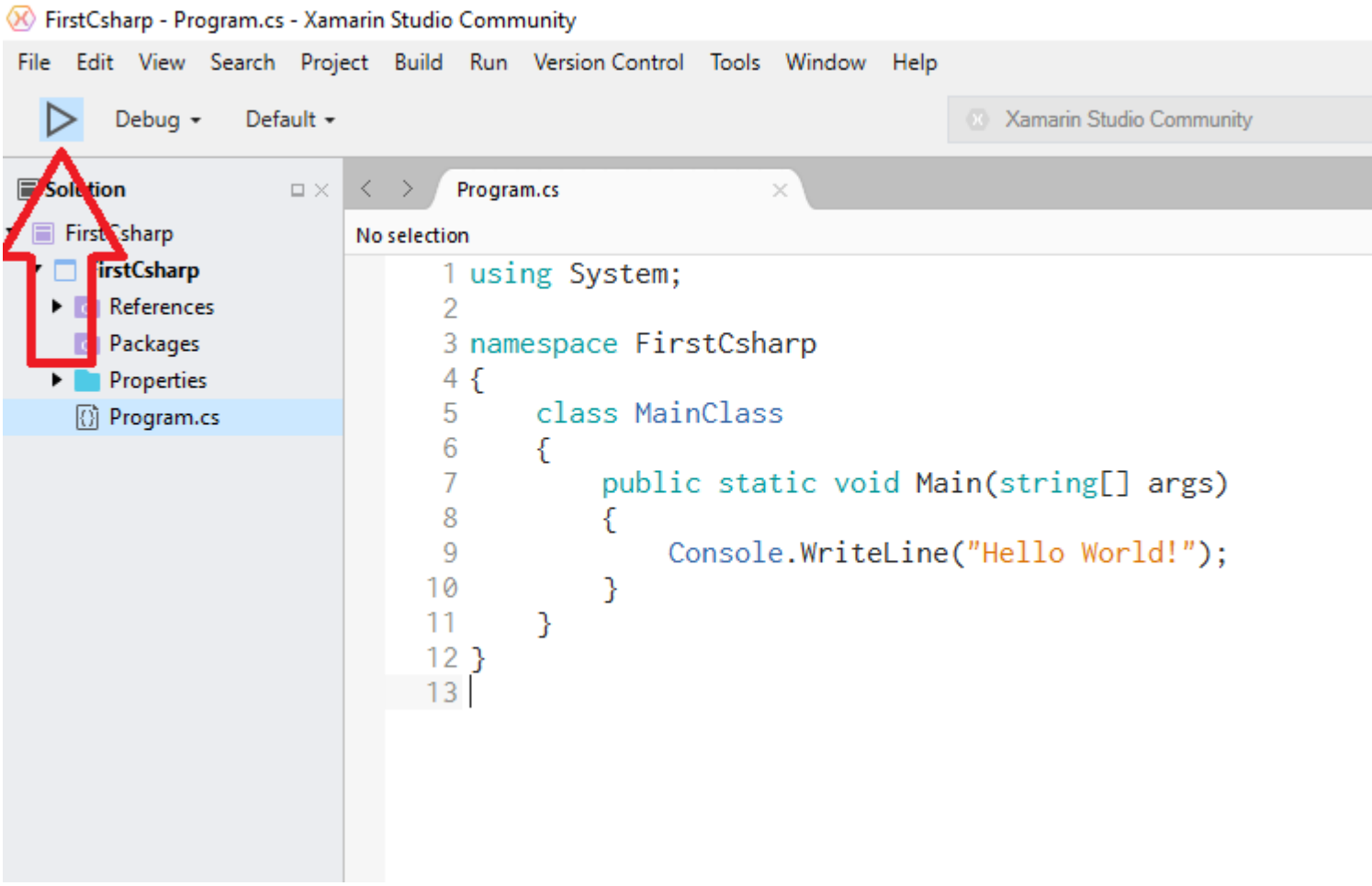


8. これはテキストエディタのコードです

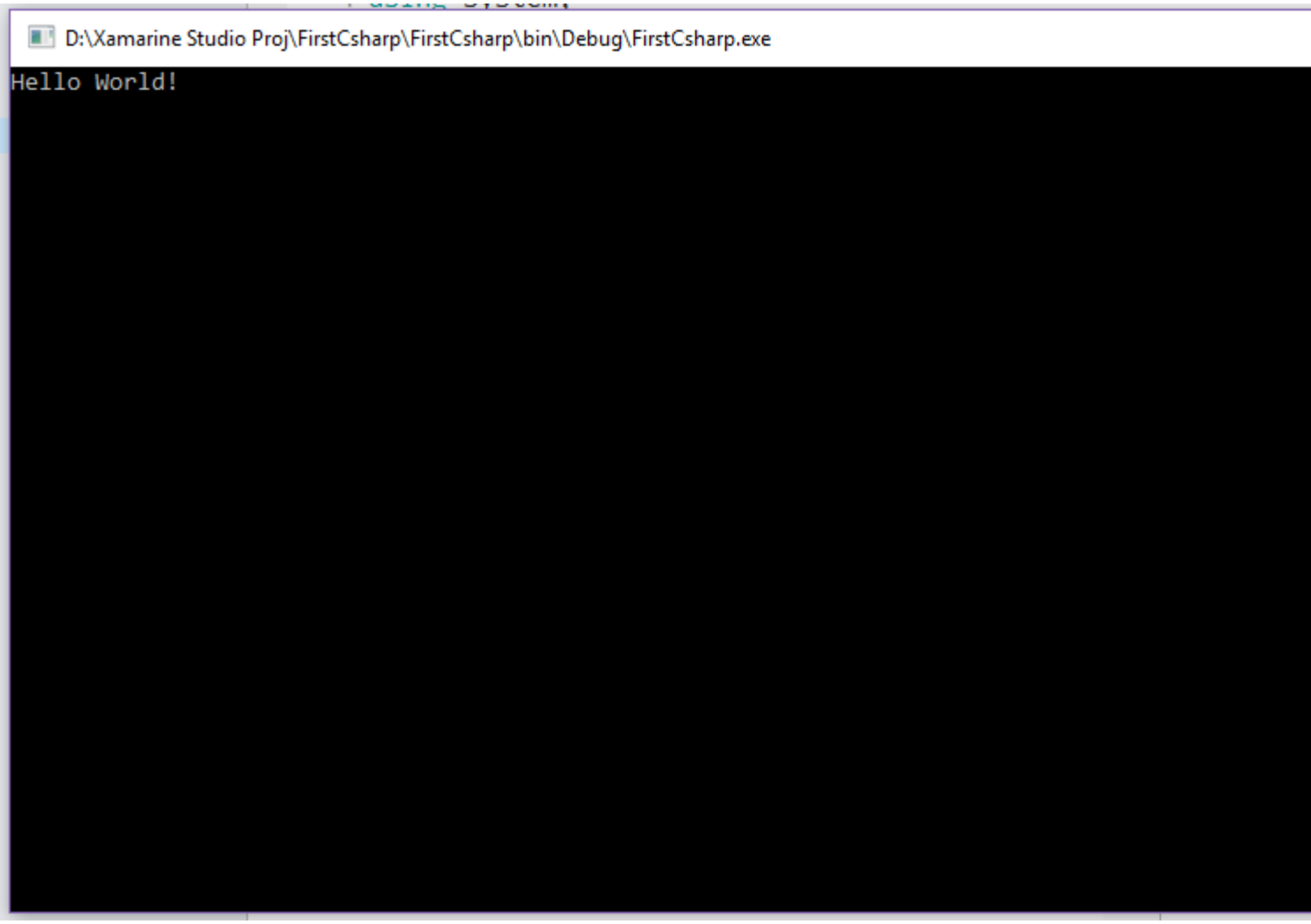
```
using System;

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

```
}
```

9. コードをするには、のようにF5キーをすか、 ボタンをクリックします。



10. はアウトプットです



オンラインでCをいめるをむ https://riptutorial.com/ja/csharp/topic/15/c-をいめる

2: .NETコンパイラプラットフォームRoslyn

Examples

MSBuildプロジェクトからワークスペースをする

するに、まず`Microsoft.CodeAnalysis.CSharp.Workspaces`ナゲットをしてください。

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

のコードをワークスペースにみむには、エラーをコンパイルしてします。その、コードはメモリにされます。ここからは、とのができるようになります。

シンタックスツリーは、、コマンド、およびマークのツリーとしてプログラムをすのデータですはエディタでされていました。

たとえば、 `compilation`というの`Microsoft.CodeAnalysis.Compilation`インスタンスがされているとします。ロードされたコードでされたすべてののをリストするはあります。このようににうには、すべてのドキュメント `DescendantNodes`メソッドのすべてのをり、Linqをしてをするノードをします。

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

するをつすべてののCがツリーにします。のタイプをすばやくつけるには、Visual Studioの`Syntax Visualizer`ウィンドウをします。これはいているをRoslynとしてします。

セマンティックモデル

セマンティックモデルは、ツリーとして、よりいレベルのとをします。がのをることができるところでは、モデルもとすべてのをえる。ツリーはメソッドびしをしますが、セマンティックモデルはメソッドがされたなをしますがされた。

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379)"
            }
        }
    }
}
```

これは、をしてローカルのリストをします。に、セマンティックモデルをしてなをし、すべてのすべてのをします。

オンラインで.NETコンパイラプラットフォームRoslynをむ
https://riptutorial.com/ja/csharp/topic/4886/-netコンパイラプラットフォーム-roslyn-

3: .NETのガベージコレクタ

Examples

ラージオブジェクトヒープ

デフォルトでは、ラージオブジェクトヒープは、メモリのにつながるのあるのオブジェクトヒープとはなり、 `OutOfMemoryException` つながるがあります

.NET 4.5.1、ラージオブジェクトヒープをガベージコレクションとににするオプションがあります 。

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

なガベージコレクションCLRがにされないためリクエストとばれますとじようにしてしてください。デフォルトでGCのをデキャリブレートしてパフォーマンスをさせるがあるのでけてください。

い

.NETでは、GCはがっていないときにオブジェクトをりてます。したがって、コードからオブジェクトにすることはできますがいがあります、GCはこのオブジェクトをりてません。きなオブジェクトがたくさんある、これはになるがあります。

ウィークリファレンスはであり、オブジェクトにアクセスできるようにしながらGCがオブジェクトをできるようにします。は、いがしないときにオブジェクトがされるまでなにのみです。ウィークリファレンスをすると、アプリケーションはききオブジェクトへのいをすることができ、オブジェクトのがげられます。したがって、いは、するのにがかかりますが、にされていないはガベージコレクションのためにできるはずのきなオブジェクトをするのにです。

ない

```
WeakReference reference = new WeakReference(new object(), false);

GC.Collect();

object target = reference.Target;
if (target != null)
  DoSomething(target);
```

したがって、えばオブジェクトのキャッシュをするために、いをすることができます。ただし、いがされるにガベージコレクタがオブジェクトにするリスクはにすることをえておくことがです。

いはメモリリークをするのにもです。なユースケースはイベントです。

ソースのイベントにするハンドラがあるとします。

```
Source.Event += new EventHandler(Handler)
```

このコードはイベントハンドラをし、イベントソースからリスニングオブジェクトへのなをします。ソースオブジェクトがリスナーよりもがく、リスナーにのがないときにリスナーがイベントをとしない、の.NETイベントをするとメモリリークがします。ソースオブジェクトはメモリのリスナーオブジェクトをします。ガベージコレクションするがあります。

この、 いイベントパターンをすることをおめします。

かのようなもの

```
public static class WeakEventManager
    {
    public static void SetHandler<S, TArgs>(
    Action<EventHandler<TArgs>> add,
    Action<EventHandler<TArgs>> remove,
    S subscriber,
    Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
        {
            var subscrWeakRef = new WeakReference(subscriber);
            EventHandler<TArgs> handler = null;

            handler = (s, e) =>
            {
                var subscrStrongRef = subscrWeakRef.Target as S;
                if (subscrStrongRef != null)
                {
                    action(subscrStrongRef, e);
                }
                else
                {
                    remove(handler);
                    handler = null;
                }
            };

            add(handler);
        }
    }
```

このようにされます

```
 EventSource s = new EventSource();
 Subscriber subscriber = new Subscriber();
 WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

この、いくつかのがあります。イベントは

```
public event EventHandler<SomeEventArgs> Event;
```

MSDNがするように

- オブジェクトのはファイナライズにできないので、にじていいをしてください。
- ポインタがきくてもきくてもかまいませんので、さなオブジェクトへのはけてください。
- メモリのにするとしてをしないでください。わりに、アプリケーションのオブジェクトをするためのなキャッシング·ポリシーをします。

オンラインで.NETのガベージコレクタをむ https://riptutorial.com/ja/csharp/topic/1287/-netのガベージコレクタ

4: .NETのでないコード

- .Netプロジェクトで`unsafe`キーワードをできるようにするには、Project Properties => Buildで「でないコードをする」チェックボックスをオンにするがあります
- でないコードをすると、パフォーマンスがするがありますが、コードのをにしていましたがって、 `unsafe`という。

たとえば、forループをすると、のようながされます。

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

.NET Frameworkでは、のをえないように、インデックスがをえたに`IndexOutOfRangeException`します。

しかし、でないコードをすると、のをえてしまうことがあります

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

Examples

でないインデックス

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

このコードをするとさ3のがされますが、に5のインデックス4をしようとします。のマシンでは、これは`1910457872`し`1910457872`が、そのはされていません。

`unsafe`ブロックがなければ、ポインタをすることはできません。そのため、のをえてにアクセスすることはできません。がスロ―されることはありません。

でないの

ポインタでにアクセスすると、チェックがないため、 `IndexOutOfRangeException`は`IndexOutOfRangeException`されません。これにより、コ―ドがになります。

ポインタをってにをする

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

でなはのようになります。

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

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

でないはにで、のはアレイののさと、それぞれのロジックにされるロジックによってなります。よりいかもしれませんが、することはしく、れやすいので、してするがあります。

でないをう

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

unsafe                // allows writing to memory; methods on System.String don't allow this
```

```
{
  fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';     // change data in memory allocated for original string "Hello"
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                      // value in memory, but the contents at that location were
                      // changed by the unsafe write above.
                      // Displays: "aaaaa"
```

オンラインで.NETのでないコードをむ https://riptutorial.com/ja/csharp/topic/81/-netのでないコード

5: .zipファイルのみき

1. public static ZipArchive OpenReadストリングarchiveFileName

パラメーター

パラメータ	
archiveFileName	オープンするアーカイブへのパス。パスまたはパスでします。パスはのディレクトリからのパスとしてされます。

Examples

zipファイルへのきみ

しい.zipファイルをきむには

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("========================");
        }
    }
}
```

Zipファイルをメモリにきむ

のでは、ファイルシステムにアクセスすることなく、されたファイルをむzipファイルの`byte[]`データをします。

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file
with this name
```

```
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
        }
        //ZipArchive must be disposed before the MemoryStream has data
        return ms.ToArray();
    }
}
```

Zipファイルからファイルをする

このでは、されたzipアーカイブバイナリデータからファイルのリストをします。

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}
```

のは、zipアーカイブをいて、すべての.txtファイルをフォルダにするをしています

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
```

```
                }
            }
        }
    }
```

オンラインで.zipファイルのみきをむ https://riptutorial.com/ja/csharp/topic/6709/-zipファイルのみき

6: ASP.NETアイデンティティ

き

ユーザー、、トークンのなど、asp.net IDにするチュートリアル

Examples

ユーザーマネージャをしてasp.net IDにパスワードリセットトークンをする。

1. MyClassesというのしいフォルダをし、のクラスをしてします

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. IDクラスの

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com",
"(any subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. web.configにをします。このでGmailをしなかったのは、Gmailのがのでブロックされており、それでもにしているからです。

```
<add key="GmailUserName" value="youremail@yourdomain.com"/>
```

```
<add key="GmailPassword" value="yourPassword"/>
<add key="GmailHost" value="yourServer"/>
<add key="GmailPort" value="yourPort"/>
<add key="GmailSsl" value="chooseTrueOrFalse"/>
<!--Smptp Server (confirmations emails)-->
```

4. アカウントコントロ―ラになをえます。のされたコ―ドをします。

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();


        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext
        {
            _context = new ApplicationDbContext();

        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signIn
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));

        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
```

```
//
// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {

        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicki
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

コンパイルしてからします。

オンラインでASP.NETアイデンティティをむ https://riptutorial.com/ja/csharp/topic/9577/asp-net-アイデンティティ

7: AssemblyInfo.csの

ファイル`AssemblyInfo.cs`は、ソースファイルとしてにされ、は、のアセンブリをするメタデータをします。

Examples

[AssemblyTitle]

このは、こののアセンブリにをけるためにされます。

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

[アセンブリ]

このは、こののアセンブリがとするをするためにされます。のアセンブリはじのコンポーネントになることがあります。その、すべてのコンポーネントはこのにしてじをできます。

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

グローバルおよびローカルAssemblyInfo

グローバルでよりいDRYnessがられるようにするには、があるプロジェクトにしてAssemblyInfo.csとなるをれるだけでみます。このは、にのビジュアルスタジオプロジェクトがあることをとしています。

GlobalAssemblyInfo.cs

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

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

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

// Version information for an assembly consists of the following four values:
// roughly translated from I reckon it is for SO, note that they most likely
// dynamically generate this file
//      Major Version  - Year 6 being 2016
//      Minor Version  - The month
```

```
//      Day Number     - Day of month
//      Revision       - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - プロジェクトごとに1つ

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

GlobalAssemblyInfo.csをローカルプロジェクトにするには、 のをします 。

1. プロジェクトのコンテキストメニューで[Add / Existing Item ...]をします。
2. GlobalAssemblyInfo.csをします。
3. のさなきのをクリックして、Add-Buttonをします。
4. ボタンのドロップダウンリストで[リンクとして]をします

[AssemblyVersion]

このは、アセンブリにバージョンをします。

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

*は、コンパイルするたびににバージョンのをインクリメントするためにされますしばしば "ビルド"のためにされます

アセンブリのみり

.NETのなリフレクションAPIをすると、アセンブリのメタデータにアクセスできます。たとえば、のコードをしてthisアセンブリのtitleをできます

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

...

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

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

バージョン

ソースのコードには、デフォルトでSVN idsまたはGit SHA1ハッシュまたはにGitタグバージョンがあります。 AssemblyInfo.csのバージョンをでするのではなく、ビルドのプロセスをして、ソースシステムからAssemblyInfo.csファイルに、つまりアセンブリにバージョンをきむことができま

す。

GitVersionTaskまたはSemVer.Git.Fody NuGetパッケージは、のです。たとえば、GitVersionTask をするには、プロジェクトにパッケージをインストールした、AssemblyInfo.csファイルから `Assembly*Version`をします。これにより、GitVersionTaskがアセンブリのバージョンをします。

セマンティックバージョニングはますますデファクトスタンダードになっているので、これらのメソッドはSemVerにくソースコントロールタグのをしています。

のフィールド

AssemblyInfoのデフォルトフィールドをさせることをおめします。はインストーラによってされ、プログラムとWindows 10をしてプログラムをアンインストールまたはするときにされます。

はのとおりです。

- AssemblyTitle - は、 つまり MyCompany.MySolution.MyProject
- AssemblyCompany - フルネーム
- AssemblyProduct - マーケティングはここでることができます
- AssemblyCopyright - そうでなければなようにえるようにのにつ

DLLのProperties Detailsタブをべるとき、 'AssemblyTitle'は 'File description'になります。

[AssemblyConfiguration]

AssemblyConfigurationAssemblyConfigurationには、アセンブリのにされたがです。きコンパイルをして、さまざまなアセンブリをにみみます。ののようなブロックをしてください。よくうをいくつでもできます。

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

[InternalsVisibleTo]

アセンブリの`internal`クラスまたはをのアセンブリからアクセスにするは、これを、`InternalsVisibleTo`およびアクセスがされているアセンブリでします。

、このコードでは`MyAssembly.UnitTests`びすことがされている`internal`からのを`MyAssembly`。

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

これは、な`public`をぐためのテストににです。

[AssemblyKeyFile]

たちのアセンブリをGACにインストーールしたいときはいつでも、なをけるがあります。なアセンブリのためには、をするがあります。 `.snk`ファイルをします。

なのキーファイルをするには

1. VS2015のコマンドプロンプトアクセスき
2. コマンドプロンプトでcd C\ Directory_Nameとし、Enterキーをします。
3. コマンドプロンプトで、sn -k KeyFileName.snkとし、Enterキーをします。

keyFileName.snkがされたディレクトリにされたら、プロジェクトでrefernceをえます。`AssemblyKeyFileAttribute`に`snk`ファイルへのパスをえ、クラスライブラリをするときにキーをします。

プロパティ - > AssemblyInfo.cs

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

ティは、ビルドになのアセンブリをします。あなたのなのアセンブリをした、それをGACにインストーールすることができます

ハッピーコーディング:)

オンラインでAssemblyInfo.csのをむ https://riptutorial.com/ja/csharp/topic/4264/assemblyinfo-csの

8: Async-Awaitのコンテキスト

Examples

async / awaitキーワードのコード

なメソッドをえてみましょう

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

すると、このコードはにはのことをします。

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

これは、 `async` / `await`キーワードがする、のコンテキストをすることをします。つまり、UI、Web、およびConsoleアプリケーションでしくするライブラリコードをできます。

ソース 。

コンテキストをにする

コンテキストをにするには、 `ConfigureAwait`メソッドをびすがあります。

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

. . .

Foo().ConfigureAwait(false);
```

ConfigureAwaitは、デフォルトのSynchronizationContextをするをします。flowContextパラメータにfalseをすと、したにSynchronizationContextをしてをできなくなります。

それはSynchronizationContextにするすべてです。

なぜSynchronizationContextがなのでしょうか

このをえてみましょう。

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

このメソッドは、 `RunTooLong`がするまでUIアプリケーションをフリーズします。アプリケーションがしなくなります。

コードをでることができます

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

しかし、このコードはボディがUIスレッドでされるがあり、UIプロパティをしてはならないため、このコードはされません。

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            lable1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

このパターンをうことをにれないでください。または、 `SynchronizationContext.Post`をしてみてください。

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
```

```
        }, null);
    });
}
```

オンラインでAsync-Awaitのコンテキストをむ https://riptutorial.com/ja/csharp/topic/7381/async-awaitのコンテキスト

9: BackgroundWorker

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

UIスレッドでされるをすると、アプリケーションがしなくなり、ユーザーがをしたようにえることがあります。これらのタスクはバックグラウンドスレッドですることがましいすると、UIをすることができます。

BackgroundWorkerのにUIをするには、UIスレッドにをびすがあります。は、するコントロールでControl.Invokeメソッドをします。そうしないと、プログラムはをスローします。

BackgroundWorkerは、、Windowsフォームアプリケーションでのみされます。 WPFアプリケーションでは、 タスクをバックグラウンドスレッドおそらくはasync / awaitとみわせて にオフロードするためにタスクがされます。 UIスレッドへののマーシャリングは、、されるプロパティがINotifyPropertyChangedをするときににされるか、UIスレッドのDispatcherをしてでわれます。

Examples

BackgroundWorkerへのイベントハンドラのりて

BackgroundWorkerのインスタンスがされたら、それがするタスクのプロパティとイベントハンドラをえなければなりません。

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

    bgWorker.DoWork += bgWorker_DoWork;


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

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

    bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

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

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

     bgWorker.ProgressChanged += bgWorker_ProgressChanged;

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

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

BackgroundWorkerへのプロパティのりて

これにより、BackgroundWorkerをタスクでキャンセルすることができます

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

これにより、はタスクののにをすることができます...

```
bgWorker.WorkerReportsProgress = true;

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

しいBackgroundWorkerインスタンスをする

BackgroundWorkerは、UIスレッドをブロックすることなく、タスクをするためによくわれ、がかかることがあります。

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

namespace BGWorkerExample
{
     public partial class ExampleForm : Form
     {
```

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

      public ExampleForm() { ...
```

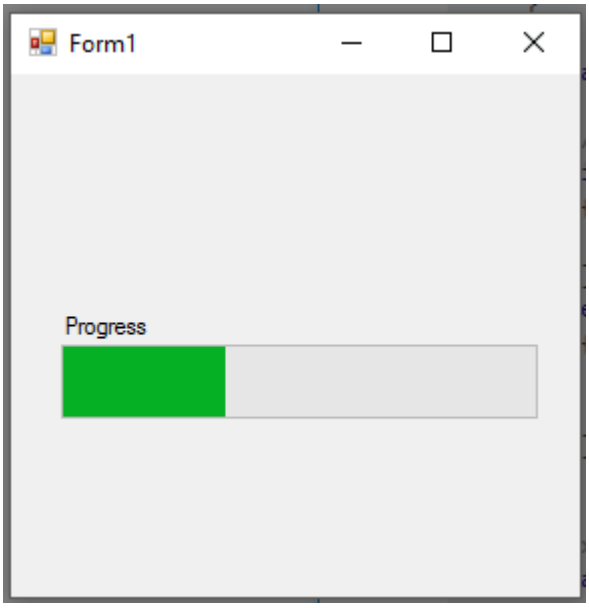
BackgroundWorkerをしてタスクをする。

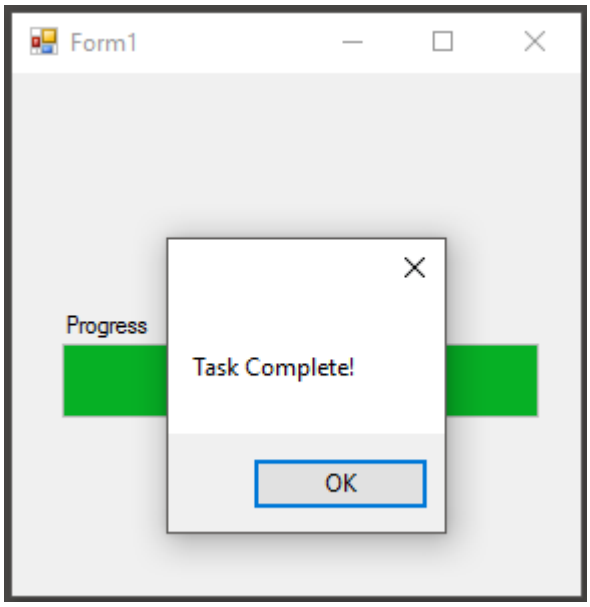
のは、BackgroundWorkerをしてWinForms ProgressBarをするをしています。
backgroundWorkerは、UIスレッドをブロックせずにプログレスバーのをするので、バックグラウンドでがわれているになUIがされます。

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

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

    public Form1()
    {
        InitializeComponent();

        prgProgressBar.Step = 1;

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

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

    }

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

    private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
    {
        //e.Error will contain any exceptions caught by the backgroundWorker
        if (e.Error != null)
        {
            MessageBox.Show(e.Error.Message);
        }
        else
        {
            MessageBox.Show("Task Complete!");
            prgProgressBar.Value = 0;
        }
```

```
    }

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

        int maxProgress = 100;
        prgProgressBar.Maximum = maxProgress;


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


}
```

はのとおりです...





オンラインでBackgroundWorkerをむ
https://riptutorial.com/ja/csharp/topic/1588/backgroundworker

10: BigInteger

いつうべきか

`BigInteger`オブジェクトはにRAMでにいです。したがって、になとき、すなわちになののにのみしてください。

さらに、これらのオブジェクトにするすべてのは、それらのなよりもいにく、このは、サイズではないので、がえるにつれてさらににになります。したがって、な`BigInteger`がなRAMをすべてすることによってクラッシュするがあります。

があなたのソリュ―ションにとってなは、 `Byte[]`をラップし、なをでオ―バ―ロ―ドするクラスをしてこのをするがです。しかし、これにはながです。

Examples

の1000のフィボナッチをする

`using System.Numerics`を`using System.Numerics`して、 `System.Numerics`へのをプロジェクトにします。

```
using System;
using System.Numerics;

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

このなアルゴリズムは、なくとも1000の10にするまでフィボナッチをし、それをします。このは、 `ulong`ができるよりもかなりきいです。

には、 `BigInteger`クラスののは、アプリケ―ションができるRAMのです。

`BigInteger`は、.NET 4.0でのみできます。

オンラインでBigIntegerをむ https://riptutorial.com/ja/csharp/topic/5654/biginteger

11: BindingList

Examples

N * 2をける

これは、Windowsフォームのイベントハンドラにされます

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

これはするのにがかかり、するにはのようにします。

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

リストにする

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

オンラインでBindingList をむ https://riptutorial.com/ja/csharp/topic/182/bindinglist--t-

12: C3.0の

Cバージョン3.0は.Netバージョン3.5のとしてリリースされました。このバージョンでされたのくは、LINQLanguage INtegrated Queriesをサポートしていました。

されたのリスト

- LINQ
- ラムダ
- メソッド
- の
- にけされた
- オブジェクトとコレクションの
- にされるプロパティ
- ツリー

Examples

にけされたvar

`var`キーワードをすると、プログラマーはコンパイルににをできます。 `var`は、にされたとじをちます。

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

ののは`int` 、 `double` 、 `StringBuilder`、およびです。

`var`はにされないことにすることがです。 `SquaredNumber = Builder` 、あなたがしようとしているため、でない`int`のインスタンスに`StringBuilder`

クエリLINQ

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IEnumerable<int> query = from x in array
                         where x % 2 == 1
                         orderby x descending
```

```
                        select x * x;
// Result: 49, 25, 1
```

C3.0のLINQサブセクションのwikipediaの

1では、SQLクエリとにえるようにされたクエリをしています。

```
//Example 2
IEnumerable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

C3.0のLINQサブセクションのwikipediaの

2では、1とじをるためにメソッドをしています。

Cでは、LINQクエリはLINQメソッドののであることにすることがです。コンパイラは、コンパイルにクエリをメソッドびしにします。いくつかのクエリは、メソッドのでするがあります。MSDNから - たとえば、されたにするのをするクエリをするには、メソッドびしをするがあります。

ラムダエクスプレッション

Lambda Expresionsは、にされたパラメータとりをにするメソッドのです。それらのはメソッドよりもではなく、プログラミングスタイルにいます。

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

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

のコードは、1から10までのの2のをコンソールにします。

のラムダは、リストのをにします。 1つのパラメータしかないので、かっこはすることができます。にじてかっこをめることができます。

```
.Select( (number) => number * number);
```

パラメータをにしてかっこをするがあります。

```
.Select( (int number) => number * number);
```

ラムダボディはであり、のリターンをちます。あなたがむなら、のをうこともできます。よりなラムダにはです。

```
.Select( number => { return number * number; } );
```

selectメソッドは、されたをむしいIEnumerableをします。

2のラムダは、selectメソッドからされたリストのをします。のパラメータがあるので、かっこがです。パラメータのはにけされていますが、これはではありません。のはです。

```
.Aggregate( (first, second) => { return first + second; } );
```

これとじように

```
.Aggregate( (int first, int second) => first + second );
```

の

は、をににすることなく、みりプロパティのセットをのオブジェクトにカプセルするなをします。はコンパイラによってされ、ソースコードレベルではできません。プロパティのは、コンパイラによってされます。

newキーワードのにカッコ { をけて、のをできます。ので、のコードのようなプロパティをすることができます。

```
var v = new { Amount = 108, Message = "Hello" };
```

また、のをすることもできます。のコードをしてください

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

またはLINQクエリでします。

```
var productQuery = from prod in products
                   select new { prod.Color, prod.Price };
```

オンラインでC3.0のをむ https://riptutorial.com/ja/csharp/topic/3820/c-3-0の

13: C4.0の

Examples

オプションのパラメータとき

そのがオプションのは、をできます。すべてのオプションは、のデフォルトをちます。をしないは、デフォルトをとります。オプションのデフォルトは、a

1. 。
2. enumやstructなどのでなければなりません。
3. フォームのデフォルトvalueTypeのであるがあります。

パラメータリストのにするがあります

デフォルトをつメソッドパラメータ

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

MSDNによると、きは、

パラメータのをけることによって、にをすことができます。にしていないパラメータのをえておくはありません。びされたのパラメータリストのパラメータのをるはありません。のパラメータは、そのでできます。

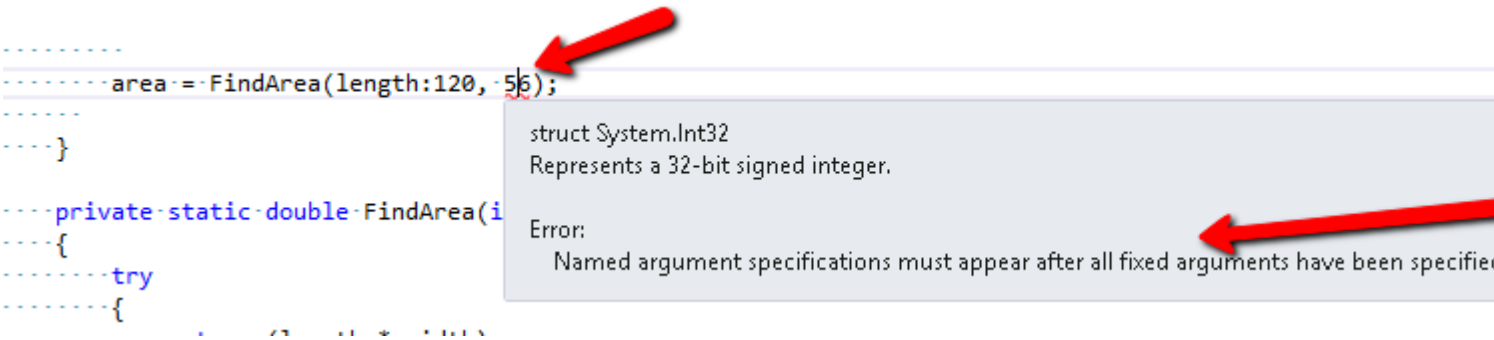
き

```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

きのの

きのは、すべてのがされたでなければなりません。

のにきをすると、のようにコンパイルエラーがします。



きのは、すべてのがされたでなければなりません

ジェネリックインターフェイスとデリゲートは、 `out`と`in`キーワードをそれぞれして、パラメータをまたはとしてマークできます。これらのは、、および、そしてコンパイルとのでされます。

たとえば、のインタフェース`IEnumerable<T>`はであるとされています。

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

のインタフェースIComparerは、なものとしてされています。

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

COMをするのオプションのキーワードref

COMインターフェイスによってされるメソッドをびすときは、メソッドのびしのためのrefキーワードがオプションになりました。きのCOMメソッド

```
void Increment(ref int x);
```

びしはのようにくことができます。

```
Increment(0); // no need for "ref" or a place holder variable any more
```

メンバー

しいの`dynamic`がCシステムにされました。 `System.Object`としてわれますが、さらに、そのようなのにするのメンバアクセスメソッドびし、フィールド、プロパティ、インデクサアクセス、またはデリゲートびしまでそのはされます。これは、ダックタイピングまたはレイトバインディングとばれます。えば

```
// Returns the value of Length property or field of any object
```

```
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");        // a string has a Length property,
GetLength(new int[] { 1, 2, 3 }); // and so does an array,
GetLength(42);                    // but not an integer - an exception will be thrown
                                  // in GetLength method at run-time
```

この、よりなReflectionをするためになが されます。それはまだフ―ドのでReflectionをしていますが、はキャッシングのおかげでくなります。

このはにとのをとしています。

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

は、きコ―ドでもアプリケ―ションをっています。たとえば、Visitorパタ―ンをせずにディスパッチがになるなどです。

オンラインでC4.0のをむ https://riptutorial.com/ja/csharp/topic/3093/c-4-0の

14: C5.0の

-
- パブリックタスク MyTask **Async** {doSomething; }

  MyTaskAsync;をちます。
- パブリックタスク`<string>` MyStringTask **Async** {return getSomeString; }

  string MyString = MyStringTaskAsyncをします。
- の
- public void MyCallerAttributesstring MyMessage、

  [CallerMemberName]MemberName = ""、

  [CallerFilePath] string SourceFilePath = ""、

  [CallerLineNumber] int LineNumber = 0
- Trace.WriteLine "のメッセージ" + MyMessage;

  Trace.WriteLine "Member" + MemberName;

  Trace.WriteLine "ソースファイルパス" + SourceFilePath;

  Trace.WriteLine "" + LineNumber;

パラメーター

パラメータきのメソッド/	
`Type<T>`	`T`はりのです

C5.0はVisual Studio .NET 2012とされています

Examples

`async`と`await`は、スレッドをし、がするのをってからにむことで、パフォーマンスをさせることをとした2つのです。

さをすにをするをにします。

```
//This method is async because:
```

```
//1. It has async and Task or Task<T> as modifiers
//2. It ends in "Async"
async Task<int> GetStringLengthAsync(string URL){
    HttpClient client = new HttpClient();
    //Sends a GET request and returns the response body as a string
    Task<string> getString = client.GetStringAsync(URL);
    //Waits for getString to complete before returning its length
    string contents = await getString;
    return contents.Length;
}

private async void doProcess(){
    int length = await GetStringLengthAsync("http://example.com/");
    //Waits for all the above to finish before printing the number
    Console.WriteLine(length);
}
```

ファイルをダウンロードし、がされたときやダウンロードがしたときのをすのをにしますこれには2りのがあります。

1

```
//This one using async event handlers, but not async coupled with await
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);
    //Download the file asynchronously
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
}

//event called for when download progress has changed
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    //example code
    int i = 0;
    i++;
    doSomething();
}

//event called for when download has finished
private void FileCompleted(object sender, AsyncCompletedEventArgs e){
    Console.WriteLine("Completed!")
}
```

2

```
//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first
example
}
```

```
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    int i = 0;
    i++;
    doSomething();
}
private void doProcess(){
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}
```

の

CIAは、となるをびすものからをするなとしてされています。には1つのしかありませんが、3つのしかありません。

```
//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}
```

```
//Message: Show my attributes.
//Member: doProcess
//Source File Path: c:\Path\To\The\File
//Line Number: 13
```

オンラインでC5.0のをむ https://riptutorial.com/ja/csharp/topic/4584/c-5-0の

15: C6.0の

き

Cのこの6のは、Roslynコンパイラによってされます。このコンパイラは、.NET Frameworkのバージョン4.6でリリースされましたが、のフレームワークバージョンをにして、のあるでコードをできます。 Cバージョン6のコードは、にのあるで.NET 4.0にコンパイルできます。のフレームワークでもできますが、のフレームワークサポートがながしくしないことがあります。

Cの6のバージョンは、20157にVisual Studio 2015と.NET 4.6とともにリリースされました。

しいをするだけでなく、コンパイラのなきえもまれています。は`csc.exe`はC ++でかれたネイティブWin32アプリケーション`csc.exe`が、C6ではCでかれた.NETアプリケーションになりました。このきしはプロジェクト "Roslyn"とばれ、コードはオープンソースでGitHubでです。

Examples

オペレータ

`nameof`は、コードのを`string`としてし`string`。これは、メソッドのにするや`INotifyPropertyChanged`にをスローするときに`INotifyPropertyChanged`。

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

`nameof`はコンパイルにされ、をリテラルにします。これは、それらのメンバーをするのにもです。のをしてください。

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

`nameof`はコンパイルなので、、 `case`ラベル、 `switch`などでできます。

`Enum` `nameof`をするとです。のわりに

```
Console.WriteLine(Enum.One.ToString());
```

をすることができます。

```
Console.WriteLine(nameof(Enum.One))
```

どちらのもは`One`になります。

`nameof`はなをしてメンバーにアクセスできます。わりに

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

のものにきえることができます

```
string lengthName = nameof(string.Length);
```

ので、は`Length`になります。しかし、はなインスタンスのをぎます。

の`nameof`はほとんどのでしますが、いくつかのがあります。たとえば、いているジェネリックまたはメソッドのりにして、 `nameof`をすることはできません。

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

さらに、ジェネリックにすると、ジェネリックのパラメータはされます。

```
Console.WriteLine(nameof(List<int>));  // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

よりくのについては、 `nameof`のトピックをしてください。

のバージョンの

`nameof`は、6.0よりのバージョンのCではしませんが、のように`MemberExpression`をすることでのをできます。

6.0

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

```
public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

このでは、すべてのびしでツリーがされるため、コンパイルにされ、にオーバーヘッドがゼロになるnameofとして、パフォーマンスがにすることにしてください。

ボディメンバー

ボディメンバは、ラムダをメンバとしてできます。シンプルなメンバの、よりクリーンでみやすいコードになります。

は、プロパティ、インデクサ、メソッド、およびにできます。

プロパティ

```
public decimal TotalPrice => BasePrice + Taxes;
```

それとです

```
public decimal TotalPrice
{
    get
    {
        return BasePrice + Taxes;
    }
}
```

きのをプロパティとともにすると、プロパティはゲッターのプロパティとしてされます。

デモをる

インデクサー

```
public object this[string key] => dictionary[key];
```

それとです

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

メソッド

```
static int Multiply(int a, int b) => a * b;
```

それとです

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

`void`メソッドでもできます。

```
public void Dispose() => resource?.Dispose();
```

`ToString`のオーバーライドを`Pair<T>`クラスに`ToString`ことができます

```
public override string ToString() => $"{First}, {Second}";
```

さらに、このされたアプローチは、 `override`キーワードをしてし`override`。

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

これはオペレータがすることもできます

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

エクスプレッションボディメンバーにはいくつかのがあります。ブロックステートメントやブロックをむステートメントをむことはできません `if` 、 `switch` 、 `for` 、 `foreach` 、 `while` 、 `do` 、 `try` など

いくつかの`if`はできえることができます。の`for`および`foreach`は、たとえばLINQクエリにできます。

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

それのは、メンバーのいをすることができます。

エクスプレッションボディメンバーは`async` / `await`むことができますが、くの、です。

```
async Task<int> Foo() => await Bar();
```

のものにきえることができます

```
Task<int> Foo() => Bar();
```

フィルタ

フィルタをすると 、はを`boolean`のでキャッチブロックにし、が`true`されたにのみ`catch`をすることができ`true`。

フィルタは、のでデバッグをさせることができます。ここでは、 `catch`ブロックの`if`をしてをスローすると、ののデバッグのがします。フィルタをすると、がたされないり 、はびしスタックでにしけます。その、フィルタにより、デバッグのがはるかににになります。 `throw`でするわりに、デバッガはのとすべてのローカルをしたまま、を`throw`でします。クラッシュダンプものをけます。

> フィルタはめからCLRでサポートされており、CL.NETのモデルのをすることで、10にわたってVB.NETとFからアクセスになっています。 C6.0のリリースにり、Cにもがされています。

フィルタの

フィルタは、 `catch`に`when`をすることによってされます。 `when`で`bool`をすのをすることができます

awaitをく。されたExceptionexは、 whenからアクセスです。

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

whenをつのcatchブロックをみわせることができます。のwhenがtrueをすと、がされます。そのcatchブロックはされますが、のcatchはされます whenはされません。えば

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                          //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                                       //someCondition evaluates to false
{ ... }
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

なwhen

あぶない

ときフィルタをするようになことができExceptionからスロ―されたwhen、 Exceptionからのwhenはされ、としてわれるfalse。このアプロ―チにより、はなケ―スをしなくてもwhenをくことができます。

のは、このようなシナリオをしています。

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
```

```
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

デモをる

フィルタは、したコードがじにあるに`throw`をするのしたのをします。たとえば、この、は3ではなく6とされます。

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

は、6の`throw`でキャッチしてスロ―されたため、6とされます。

フィルタでもじことはこりません。

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }
```

この`a`は`a`は0で、 `catch`はされますが、3がとしてされます。これはスタックをきさないためです。には、5にがキャッチされないの `a` 、には`0`であるため、6がされないため、6にがスロ―されるがないからです。

としてのロギン グ

こののメソッドびしはをきこすがあるため、フィルタをしてをすることなくをすることができます。これをするなは、に`false`す`Log`メソッドです。これにより、をスロ―するなく、デバッグにログをトレ―スすることができます。

> サ―ドパ―ティのログアセンブリがされているはに、これは、ログのなのようですが、それはなことができることにしてください 。これらは、にされないのあるでないでログイン しているにをスロ―するがありますの**`when(...)`**を。

```
try
```

```
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // This catch block will never be reached
}

// ...

static bool Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
    return false;
}
```

デモをる

のバージョンのCでのなアプローチは、のログとスローでした。

6.0

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
     Log(ex, "An error occurred");
     throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

デモをる

---

finallyブロック

finallyブロックは、がスローされるかどうかになくされます。 whenをつ1つのながフィルタであり、のfinallyブロックにるにスタックのでさらにされます。これにより、コードがグローバルのスレッドのユーザーやカルチャなどをしてfinallyブロックにそうとすると、しないやがするがあります。

finallyブロック

```
private static bool Flag = false;
```

```
static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

EvaluatesToTrue
にインナー—
キャッチ
にに

デモをる

のでは、 `SomeOperation`メソッドがグロ—バルのをびしの`when`に「リ—ク」させたくないは、をする`catch`ブロックもめるがあります。えば

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
```

```
    {
       Flag = false;
       throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

IDisposable.Disposeは、 usingブロックでびされたがスタックをバブリングするににびされるため、じをするためにブロックをするセマンティクスをするIDisposableヘルパクラスをすることもです。

プロパティ

き

プロパティは、に=でできます}。のCoordinateクラスは、プロパティをするためにできるオプションをしています。

6.0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

のなるアクセサ

アクセサ—になるをつプロパティ—をすることができます。されたのをにします。

```
    public string Name { get; protected set; } = "Cheeze";
```

アクセサもすることができinternal、 internal protected、またはprivate。

みりのプロパティ

のにえて、みりのプロパティをすることもできます。ここにがあります

```
    public List<string> Ingredients { get; } =
        new List<string> { "dough", "sauce", "cheese" };
```

このでは、のプロパティをするもしています。また、プロパティはきみにすることもできないため、きみのもされます。

いスタイルpre C6.0

C6よりでは、よりなコードがでした。プロパティのバッキングプロパティとばれる1つのなをして、デフォルトをえたり、のようなpublicプロパティをしたりしました。

6.0

```
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

*C6.0*よりでは、コンストラクタからされたプロパティ *getter*と*setter*をつプロパティ のみみときみをできましたが、そのでインラインプロパティをできませんでした

デモをる

イニシャライザはフィールドイニシャライザとになにされなければなりません。メンバーをするがあるは、とじようにコンストラクターのプロパティをするか、きのプロパティをできます。のようなスタティックコメントアウトは、コンパイラエラーをします

```
// public decimal X { get; set; } = InitMe();  // generates compiler error

decimal InitMe() { return 4m; }
```

しかし、メソッドをってプロパティをすることができます

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);
```

```
    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

このメソッドは、アクセサーのレベルがなるプロパティにもできます。

```
public short Type { get; private set; } = 15;
```

プロパティでは、でプロパティをりてることができます。みりプロパティの、プロパティがであることをするためになすべてのをします。たとえば、ののFingerPrintクラスをえてみましょう。

```
public class FingerPrint
{
  public DateTime TimeStamp { get; } = DateTime.UtcNow;

  public string User { get; } =
    System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

  public string Process { get; } =
    System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

デモをる

オートプロパティまたはフィールドイニシャライザと、 =>とはに=>をするのたのボディメソッドとしないようにしてください{ get; }はまれません{ get; }。

たとえば、のはそれぞれなります。

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

{ get; }は、プロパティのpublicフィールドになります。みりのプロパティUsers1とみり/きみフィールドUsers2は、だけされますが、パブリックフィールドではクラスのコレクションインスタンスがされることがありますが、これははましくありません。をつみりプロパティーをでみりプロパティーにするには、 > from =>するだけでなく、 { get; }。

Users3のなるシンボル =>わりに= は、 HashSet<UserDto>しいインスタンスをすプロパティへのアク

セスをもたらしますが、コンパイラのからてなCは、コレクションメンバーにされます。

のコードはのものとじです

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

インデックス

ィニシャライザをすると、きのオブジェクトをにおよびできます。

これにより、のはににになります。

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

ィンデックスきのゲッターまたはセッターをつオブジェクトは、のでできます。

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };

        Console.ReadKey();
```

```
    }
}
```

インデックスfoo、34
インデックスbar、42

デモをる

クラスにのインデクサがあるは、それらをすべてのステートメントグループにりてることができます。

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}
```

インデックスfoo、34
インデックスbar、42
インデックス10、10
42、の

インデクサ`set`アクセサは、コレクションでされる`Add`メソッドとはなるをするがあることにして`Add`。

えば

```
var d = new Dictionary<string, int>
```

```
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one
```

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

により、 variablesとテキストをしてをすることができます。

な

2つのintfooとbarがされbar。

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

fooは34、barは42です。

デモをる

のはききのようにできます。

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is {{foo}}, and the bar is {{bar}}.");
```

これにより、のがされます。

fooは{foo}、バーは{bar}です。

なリテラルによるの

のに@をすると、がそのままされます。したがって、たとえば、Unicodeやは、したとおりにそのままります。ただし、ののように、されたのにはしません。

```
Console.WriteLine($@"In case it wasn't clear:
```

```
\u00B9
The foo
is {foo},
and the bar
is {bar}.");
```

それがらかでなかった
\ u00B9
フ—
34、
とバ—
42です。

デモをる

では、{}のもできます。はのするにされます。たとえば、 fooとbarをしてするには、でMath.Maxします。

```
Console.WriteLine($"And the greater one is: { Math.Max(foo, bar) }");
```

そして、よりきなものは42

かっことののまたはのスペ—ス、タブ、*CRLF /*をむはにされ、にはまれません

デモをる

のとして、はとしてフォ—マットできます。

```
Console.WriteLine($"Foo formatted as a currency to 4 decimal places: {foo:c4}");
```

Fooは、4までのとしてフォ—マットされました$ 34.0000

デモをる

または、としてすることもできます。

```
Console.WriteLine($"Today is: {DateTime.Today:dddd, MMMM dd - yyyy}");
```

はMonday、July、20 - 2015

デモをる

き3をつステ—トメントもでできます。ただし、これらはでむがあります。コロンは、のようにをするためにされるためです。

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

バーはfooよりもきい

デモをる

とをさせることができます

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00'-bit'} process");
```

32ビットプロセス

エスケープシーケンス

バックスラッシュ \ と " のエスケープは、されていないとまったくじように、およびなのリテラルにしてします。

```
Console.WriteLine($"Foo is: {foo}. In a non-verbatim string, we need to escape \" and \\ with
backslashes.");
Console.WriteLine($@"Foo is: {foo}. In a verbatim string, we need to escape "" with an extra
quote, but we don't need to escape \");
```

Fooは34です。そのままのでは、\と\をバックスラッシュでエスケープするがあります
。
Fooは34です。なでは、 "なでエスケープするがありますが、エスケープするはありま
せん\

されたにかっこ{または}をめるには、2つの{{または}}ます。

```
$"{{foo}} is: {foo}"
```

{foo}は34

デモをる

FormattableString

$"..."のは、 ずしもなではありません 。コンパイラは、コンテキストにじてどのタイプをりてるかをします。

```
string s = $"hello, {name}";
System.FormattableString s = $"Hello, {name}";
System.IFormattable s = $"Hello, {name}";
```

これは、コンパイラがどのオーバーロードされたメソッドがびされるかをするがあるにも、タイ

プのです。

しいの `System.FormattableString` 、されるとともに、をします。これをして、をするアプリケーションをにします。

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

のメソッドをのようにびします。

```
AddLogItem($"The foo is {foo}, and the bar is {bar}.");
```

えば、ロギング･レベルがすでにログをしようとしている、をフォーマットするのパフォーマンス･コストがしないようにすることができます。

のコンバージョン

されたからのながあります。

```
var s = $"Foo: {foo}";
System.IFormattable s = $"Foo: {foo}";
```

また、のでをするための`IFormattable`をすることもできます

```
var s = $"Bar: {bar}";
System.FormattableString s = $"Bar: {bar}";
```

およびの

コードがオンの、されたはすべてCA1305 Specify `IFormatProvider` をします。メソッドをしてのカルチャをすることができます。

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
```

```
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }
    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

に、のカルチャのしいをするには、のをします。

```
Culture.Current($"interpolated {typeof(string).Name} string.")
Culture.Invariant($"interpolated {typeof(string).Name} string.")
```

デフォルトでは、コンパイラはのコードのコンパイルにするされたに `String`をりてているため、`Current`および`Invariant`をメソッドとしてすることはできません。

```
$"interpolated {typeof(string).Name} string.".Current();
```

`FormattableString`クラスにはすでに`Invariant()`メソッドがまれているため、のカルチャにりえるもなは、 `using static`を`using static`です。

```
using static System.FormattableString;



string invariant = Invariant($"Now = {DateTime.Now}");
string current = $"Now = {DateTime.Now}";
```

されたは、 `String.Format()`なるのです。コンパイラ Roslyn はそので`String.Format`します

```
var text = $"Hello {name + lastName}";
```

はのようなものにされます

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

のとLinq

Linqでされたをすると、みやすくすることができます。

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

のようにきすことができます

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select $"{x["foo"]}{x["bar"]}").ToList();
```

な

`string.Format`をすると、なをできます。

```
public const string ErrorFormat = "Exception caught:\r\n{0}";

// ...

Logger.Log(string.Format(ErrorFormat, ex));
```

ただし、されたは、しないをするプレースホルダとはコンパイルされません。はコンパイルされません

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

わりに、をし、 `String`を`Func<>`をします。

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";

// ...

Logger.Log(FormatError(ex));
```

のとローカリゼーション

アプリケーションをローカライズするは、とローカライゼーションをすることがかどうかにうかもしれません。、リソースファイルに`String`をするをつことはいことです

```
"My name is {name} {middlename} {surname}"
```

ずっとみにくいのではなく、

```
"My name is {0} {1} {2}"
```

`String`は、にする`string.Format`でをするのとはなり、コンパイルにします 。されたのは、のコンテキストのをするがあり、リソースファイルにするがあります。つまり、ローカリゼーションをするは、のようにするがあります。

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

でされたのリソースが々のリソースファイルにしくされている、のがられます。

> Bonjour、mon nom est
> ハロー、のistジョン
> こんにちは、のはジョンです

これははすべてのでローカライズされたを、のことをすることにしてください。そうでないは、リソースにプレースホルダをしてのをするか、のカルチャをし、なるケースをむswitch caseをするがあります。リソースファイルのについては、「Cでのローカリゼーションのい」をしてください。

ができないは、ほとんどのができるデフォルトのフォールバックをすることをおめします。はデフォルトのフォールバックとしてをすることをおめします。

それほどではありませんが、ののにされたstringにすることはされます

```
Console.WriteLine($"String has {$"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");
```

> は27です
>
> のクラスはMyClassとばれています。

ついにキャッチをつ

awaitをしawait 、C6のcatchおよびfinallyブロックのTasksまたはTaskOf TResultにawaitをできます。

コンパイラののために、のバージョンのcatchおよびfinallyブロックでawaitをすることはできませんでした。 C6は、 awaitをにすることで、タスクのちをにawaitます。

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

C5では、 boolをするか、tryキャッチのExceptionををするためにするがありました。このメソッドをのにします。

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

Null

?.と?[...]は、 NULLとばれます 。また、 セーフ･ナビゲーション･オペレータなどののでばれることもあります。

これはにち. メンバアクセサがnullにされるにされると、プログラムはNullReferenceExceptionをスローします。がわりに?.をする?. null-conditionalの、はをスローするのではなくヌルにされます。

もし?.がされ、はnullではあり?.と.です。

```
var teacherName = classroom.GetTeacher().Name;
// throws NullReferenceException if GetTeacher() returns null
```

デモをる

classroomがいない、 GetTeacher()はnullすことがありnull。 nullでNameプロパティにアクセスすると、 NullReferenceExceptionがスローされます。

?.をするようにこのステートメントをした?.では、のはnullになりnull。

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

デモをる

その、 classroomもnullでもclassroomませんが、このをのようにくこともできnull

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

デモをる

これはのです。ヌルきをするきアクセスがNULLにされると、チェーンをせずにがただちにNULLにされます。

ヌルきをむのメンバがの、はそののNullable<T>にされるため、 ?.いたのとしてすることはできませ?.。

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null
```

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

NULLでする??

nullきとNULL `??` をみわせて、が`null`されたにデフォルトをすことができ`null`。のをして

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

インデクサーとの

ヌルきはインデクサでできます。

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

のでは、

- の`?.` `classroom`が`null`でないことをし`null`。
- `?` `Students`コレクションが`null`でないことをし`null`。
- `?.`インデクサが`[0]`インデクサが`null`オブジェクトをさなかったことをした。このでもきき `IndexOutOfRangeException` がスローされることにしてください。

---

voidとにう

ヌルきは、 `void`とともにすることもできます。ただし、この、ステートメントは`null`されません。これは`NullReferenceException`ぎます。

```
List<string> list = null;
list?.Add("hi");          // Does not evaluate to null
```

イベントびしで

のイベントをします。

```
private event EventArgs OnCompleted;
```

にイベントをびすときは、サブスクライバがしないにイベントが`null`かどうかをチェックすることがベストプラクティスです。

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

nullきがされたため、びしを1にらすことができます。

```
OnCompleted?.Invoke(EventArgs.Empty);
```

ヌルきはではなくをします。つまり、プロパティりて、イベントサブスクリプションなどにできません。たとえば、のコードはしません。

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or -=
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

ゴッチャ

ごください

```
int? nameLength = person?.Name.Length;    // safe if 'person' is null
```

じではありません

```
int? nameLength = (person?.Name).Length;  // avoid this
```

が

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

は

```
int? nameLength = (person != null ? person.Name : null).Length;
```

にもかかわらず、`?:`が2つののいをするためにここでされていますが、これらのはではありません。これはのでにできます。

```
void Main()
```

```
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

どの

Null
はんだ
0

はんだ
はんだ
0

デモをる

のびしをけるには、のようにします。

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

そしてこのいは、ヌルがなぜでまだサポートされていないのかをします。

の

`using static [Namespace.Type]`ディレクティブをすると、とのメンバーをインポートできます。メソッドは、トップレベルのスコープではなく、メソッド1つのタイプのみとしてインポートされます。

6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;
```

```
class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

ライブデモフィドル

6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

の

のスニペットは、デリゲートがされるときにラムダとはにメソッドグループをすをしています。オーバーロードのは、されたメソッドのりのをチェックするC6のためにあいまいなオーバーロードエラーをさせるわりに、これをします。

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
       Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
       Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

6.0

Func <int>をったオーバーロードがびされました

デモをる

5.0

エラー

エラーCS0121 'Program.OverloadedSystem.Action'および 'Program.Overloaded System.Func'のメソッドまたはプロパティでびしがです。

C6では、 **C5**でエラーがしたラムダののののケースもうまくできます。

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

マイナーなとバグ

はきパラメータのりではされています。 C5ではコンパイルされますが、C6ではコンパイルされません

5.0

```
Console.WriteLine((value: 23));
```

オペランド`is`と`as` 、もはやのグループであることをされません。 C5ではコンパイルされますが、C6ではコンパイルされません

5.0

```
var result = "".Any is byte;
```

ネイティブコンパイラはこれをしましたがをしましたが、にはメソッドのもチェックしていなかったため、 `1.Any is string`または`IDisposable.Dispose is object`です。

のについては、 このリファレンスをしてください。

コレクションのにメソッドをする

コレクションのは、 `IEnumerable`をするクラスをインスタンスするときにでき、のパラメータをる

Addというメソッドがあります。

のバージョンでは、このAddメソッドは、されるクラスのインスタンスメソッドでなければなりませんでした。 C6では、メソッドにすることもできます。

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}
```

これはされます

インスタンスメソッドでされたアイテム1
インスタンスメソッドでされたアイテム2
インスタンスメソッドでされたアイテム3
エクステンションのをしたアイテム4
エクステンションのをしたアイテム5
メソッド6

をにする

C5.0では、だけをすることができました。 Roslyn Analyzersのにより、Cはのライブラリからさ

れたをにするをとします。 C6.0では、プラグマディレクティブはでをすることができます。

```
#pragma warning disable 0501
```

C6.0

```
#pragma warning disable CS0501
```

オンラインでC6.0のをむ https://riptutorial.com/ja/csharp/topic/24/c-6-0の

16: C7.0の

き

C7.0はCの7のバージョンです。このバージョンには、タプルのサポート、ローカル、 `out var`、り、バイナリリテラル、パターンマッチング、スロー、 `ref return`および`ref local`およびボディメンバーリストなどのがまれています。

リファレンス C7の

Examples

アウト var

Cのなパターンは`bool TryParse(object input, out object value)`をにするために`bool TryParse(object input, out object value)`をしています。

`out var`は、みやすくするためのなです。をoutパラメータとしてすとにすることができます。

このようにされたは、それがされたでボディのりのにスコープされます。

C7.0よりの`TryParse`をするは、をびすにをけるをするがあります。

7.0

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

C7.0では、 `out`パラメータにされるのをインラインすることができ、ののはあり`out`。

7.0

```
if (int.TryParse(input, out var value))
{
    Foo(value); // ok
}
else
{
```

```
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body
```

がすにパラメータのならば`out`とされていないあなたは、 をすることができます`_`。

```
p.GetCoordinates(out var x, out _); // I only care about x
```

`out var`は、にっているのをすることができます`out`のパラメータを。のはわりません。また、を`out var`とをたせるためののはありません。このはになです。

`out var`のもうつのは、でできることです。

7.0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
                                  {
                                      Source = x,
                                      Mod2 = x % 2
                                  })
                     .GroupBy(x => x.Mod2)
                     .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

このコードでは、 `int`キーとのをつ`Dictionary`をします。のバージョンのCでは、 `out`これはですをするがあったため、 `TryGetValue`メソッドをすることはでした。しかし、で`out var`々はのタイプをするはありません`out`を。

がラムダのとしてされるので、varはLINQクエリでのがされているため、されるのはこれらのラムダにされます。たとえば、のコードはしません。

```
var nums =
    from item in seq
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

- GitHubのバリデーションをにす

バイナリリテラル

0bは、バイナリリテラルをすためにできます。

バイナリリテラルでは、ゼロと1からのをすることができます。これにより、のバイナリにどのビ

ットがされているかがわかりやすくなります。これは、バイナリフラグをうにです。

は、$34 = 2^5 + 2^1$ の`int`をするのです。

```
// Using a binary literal:
//   bits: 76543210
int a1 = 0b00100010;          // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                  // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

フラグ

これまで、`enum`フラグをするには、このの3つのメソッドのいずれかをするがありました。

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //          decimal        hex       bit shifting
    Monday    =  1,    //    = 0x01    = 1 << 0
    Tuesday   =  2,    //    = 0x02    = 1 << 1
    Wednesday =  4,    //    = 0x04    = 1 << 2
    Thursday  =  8,    //    = 0x08    = 1 << 3
    Friday    = 16,    //    = 0x10    = 1 << 4
    Saturday  = 32,    //    = 0x20    = 1 << 5
    Sunday    = 64,    //    = 0x40    = 1 << 6

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

バイナリリテラルでは、どのビットがされているのがよりで、16とビットのをするはありません。

```
[Flags]
public enum DaysOfWeek
{
    Monday    = 0b00000001,
    Tuesday   = 0b00000010,
    Wednesday = 0b00000100,
    Thursday  = 0b00001000,
    Friday    = 0b00010000,
    Saturday  = 0b00100000,
    Sunday    = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

り

アンダースコア_はりとしてできます。きなリテラルでをグループできることは、にきなをえます。

アンダースコアは、にされているをき、リテラルののでします。なるグループは、なるシナリオやなるベースでをなさないかもしれません。

ののシーケンスは、1つのアンダースコアでることができます。 _はおよびでできます。セパレータにははなはなく、にされます。

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

_りをできない

- の `_121`
- のに `121_`または`121.05_`
- `10_.0`
- `1.1e_1` のに
- `10_f` のに
- バイナリと16のリテラルで`0x`または`0b` たとえば0b_1001_1000をするようにされるがあります

タプルのサポート

タプルは、のけられたリストです。タプルは、タプルのをにするわりに、のエンティティをまとめてし、リレーショナルデータベースの々のつまりレコードをするとしてプログラミングでにされます。

C7.0では、メソッドはのりをつことができます。そのでは、コンパイラはしいValueTupleをします。

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

サイドノート Visual Studio 2017でこれをさせるには、 `System.ValueTuple`パッケージをするがあります。

タプルをすメソッドのがのにされた、メソッドのシグネチャでされたでメンバーにアクセスできます。

```
var result = GetTallies();
```

```
// > result.sum
// 1
// > result.count
// 2
```

タプル

タプルはタプルをそのにします。

たとえば、`GetTallies`をびしてりを2つの々のにすると、その2つのにタプルがされます。

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var`もします

```
(var s, var c) = GetTallies();
```

あなたはまたして、いをすることができ`var`の`()`

```
var (s, c) = GetTallies();
```

のにすることもできます

```
int s, c;
(s, c) = GetTallies();
```

スワップははるかににになりましたはありません

```
(b, a) = (a, b);
```

いことに、どのオブジェクトも、クラスに`Deconstruct`メソッドをすることでできます。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

この、`(localFirstName, localLastName) = person`がその`person Deconstruct`をびしています。

はでもできます。これはとじです

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string
lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

`Person`クラスののアプローチは、 `Name`を`Tuple`としてすることです。のをしてください。

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

それで、をインスタンスすることができますタプルをとしてることができます

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last;   // "Smith"
```

タプルの

コードにタプルをにすることもできます。

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

タプルをするときに、タプルのメンバーになをりてることができます。

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

じシグネチャでされたのタプルするとカウントは、するとしてされます。えば

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

`stats`の、すことができる`stats`とメソッドのりがしています。

リフレクションとタプルフィールド

にメンバーがしません。 Reflectionでは、メンバがしなくてもじとタイプのメンバをつタプルがじとみなされます。タプルを`object`してからじメンバであるがなるをつタプルにしても、はしません。

ValueTupleクラスはメンバーのをしませんが、はTupleElementNamesAttributeのリフレクションをじてできます。このはタプルにはされず、メソッドのパラメータ、り、プロパティ、およびフィールドにされます。これにより、タプルがアセンブリでされるようになります。つまり、メソッドが、int countをす、りにはをむTupleElementNameAttributeがマークされるため、のとカウントがのアセンブリのメソッドのびしでになります""と "カウント"。

ジェネリックスと`async`する

ジェネリックをにサポートするしいタプルな`ValueTuple`をはジェネリックパラメータとしてできます。そうすれば、 `async` / `await`パターンでそれらをすることができます

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

コレクションでする

コードのをけるにづいてするタプルをつけようとしているシナリオで、としてタプルのコレクションをつことはになるかもしれません。

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string,
string>>()
{
```

```
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

しいタプルはのようになります。

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)>
labels = new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

のサンプルタプルのはかなりですが、するラベルのえにより、 "item1"、 "item2"、および "item3" のにして、コードでがみられているかをよりくすることができます。

ValueTupleとTupleのい

`ValueTuple`のなはパフォーマンスです。

	ValueTuple	Tuple
クラスまたは	`struct`	`class`
のの	な	
ネーミング·メンバーおよびそののサポート	はい	いいえ

- GitHubのオリジナルタプルの
- C7.0のなVS 15ソリューション
- NuGetタプルパッケージ

ローカル

ローカルはメソッドでされ、メソッドのではできません。らはすべてのローカルにアクセスし、iterators、 `async` / `await` 、lambdaをサポートします。このようにして、にのりしは、クラスをさせることなくすることができます。として、Intellisenseののパフォーマンスがします。

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

ローカルはLINQのコードをにします。、チェックとのロジックをして、までチェックをにうがあります。

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

ローカルは`async`および`await`キーワードもサポートして`await`ます。

```
async Task WriteEmailsAsync()
```

```
{
    var emailRegex = new Regex(@"(?i)[a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-.]+");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");

    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
    {
        string text;

        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}
```

あなたがづいたかもしれないなことの1つは、ロ―カルを`return`のでることができ、そのにするはないということです。さらに、ロ―カルは、 "lowerCamelCase"にい、クラススコ―プとのをにします。

パタ―ンマッチング

Cのパタ―ンマッチングは、からのパタ―ンマッチングののくをにしますが、になるのとスム―ズにします

switch

パタ―ンマッチングは、 `switch`をしてをオンにします。

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
```

```
}

class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

isは、

パターンマッチングは、 `is`をしてをチェックし、にしいをします。

7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

のようにきすことができます。

7.0

```
if(o is string s)
{
    //Do something with s
};
```

また、パターン`s`スコープは、みスコープのわりにする`if`ブロックのにされています

```
if(someCondition)
{
   if(o is string s)
   {
      //Do something with s
   }
   else
   {
     // s is unassigned here, but accessible
   }

   // s is unassigned here, but accessible
}
// s is not accessible here
```

refリターンとrefローカル

Refりとは、でないポインタにらずにメモリをコピーするのではなく、メモリブロックへのをしてすのにです。

リフレクションリターン

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

これにより、2つのをですことができます。そのうちの1つは、あるにづいてされます。

```
Matrix3D left = …, right = …;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Refローカル

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
…
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

でない

`System.Runtime.CompilerServices.Unsafe`では、にポインタであるかのように`ref`をできるでないがされています。

たとえば、メモリアドレス `ref` をのタイプとしてすると、のようになります。

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

ただし、これをうにはエンディアンにして`BitConverter.IsLittleEndian`。 にじて`BitConverter.IsLittleEndian`チェックし、 `BitConverter.IsLittleEndian`じてして`BitConverter.IsLittleEndian`。

または、でないでをする

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

またはの`Subtract`

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

さらに、2つの`ref`がじかどうか、つまりじアドレスであるかどうかをできます。

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

リンク

Roslyn Github Issue

githubのSystem.Runtime.CompilerServices.Unsafe

をげる

C7.0では、のでとしてスロ―することができます。

```
class Person
{
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No
name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

C7.0では、のからをスロ―したいは、のようにするがあります。

```
var spoons = "dinner,desert,soup".Split(',');

var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

または

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
      {
          throw new Exception("There are no spoons");
      })();
```

C7.0では、はのようにされました。

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

のボディメンバ―リスト

C7.0では、をつことができるもののリストにアクセサ、コンストラクタ、ファイナライザをして

います。

```
class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int,
string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _);              // finalizers

    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

のout varセクションもしてください。

ValueTask

`Task<T>`はクラスであり、がすぐににになったときにりてのなオーバーヘッドがします。

`ValueTask<T>`はであり、 のが`ValueTask<T>`にすでになに`Task`オブジェクトのりてをするためにされました。

したがって、 `ValueTask<T>`は2つのがあります。

1.パフォーマンスの

ここに`Task<T>`があります

- ヒープりてが
- JITで120nsかかる

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

ここに、アナログ`ValueTask<T>`があります

- いいえヒープりてないがそれがで、このにはされていないられている`Task.Delay`が、くの、くのので`async` / `await`シナリオ
- JITで65nsかかる

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2.のの

したいインターフェースのは、さもなければ、 `Task.Run`または`Task.FromResult`いずれかをするようにされるのパフォーマンスペナルティがじる。したがって、にするいくつかのがあります。

しかし、 `ValueTask<T>`では、びしにをえることなく、がまたはのどちらかをにできます。

たとえば、メソッドをしたインタフェースをにします。

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

...そのメソッドがびされるはのとおりです。

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

`ValueTask`では、のコードは、またはのどちらでもします 。

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

ノート

`ValueTask`

はC7.0にされるでしたが、はのライブラリとしてされています。 ValueTask <T>
`System.Threading.Tasks.Extensions`パッケージは、 Nuget Galleryからダウンロードできます。

オンラインでC7.0のをむ https://riptutorial.com/ja/csharp/topic/1936/c-7-0の

17: CStructsをしてUnionをするC Unionと

は、の、にCでされ、じメモリで「なりう」いくつかのなるがまれています。つまり、さとがなるでも、じメモリオフセットからまるなるフィールドをむがあります。これには、メモリのとののがあります。

のコンストラクタのコメントをきめてください。フィールドがされるはにです。にのフィールドをすべてし、のステートメントとしてするをします。フィールドがするため、ののはカウントされます。

Examples

CスタイルのC

は、Cなどのいくつかのでされ、「」するのあるいくつかのなるをみます。つまり、さとがなるでも、じメモリオフセットからまるなるフィールドをむがあります。これには、メモリのとののがあります。としてIPアドレスをえてみましょう。には、IPアドレスはでされますが、Byte1.Byte2.Byte3.Byte4のようになるByteコンポーネントにアクセスしたいことがあります。これは、Int32やlongなどのプリミティブであるか、でするのであれ、どのようなでもします。

Explicit Layout StructsをすることでCでもじをることができます。

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

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

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

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

このでStructをしたら、CでUnionをするときとじようにすることができます。たとえば、IPアドレスをRandom Integerとしてし、アドレスののトークンを '100'にします'ABCD'から '100.BCD'ま

で

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

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

デモをる

CのUnion Typesには、Structフィールドもめることができます

プリミティブとはに、CのExplicit Layoutにはのをめることもできます。フィールドがではなくValueであるり、それはUnionにめることができます

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

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

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

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

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

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

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

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

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

サービスユニオンがい8バイトサイズにまることをできます。

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

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

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

デモをる

オンラインでCStructsをしてUnionをするC Unionとをむ
https://riptutorial.com/ja/csharp/topic/5626/c-structsをしてunionをする-c-unionと-

18: Cコレクションの

Examples

HashSet

これは、ユニークなアイテムのコレクションで、O1ルックアップをしています。

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

のために、リストに`Contains`れるものをすると、パフォーマンスがします。

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains`はハッシュテーブルをしているため、コレクションのになくがにです。

SortedSet

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

T []Tの

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
```

```
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";


// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two
```

リスト

List<T>は、されたのリストです。アイテムは、インデックスによって、、、およびアドレスがです。

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]);    // 5
```

List<T>は、サイズをできるとえることができます。コレクションをにすることは、インデックスをして々のにアクセスするのとにです。そののらかのやそののキ―にづいてにアクセスするには、 Dictionary<T>がよりなをします。

<TKey、TValue>はです。えられたキ―にして、には1つのがあります。

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}
```

```
// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

コレクションのをするときにするキー

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

スタック

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
Console.WriteLine(stack.Pop()); // prints 3
```

LinkedList

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
```

```
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

`LinkedList<T>`は、 にリンクされたリストをします。したがって、にノードのであり、ノードには`T`のがまれています。ノードは、のノードとのノードにリンクされています。

キュー

```
// Initalize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

> スレッドセーフなヘッドアップマルチスレッドではConcurrentQueueをしてください。

オンラインでCコレクションのをむ https://riptutorial.com/ja/csharp/topic/2344/c-コレクションの

19: Cスクリプト

Examples

なコード

なCコードをすることができます

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

typeがされていない、は`object`です。

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

オンラインでCスクリプトをむ https://riptutorial.com/ja/csharp/topic/3780/c-スクリプト

20: CでSQLiteをう

Examples

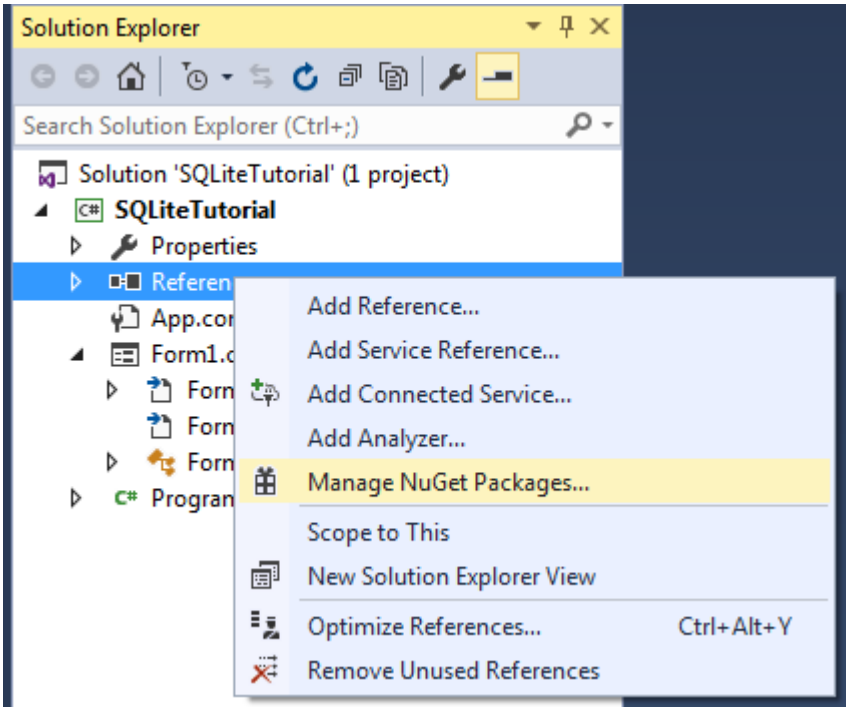
CでSQLiteをってなCRUDをする

まず、アプリケーションにSQLiteサポートをするがあります。それをうには2つのがあります

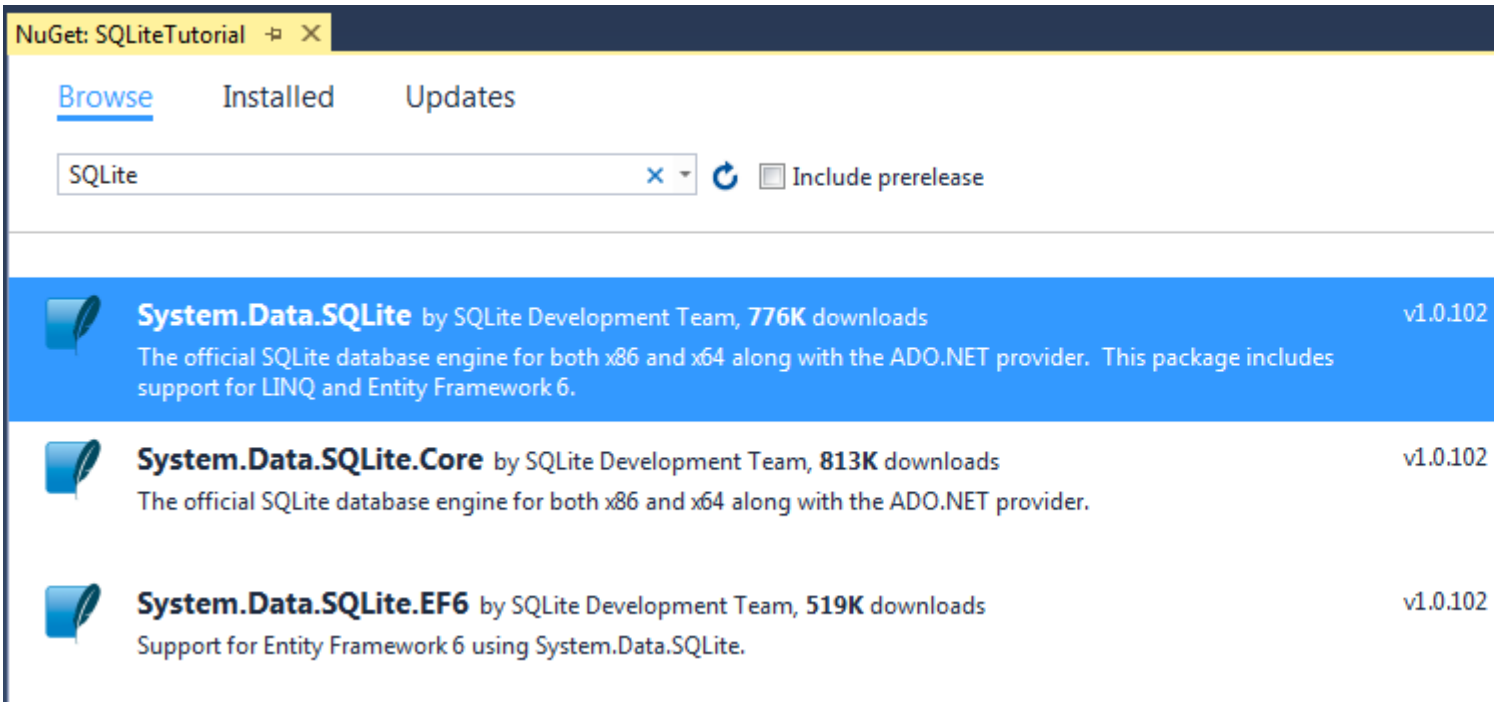
- SQLiteダウンロードページからあなたのシステムにったDLLをダウンロードし、プロジェクトにでしてください
- NuGetでSQLiteをする

たちはそれを2のでやります

にNuGetメニューをきます



System.Data.SQLiteをし、それをして**Install**をします



パッケージマネージャコンソールからインストールすることもできます。

```
PM> Install-Package System.Data.SQLite
```

または、コアのみ

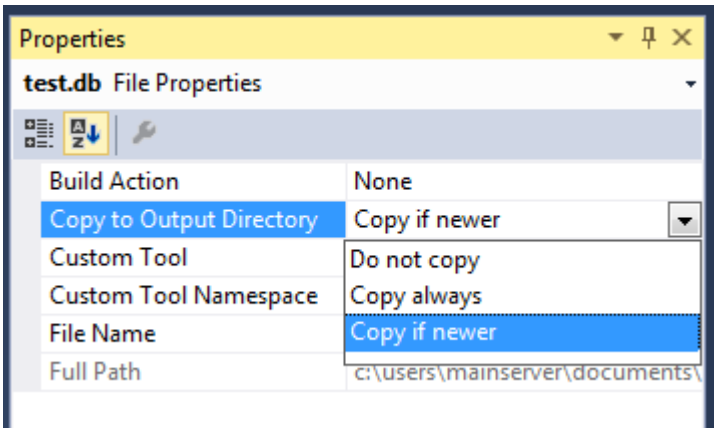
```
PM> Install-Package System.Data.SQLite.Core
```

それはダウンロードですので、コーディングにることができます。

まず、このテーブルをってなSQLiteデータベースをし、それをファイルとしてプロジェクトにします

```
CREATE TABLE User(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL
);
```

また、あなたのニーズにじてに のコピーが あれば 、 コピーのディレクトリプロパティをコピーにすることをれないでください



Userというクラスをします。これはデータベースのエンティティになります

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

クエリのには、データベースに、、またはするための2つのメソッドをします

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }

        return numberOfRowsAffected;
    }
}
```

もう1つはデータベースからのみりです

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;

    using (var con = new SQLiteConnection("Data Source=test.db"))
```

```
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);

            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

CRUDメソッドにる

ユーザーをする

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName,
@lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

ユーザーの

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE
Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

ユーザーの

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}
```

Idユーザーをする

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

クエリの

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
    {
       cmd.CommandText = "query";
       using (SqlDataReader dr = cmd.ExecuteReader())
       {
           while(dr.Read())
           {
```

```
                //do stuff
            }
        }
    }
}
```

`FailIfMissing`をtrueにすると、 `data.db`ファイルがされます。ただし、ファイルはになります。したがって、なテーブルをするがあります。

オンラインでCでSQLiteをうをむ https://riptutorial.com/ja/csharp/topic/4960/c-でsqliteをう

21: Cでをする

- ランダム
- ランダムintシード
- int Next
- intへint maxValue
- intへint minValue、int maxValue

パラ メーター

パラ メーター	
シード	をするための。されていない、デフォルトはのシステムによってされます。
minValue	されるはこのよりさくなりません。されていない、デフォルトは0です。
maxValue	されるはこのよりさくなります。されていない、デフォルトは`Int32.MaxValue`です。
り	ランダムなをつをします。

システムによってされたランダムシードは、なるごとにじではありません。

じにされたはじかもしれません。

Examples

ランダムなintをする

このでは、02147483647ののをします。

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

ランダムなdoubleをする

0と1.0ののをします。 1.0をまない

```
Random rnd = new Random();
```

```
var randomDouble = rnd.NextDouble();
```

されたにランダムなintをする

minValueとmaxValue - 1のをします。

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

のじシーケンスをりしする

じシードのRandomインスタンスをする、じがされます。

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
   Console.WriteLine("Random instance " + i);
   Random rnd = new Random(seed);
   for (int j = 0; j < 5; j++)
   {
      Console.Write(rnd.Next());
      Console.Write(" ");
   }

   Console.WriteLine();
}
```

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

なるをつのランダムなクラスをにする

にされた2つのランダムなクラスは、じシードをちます。

System.Guid.NewGuid().GetHashCode()をするSystem.Guid.NewGuid().GetHashCode()はじでもなるをることができます。

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
```

```
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

なるシードをするのは、のRandomインスタンスをしてシードをすることです。

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

これにより、 rndSeedsシードだけをすることによって、すべてのRandomインスタンスのをすることもになります。のすべてのインスタンスは、そののシードからにされます。

ランダムなをする

されたのにしてNext()オーバーロードをしてaとzにランダムなをし、のintをcharします

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

のパーセンテージであるをする

のなは、いくつかののX%であるをすることです。これは、 NextDouble()をパーセンテージとしてうことでNextDouble()できます。

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

オンラインでCでをするをむ https://riptutorial.com/ja/csharp/topic/1975/c-でをする

22: Cハンドラ

Examples

ハンドラ

```
 public class AuthenticationHandler : DelegatingHandler
    {
        /// <summary>
        /// Holds request's header name which will contains token.
        /// </summary>
        private const string securityToken = "__RequestAuthToken";

        /// <summary>
        /// Default overridden method which performs authentication.
        /// </summary>
        /// <param name="request">Http request message.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/>
class asynchronously.</returns>
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
        {
            if (request.Headers.Contains(securityToken))
            {
                bool authorized = Authorize(request);
                if (!authorized)
                {
                    return ApiHttpUtility.FromResult(request, false,
HttpStatusCode.Unauthorized, MessageTypes.Error, Resource.UnAuthenticatedUser);
                }
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
MessageTypes.Error, Resource.UnAuthenticatedUser);
            }

            return base.SendAsync(request, cancellationToken);
        }

        /// <summary>
        /// Authorize user by validating token.
        /// </summary>
        /// <param name="requestMessage">Authorization context.</param>
        /// <returns>Returns a value indicating whether current request is authenticated or
not.</returns>
        private bool Authorize(HttpRequestMessage requestMessage)
        {
            try
            {
                HttpRequest request = HttpContext.Current.Request;
                string token = request.Headers[securityToken];
                return SecurityUtility.IsTokenValid(token, request.UserAgent,
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
            }
```

```
            catch (Exception)
            {
                return false;
            }
        }
    }
```

オンラインでCハンドラをむ https://riptutorial.com/ja/csharp/topic/5430/c-ハンドラ

23: CLSCompliantAttribute

1. [アセンブリCLSComplianttrue]
2. [CLSComplianttrue]

パラメーター

コンストラクタ	パラメータ
CLSCompliantAttributeブール	CLSCompliantAttributeクラスのインスタンスを、されたプログラムがCLSにしているかどうかをすブールでします。

CLSCommon Language Specificationは、CLIをターゲットとするインフラストラクチャのをするがのCLSとするためにするルールのセットです。

CLIのリスト

ライブラリをする、ほとんどの、アセンブリをCLSCompliantとしてマークするがあります。このは、すべてのCLSでコードをできることをします。これは、CLR Common Language Runtime でコンパイルしてできるでコードをできることをします。

アセンブリに`CLSCompliantAttribute`がされて`CLSCompliantAttribute`、コンパイラはコードがCLSルールにしていないかどうかをチェックし、にじてをします。

Examples

CLSがされるアクセス

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

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

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

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

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

            return increasedAge;
        }

    }
}
```

CLSのルールは、パブリック/されたコンポーネントにのみされます。

CLSルールのなしタイプ/ sbyte

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

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

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


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


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


            return ptr;
        }

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

```
    }
}
```

CLSルールのじ

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

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

        public int CALCULATEAge()
        {
            return 0;
        }

    }
}
```

Visual Basicはとをしない

CLSルール_

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

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

}
```

_でをすることはできません

CLSルールのCLSComplaintクラスからの

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
```

```
{

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

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

}
```

オンラインでCLSCompliantAttributeをむ
https://riptutorial.com/ja/csharp/topic/7214/clscompliantattribute

24: DateTime メソッド

Examples

DateTime.AddTimeSpan

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

DateTime.AddDaysDouble

dateTimeオブジェクトにをします。

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

また、のをすをすることもできます。

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

DateTime.AddHoursDouble

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

DateTime.AddMillisecondsDouble

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
```

```
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

DateTime.CompareDateTime t1、DateTime t2

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

DateTime.DaysInMonthInt32、Int32

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

DateTime.AddYearsInt32

dateTimeオブジェクトにをする

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
   Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
```

```
   Console.WriteLine("{0,2} year(s) from now: {1:d}",
                     ctr, baseDate.AddYears(ctr));
```

DateTimeをうのなの

Wikipediaは、のようなをしています。

1. このは、にじのをしてじをします。のは、プログラムのまたはプログラムのなるでされるのあるれやにすることも、I / Oデバイスからのにすることもできません。
2. のは、なオブジェクトのやI / Oデバイスへのなど、になまたはをきこさない

は、なにづくがあり、くのでこれらをくえています。 1つは、くのジュニアがDateTimeクラスメソッドをしていることをっています。これらのくはですが、あなたがこれらのことをらないは、あなたはきのためにることができます。

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

のでは、コンソールにされたは'26 / 12/2016 'になるとされますが、にはじになります。これは、AddDaysはなメソッドであり、のにはしないためです。されるをるには、のようにAddDaysびしをするがあります。

```
        sample = sample.AddDays(1);
```

DateTime.ParseString

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

DateTime.TryParseString、DateTime

```
// Converts the specified string representation of a date and time to its DateTime equivalent
and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);
```

```
    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

ParseとTryParseで

さまざまなの DateTimesをするときに、それをすることができますオランダの 。

```
DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}
```

の

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

for-loopのとしてのDateTime

```
// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}
```

*TimeSpan*りしはじでします。

DateTime ToString、ToShortDateString、ToLongDateStringおよびToStringがフォーマットされた

```
using System;

public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);
```

```
        Console.WriteLine(date.ToString());        //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy"));    //Outputs: 31/12/2016
    }
}
```

の

のをするには、 `DateTime.Today`プロパティをします。ののDateTimeオブジェクトをします。これが`.ToString()`にされると、デフォルトでシステムの`.ToString()`でわれます。

えば

```
Console.WriteLine(DateTime.Today);
```

のをロ―カルでコンソ―ルにきみます。

するDateTime

の

DateTimeFormatInfoは、なとののためののをします。すべてのは、のDateTimeFormatInfoパターンにしています。

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"                                ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"                               ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"                            LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"                 LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"   FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                              MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                           YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"      RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                   SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
UniversalSortableDateTime
```

カスタムの

のカスタムがあります。

- `y`
- `M`

- `d`
- `h` 12
- `H` 24
- `m`
- `s`
- `f` 2の
- `F` 2の、のゼロはトリムされます
- `t` PMまたはAM
- `z` タイムゾーン。

```
var year =       String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"  year
var month =      String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August"  month
var day =        String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =       String.Format("{0:h hh H HH}",     dt); // "6 06 18 18"       hour 12/24
var minute =     String.Format("{0:m mm}",          dt); // "50 50"            minute
var secound =    String.Format("{0:s ss}",          dt); // "23 23"            second
var fraction =   String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300"    sec.fraction
var fraction2 =  String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"       without zeroes
var period =     String.Format("{0:t tt}",          dt); // "P PM"             A.M. or P.M.
var zone =       String.Format("{0:z zz zzz}",      dt); // "+0 +00 +00:00"    time zone
```

り`/` スラッシュとsepatator `:`コロンもできます。

コード

については、 MSDNをしてください。

DateTime.ParseExactString、String、IFormatProvider

したとカルチャのをして、したとのを、するDateTimeにします。のは、されたとにするがあります。

のをの**DateTime**にする

たとえば、のDateTimeが`08-07-2016 11:30:12 PM`を`MM-dd-yyyy hh:mm:ss tt`とし、それをの`DateTime`オブジェクトにしたいとします

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

のを、のカルチャをたないの**`DateTime`**オブジェクトにする

DateTimeが`dd-MM-yy hh:mm:ss tt`であり、のカルチャをたないの`DateTime`オブジェクトにしたいとします

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

なるフォーマットののカルチャフォーマットをせずにをの**DateTime**オブジェクトにする

たとえば、'23 -12-2016 'や'12 / 23/2016'のようながあり、のカルチャがなくてもの`DateTime`オブジェクトにしたいとします

```
   string date =  '23-12-2016' or date = 12/23/2016';
   string[] formats = new string[] {"dd-MM-yyyy","MM/dd/yyyy"}; // even can add more possible
formats.
   DateTime date = DateTime.ParseExact(date,formats,
CultureInfo.InvariantCulture,DateTimeStyles.None);
```

CultureInfoクラスの**`System.Globalization`**をするがあります。

DateTime.TryParseExactString、String、IFormatProvider、DateTimeStyles、DateTime

した、カルチャの、およびスタイルをして、したとのを、するDateTimeにします。のは、されたとにするがあります。このメソッドは、がしたかどうかをすをします。

えば

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

スタイルフラグのないをします。

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

MとMMのカスタムをします。

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
```

```
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

タイムゾーンをむをします。

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

UTCのrepresengtingをします。

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None,
out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
  Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

```
if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

オンラインでDateTimeメソッドをむ https://riptutorial.com/ja/csharp/topic/1587/datetimeメソッド

25: FileSystemWatcher

- public FileSystemWatcher
- パブリックFileSystemWatcherパス
- パブリックFileSystemWatcherパス、フィルタ

パラメーター

パス	フィルタ
またはUNCUniversal Naming Conventionでするディレクトリ。	るファイルの。たとえば、「* .txt」はすべてのテキストファイルのをします。

Examples

FileWatcher

のでは、にされたディレクトリをする`FileSystemWatcher`をし`FileSystemWatcher`。 このコンポーネントは、 **LastWrite**および**LastAccess**の、ディレクトリのテキストファイルの、、またはをするようにされています。ファイルが、、またはされると、ファイルへのパスがコンソールにされます。ファイルのがされると、いパスとしいパスがコンソールにされます。

このでは、System.DiagnosticsとSystem.IOのをします。

```
FileSystemWatcher watcher;

private void watch()
{
  // Create a new FileSystemWatcher and set its properties.
  watcher = new FileSystemWatcher();
  watcher.Path = path;

 /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
  watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                         | NotifyFilters.FileName | NotifyFilters.DirectoryName;

  // Only watch text files.
  watcher.Filter = "*.txt*";

  // Add event handler.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  // Begin watching.
  watcher.EnableRaisingEvents = true;
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
```

```
  //Copies file to another directory or another action.
  Console.WriteLine("File: " +  e.FullPath + " " + e.ChangeType);
}
```

IsFileReady

よくあるいFileSystemWatcherでまるくのは、ファイルがされるとすぐにFileWatcherイベントがすることをしていません。ただし、ファイルがするまでにがかかることがあります。

たとえば、1 GBのファイルサイズをとります。ファイルaprはのプログラムExplorer.exeはどこかからコピーしていますによってされますが、そのプロセスをするのにかかります。このイベントはがくなり、ファイルのコピーがうのをつがあります。

これは、ファイルがであるかどうかをチェックするです。

```
 public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }

        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

オンラインでFileSystemWatcherをむ
https://riptutorial.com/ja/csharp/topic/5061/filesystemwatcher

26: Funcデリゲート

- `public delegate TResult Func<in T, out TResult>(T arg)`
- `public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)`
- `public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)`
- `public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)`

パラメーター

パラメータ	
`arg`または`arg1`	メソッドののパラメータ
`arg2`	メソッドの2のパラメータ
`arg3`	メソッドの3のパラメータ
`arg4`	メソッドの4のパラメータ
`T`または`T1`	メソッドののパラメータの。
`T2`	メソッドの2パラメータの
`T3`	メソッドの3のパラメータの
`T4`	メソッドの4パラメータの
`TResult`	メソッドのりの

Examples

パラメータなし

このは、をすメソッドをカプセルするデリゲートをするをしています

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}

static void Main(string[] args)
{
```

```
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}
```

の

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}
```

ラムダメソッド

デリゲートがなは、メソッドをりてることができます。

```
Func<int, int> square = delegate (int x) { return x * x; }
```

ラムダをってじことをすることができます

```
Func<int, int> square = x => x * x;
```

どちらのでも、 `square`にされたメソッドをのようにびすことができます

```
var sq = square.Invoke(2);
```

またはとして

```
var sq = square(2);
```

があるためには、メソッドのパラメータとりのがのとしなければならないことにしてください。

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

およびパラメータ

Funcまた、 およびをサポートします

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

オンラインでFuncデリゲートをむ https://riptutorial.com/ja/csharp/topic/2769/funcデリゲート

27: Googleのをインポートする

ユーザーのデータはJSONでされ、データがされ、にこのデータがループされ、Googleのがされます。

Examples

ASP.NET MVCアプリケーションでGoogleGmailのをインポートするには、まず「Google API」をダウンロードします。これにより、のがされます

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

するアプリケーションにこれらをします。

コントローラのソースコード

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = "";  // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
redirectUrl + "&amp&response_type=code&amp&client_id=" + clientId +
"&amp&scope=https://www.google.com/m8/feeds/&amp;approval_prompt=force&amp;access_type=offline");

            return View();
        }
```

```
        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {
                  // You will get all contacts here
                    return View("Index",contacts);
                }
                else
                {
                    return RedirectToAction("Index","Home");
                }
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        public List<GmailContacts> GetAccessToken()
        {
            string code = Request.QueryString["code"];
            string google_client_id = ""; //your google client Id
            string google_client_sceret = ""; // your google secret key
            string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";


            HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
            webRequest.Method = "POST";
            string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
            byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
            webRequest.ContentType = "application/x-www-form-urlencoded";
            webRequest.ContentLength = byteArray.Length;
            Stream postStream = webRequest.GetRequestStream();
            // Add the post data to the web request
            postStream.Write(byteArray, 0, byteArray.Length);
            postStream.Close();
            WebResponse response = webRequest.GetResponse();
            postStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(postStream);
            string responseFromServer = reader.ReadToEnd();
            GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
            /*End*/
            return GetContacts(serStatus);
        }

        public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
        {
            string google_client_id = "";  //client id
            string google_client_sceret = ""; //secret key
            /*Get Google Contacts From Access Token and Refresh Token*/
            // string refreshToken = serStatus.refresh_token;
            string accessToken = serStatus.access_token;
            string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
```

```
            OAuth2Parameters oAuthparameters = new OAuth2Parameters()
            {
                ClientId = google_client_id,
                ClientSecret = google_client_sceret,
                RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
                Scope = scopes,
                AccessToken = accessToken,
                //  RefreshToken = refreshToken
            };

            RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
            ContactsRequest cr = new ContactsRequest(settings);
            ContactsQuery query = new
ContactsQuery(ContactsQuery.CreateContactsUri("default"));
            query.NumberToRetrieve = 5000;
            Feed<Contact> ContactList = cr.GetContacts();

            List<GmailContacts> olist = new List<GmailContacts>();
            foreach (Contact contact in ContactList.Entries)
            {
                foreach (EMail email in contact.Emails)
                {
                    GmailContacts gc = new GmailContacts();
                    gc.EmailID = email.Address;
                    var a = contact.Name.FullName;
                    olist.Add(gc);
                }
            }
            return olist;
        }

        public class GmailContacts
        {
            public string EmailID
            {
                get { return _EmailID; }
                set { _EmailID = value; }
            }
            private string _EmailID;
        }

        public class GooglePlusAccessToken
        {

            public GooglePlusAccessToken()
            { }


            public string access_token
            {
                get { return _access_token; }
                set { _access_token = value; }
            }
            private string _access_token;

            public string token_type
            {
                get { return _token_type; }
                set { _token_type = value; }
            }
            private string _token_type;
```

```
            public string expires_in
            {
                get { return _expires_in; }
                set { _expires_in = value; }
            }
            private string _expires_in;

        }
    }
}
```

ビューのソースコード。

するがあるのアクションメソッドは、のアクションリンクをすることです

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

オンラインでGoogleのをインポートするをむ https://riptutorial.com/ja/csharp/topic/6744/googleのをインポートする

28: HTTPリクエストの

Examples

HTTP POSTリクエストのと

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

HTTP GETリクエストのと

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
```

```
string responseBodyFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

のHTTPコードのエラー—404がつかりませんなど

```
using System.Net;

...

string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as
WebExceptionStatus.ConnectFailure.
        // ...
    }
}
```

JSONでのHTTP POSTリクエストの

```
public static async Task PostAsync(this Uri uri, object value)
{
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());

    using (var client = new HttpClient())
    {
        return await client.PostAsync(uri, content);
```

```
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

HTTP GETリクエストのとJSONリクエストのみり

```
public static async Task<TResult> GetAnsync<TResult>(this Uri uri)
{
    using (var client = new HttpClient())
    {
        var message = await client.GetAsync(uri);

        if (!message.IsSuccessStatusCode)
            throw new Exception();

        return message.ReadAsAsync<TResult>();
    }
}

. . .

public class Result
{
    public double foo { get; set; }

    public string bar { get; set; }
}

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
var result = await uri.GetAsync<Result>();
```

Webページの HTMLをする

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

オンラインでHTTPリクエストのをむ https://riptutorial.com/ja/csharp/topic/1971/httpリクエストの

29: ICloneable

- オブジェクトICloneable.Clone{return Clone; } //カスタムパブリックCloneをするインターフェイスメソッドのプライベート。
- public Foo Clone{しいFooをします。 } //パブリッククローンメソッドはコピーコンストラクタロジックをするがあります。

CLRはされていないメソッドobject Clone()がです。このるいをオーバーライドし、クラスのコピーをすメソッドをするのがなです。

クローンがいコピーかいコピーかをするのはです。をむのは、コピーをうことをおめします。クラスそのものがされているは、いコピーをしてもありません。

C# 、のでインターフェイスメソッドをプライベートにできます。

Examples

ICloneableをクラスにする

ICloneableをクラスにしてICloneable。 のなClone()をし、 object Clone()プライベートにしobject Clone()。

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

その、のようにされます。

```
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}
```

bobのをしても、 bob_cloneのはされません。これは、をするのではなく、をするためです。

にICloneableをする

は=メンバーワイドコピーをうので、のICloneableのははにはありません。しかし、 ICloneableをするのインターフェイスをするがあるかもしれません。

のは、にもコピーがなまたはがまれているです。

```
// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

その、のようにされます。

```
static void Main(string[] args)
{
```

```
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

オンラインでICloneableをむ https://riptutorial.com/ja/csharp/topic/7917/icloneable

30: IComparable

Examples

バージョンをくらべる

クラス

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[0-9]+(\.[0-9]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

テスト

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
```

```
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");
```

ヌル
1
1.0.1
1.1.5
2.0
3.0.10

デモ

Ideoneのライブデモ

オンラインでIComparableをむ https://riptutorial.com/ja/csharp/topic/4222/icomparable

31: IDisposableインターフェイス

- `IDisposable`をしているクラスのクライアントは、オブジェクトのがしたら`Dispose`メソッドをびすようにしています。 CLRには、びす`Dispose`メソッドのオブジェクトをするものはもありません。
- オブジェクトにされたリソースのみがまれているは、ファイナライザをするはありません。の`Dispose`メソッドをするときにクラスがするすべてのオブジェクトにして`Dispose`をびすようにしてください。
- にはだけびすがありますが、クラスを`Dispose`へののびしにしてにすることをおめします。これは、クラスに`private bool`をし、 `Dispose`メソッドがされたときにを`true`にすることで`true`できます。

Examples

リソースのみをむクラス

リソースとは、ランタイムのガベージコレクタがししているリソースのことです。 BCLには、されていないリソースのラッパークラスである`SqlConnection`など、のクラスがされています。これらのクラスはに`IDisposable`インターフェイスをしています。コードをすると、そのコードをクリーンアップすることができます。

クラスにマネージリソースのみがまれているは、ファイナライザをするはありません。

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

リソースとリソースをつクラス

ファイナライズではリソースをすることがです。ファイナライザはのスレッドでします。ファイナライザがされるまでにオブジェクトがしなくなるがあります。された`Dispose(bool)`メソッドのは、されたリソースがファイナライザからびされた`Dispose`メソッドをたないようにするためのなです。

```
public class ManagedAndUnmanagedObject : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
```

```
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

IDisposable、Dispose

.NET Frameworkは、ティアダウンメソッドをとするタイプのインターフェイスをします。

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()`はに、アンマネージなどのリソースのクリーンアップにされます。ただし、のリソースをしていてもにするとです。 GCがにデータベースをクリーンアップするのをつのではなく、の`Dispose()`でしたことをすることができます。

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
   {
       this.CurrentDatabaseConnection.Dispose();
       this.CurrentDatabaseConnection = null;
   }
}
```

されていないポインタやwin32リソースなどのアンマネージリソースにアクセスするがあるは、

SafeHandleをするクラスをし、そのクラスの/ツールをします。

されたリソースをつクラス

IDisposableをするクラスをしてから、されたリソースもむクラスをすることは、かなりです。Disposeメソッドをvirtualキーワードでマークして、クライアントがするリソースをすべてクリーンアップできるようにすることをおめします。

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

キーワードをして

オブジェクトがIDisposableインターフェイスをするとき、それはusingですることができます

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

デモをる

`try/finally`ブロックに[syntatic sugar]を`using`ます。のは、おおよそのようにされます。

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

オンラインでIDisposableインターフェイスをむ
https://riptutorial.com/ja/csharp/topic/1795/idisposableインターフェイス

32: IEnumerable

き

`IEnumerable`は、できるArrayListのようなジェネリックコレクションのインターフェイスです。`IEnumerator<T>`はList <>のようなすべてののインターフェイスです。

`IEnumerable`は、メソッド`GetEnumerator`をするインターフェイス`GetEnumerator`。 `GetEnumerator`メソッドは、foreachのようなコレクションをするオプションをする`IEnumerator`をします。

IEnumerableは、できるすべてのジェネリックコレクションのインターフェイスです。

Examples

IEnumerable

もなでは、IEnumerableをするオブジェクトはのオブジェクトをします。のオブジェクトは、c`foreach`キーワードをしてすることができます。

のでは、オブジェクト`sequenceOfNumbers`はIEnumerableをしています。のをします。 `foreach`ループは、それぞれをにりします。

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

カスタムでIEnumerable

IEnumerableインターフェイスをすると、BCLコレクションとじでクラスをできます。これには、のをするEnumeratorクラスをするがあります。

なコレクションをするにも、のようながあります。

- オブジェクトのではなくにづくのをう
- グラフコレクションのDFSやBFSなど、コレクションになるアルゴリズムをする

```
public static void Main(string[] args) {

    foreach (var coffee in new CoffeeCollection()) {
        Console.WriteLine(coffee);
    }
}
```

```
public class CoffeeCollection : IEnumerable {
    private CoffeeEnumerator enumerator;

    public CoffeeCollection() {
        enumerator = new CoffeeEnumerator();
    }

    public IEnumerator GetEnumerator() {
        return enumerator;
    }

    public class CoffeeEnumerator : IEnumerator {
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };
        int currentIndex = -1;

        public object Current {
            get {
                return beverages[currentIndex];
            }
        }

        public bool MoveNext() {
            currentIndex++;

            if (currentIndex < beverages.Length) {
                return true;
            }

            return false;
        }

        public void Reset() {
            currentIndex = 0;
        }
    }
}
```

オンラインでIEnumerableをむ https://riptutorial.com/ja/csharp/topic/2220/ienumerable

33: ILGenerator

Examples

UnixTimestampヘルパーメソッドをむDynamicAssemblyをします。

このでは、ILGeneratorのをしています。これは、のメンバーおよびされたメンバーをし、なをするコードをします。のコードは、このCコードにするDynamicAssemblyをします。

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds =
checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's to late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubstract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubstract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an,
AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");

var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);
```

```
var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubstract); //Call the substract method on the input
DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on
overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's to late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it

dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

メソッドオーバーライドの

このは、されたクラスの`ToString`メソッドをオーバーライド`ToString`をしています。

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public |
TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

オンラインでILGeneratorをむ https://riptutorial.com/ja/csharp/topic/667/ilgenerator

34: INotifyPropertyChanged インターフェイス

`INotifyPropertyChanged`インタフェースは、クラスがそのプロパティにしてわれたをするがあるに`INotifyPropertyChanged`。 インタフェースは、のイベント`PropertyChanged`します。

XAMLバインディングをすると、 `PropertyChanged`イベントがにされるため、ビューモデルまたはデータコンテキストクラスでINotifyPropertyChangedインターフェイスをするだけで、XAMLバインディングをできます。

Examples

C6でINotifyPropertyChangedをする

`INotifyPropertyChange`のでは、インターフェイスでプロパティをとしてするがあるため、エラーがするがあります。 をよりにするために、`CallerMemberName`をすることができます。

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

`INotifyPropertyChanged`するクラスがいくつかあるは、インターフェイスのをリファクタリングし、ヘルパーメソッドをのクラスにリファクタリングするとです。

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

```
    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

INotifyPropertyChanged メソッドを

の`NotifyPropertyChangedBase`クラスは、のからびすことができるのSetメソッドをしています。

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName =
null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

このなSetメソッドをするには、NotifyPropertyChangedBaseからしたクラスをするだけです。

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

のように、 `Set(ref _fieldName, value);`びすことができます`Set(ref _fieldName, value);`プロパティのセッターで、にじてにPropertyChangedイベントをさせます。

プロパティのをするがあるのクラスからPropertyChangedイベントにできます。

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }

    private readonly SomeViewModel _vm;

}
```

オンラインでINotifyPropertyChangedインターフェイスをむ
https://riptutorial.com/ja/csharp/topic/2990/inotifypropertychangedインターフェイス

35: IQueryableインターフェイス

Examples

LINQクエリをSQLクエリにする

`IQueryable`および`IQueryable<T>`インターフェイスをすると、はLINQクエリ「」クエリをのデータソースリレーショナルデータベースなどにできます。 Cでかれたこのコードを

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

`books`が`IQueryable<Book>`をするである、のクエリは、コードのをするデータであるツリーのでプロバイダ `IQueryable.Provider`プロパティでにされます。

プロバイダは、にツリーをしてをできます。

- `Book`クラスの`Author`プロパティのがあること。
- メソッドが 'equals' `==` であること。
- それとのが`"Stephen King"`。

このをすると、プロバイダはにCクエリをSQLクエリにし、そのクエリをリレーショナルデータベースにして、とするのみをできます。

```
select *
from Books
where Author = 'Stephen King'
```

`query`がされると、プロバイダがびされ`query IQueryable`は`IEnumerable`し`IEnumerable`。

このでされるプロバイダは、クエリするテーブルをり、Cクラスのプロパティをテーブルのにさせるをるためにいくつかののメタデータがですが、そのようなメタデータは`IQueryable`インターフェイスのです。

オンラインでIQueryableインターフェイスをむ
https://riptutorial.com/ja/csharp/topic/3094/iqueryableインターフェイス

36: json.netの

き

JSON.net JsonConverterクラスの。

Examples

なでJsonConverterをう

JsonCoverterをして、ムービーモデルのタイムスパンオブジェクトへのapiレスポンスからランタイムプロパティをシリアルする

JSON http://www.omdbapi.com/?i=tt1663662

```
{
    Title: "Pacific Rim",
    Year: "2013",
    Rated: "PG-13",
    Released: "12 Jul 2013",
    Runtime: "131 min",
    Genre: "Action, Adventure, Sci-Fi",
    Director: "Guillermo del Toro",
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham
(story)",
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and
a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to
save the world from the apocalypse.",
    Language: "English, Japanese, Cantonese, Mandarin",
    Country: "USA",
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",
    Poster: "https://images-na.ssl-images-
amazon.com/images/M/MV5BMTY3MTI5NjQ4Nl5BMl5BanBnXkFtZTcwOTU1OTU0OQ@@._V1_SX300.jpg",
    Ratings: [{
            Source: "Internet Movie Database",
            Value: "7.0/10"
        },
        {
            Source: "Rotten Tomatoes",
            Value: "71%"
        },
        {
            Source: "Metacritic",
            Value: "64/100"
        }
    ],
    Metascore: "64",
    imdbRating: "7.0",
    imdbVotes: "398,198",
    imdbID: "tt1663662",
```

```
    Type: "movie",
    DVD: "15 Oct 2013",
    BoxOffice: "$101,785,482.00",
    Production: "Warner Bros. Pictures",
    Website: "http://pacificrimmovie.com",
    Response: "True"
}
```

モデル

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

RuntimeSerializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
```

```
namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}
```

それをびす

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

JSONオブジェクトのすべてのフィールドをする

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

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
        CollectFields(token);
    }
```

```
    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}
```

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");
```

デモ
このJSONオブジェクト

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

されるはのようになります。

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

オンラインでjson.netのをむ https://riptutorial.com/ja/csharp/topic/9879/json-netの

37: LINQ to XML

Examples

LINQ to XMLをってXMLをむ

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

LINQをしてそのXMLファイルをみるには

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

にアクセスするには

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
```

```
    Console.WriteLine(employee.Element("Name").Value);
}
```

のにアクセスするには

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
    employee.Element("Name").Value,
    employee.Element("EmpId").Value);
}
```

のをつすべてのにアクセスするには

```
XElement xelement = XElement.Load("Employees.xml");
var name = from nm in xelement.Root.Elements("Employee")
       where (string)nm.Element("Sex") == "Female"
       select nm;
Console.WriteLine("Details of Female Employees:");
foreach (XElement xEle in name)
Console.WriteLine(xEle);
```

のをつのにアクセスするには

```
XElement xelement = XElement.Load("..\\..\\Employees.xml");
var homePhone = from phoneno in xelement.Root.Elements("Employee")
            where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
            select phoneno;
Console.WriteLine("List HomePhone Nos.");
foreach (XElement xEle in homePhone)
{
    Console.WriteLine(xEle.Element("Phone").Value);
}
```

オンラインでLINQ to XMLをむ https://riptutorial.com/ja/csharp/topic/2773/linq-to-xml

38: Linqからオブジェクトへ

き

LINQ to Objectsは、のIEnumerableコレクションでLINQクエリをすることをします。

Examples

LINQ to Objectによるクエリの

LINQクエリはすぐにされません。クエリをするときは、にののためにクエリをするだけです。あなたがにクエリをすることをしたときにのみ、されたクエリえばforループ、ToList、Count、Max、Average、Firstなどをびすとき

これは、 となされます。これにより、クエリをのステップでし、きステートメントについてクエリをし、をとしたにすることができます。

えられたコード

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

のでは、クエリを`query`にするだけです。クエリはされません。

`foreach`はクエリをにします。

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

のLINQメソッドは、クエリの、 `Count` 、 `First` 、 `Max` 、 `Average`もトリガします。らはのをします。 `ToList`と`ToArray`はそれぞれをめてリストまたはにします。

じクエリでのLINQをびすと、にわたってクエリをりしできることにしてください。これは、びしでなるをえるがあります。 1つのデータセットでのみするは、ずリストまたはアレイにしてください。

Cでのオブジェクトへの**LINQ**の

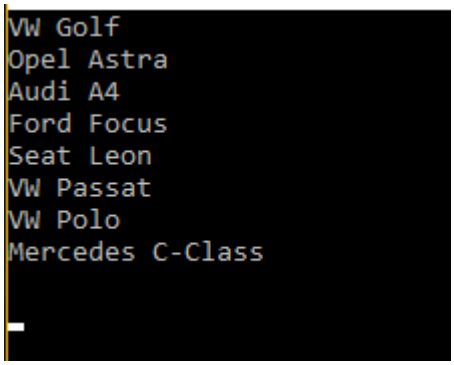
Linqでのな**SELECT**クエリ

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
```

```
                        "Opel Astra",
                        "Audi A4",
                        "Ford Focus",
                        "Seat Leon",
                        "VW Passat",
                        "VW Polo",
                        "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

のでは、のは、LINQをしてするオブジェクトのコレクションとしてされています。 LINQクエリでは、デ―タソ―スcarsとcarをするために、fromがにます。クエリがされると、rangeはのするへのとしてします。コンパイラはのタイプをできるので、にするはありません

のコ―ドをコンパイルしてすると、のがされます。



WHEREをむSELECT

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

WHEREは、WHEREをたすのサブセットをつけてすために、をするためにされます。

のコ―ドをコンパイルしてすると、のがされます。

きリストの

```
var list = from car in cars
           orderby car ascending
           select car;
```

されたデータをソートするとなことがあります。 orderbyをすると、ソートののデフォルトのにってがソートされます。

のコードをコンパイルしてすると、のがされます。

カスタムタイプの

このでは、きリストがされ、された、されます

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
        var car6 = new Car("VW Passat", 97586);
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
```

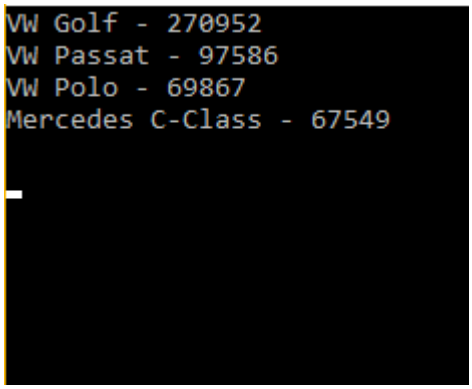
```
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}
```

のコーをコンパイルしてすると、のがされます。

これまでのでは、をりしてにじことをうことができるので、くようなはありません。ただし、のいくつかのでは、LINQ to Objectをしてよりなクエリをし、コードをにらしてよりくのをるをできます。

のでは、60000られているをんでをべえることができます

```
var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```



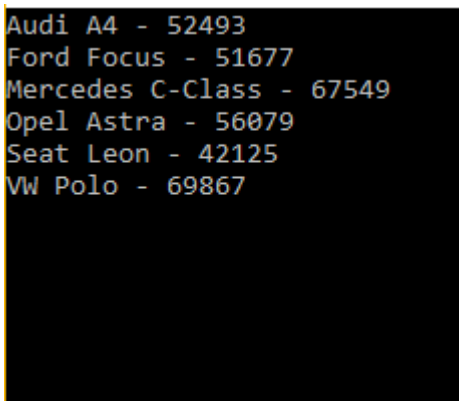
のコーをコンパイルしてすると、のがされます。

のでは、のユニットをしているをし、のアルファベットにべえることができます。

```
var list = from car in cars
```

```
            where car.UnitsSold % 2 != 0
            orderby car.Name ascending
            select car;
```

のコードをコンパイルしてすると、のがされます。



オンラインでLinqからオブジェクトへをむ https://riptutorial.com/ja/csharp/topic/9405/linqからオブジェクトへ

39: LINQクエリ

き

LINQはtegrated **Q uery**で**IN L**の anguageのです。これは、さまざまなのデータソースおよびにわたるデータをするためのしたモデルをすることによって、クエリをするです。じコーディングパターンをして、XMLドキュメント、SQLデータベース、ADO.NETデータセット、.NETコレクション、およびLINQプロバイダがなそののデータをクエリおよびします。

- クエリ
    - <コレクション>の<>から
    - [from <range variable> in <collection>、...]
    - <フィルタリング、、グループ、、...> <ラムダ>
    - <selectまたはgroupBy> <をする>
- メソッドの
    - Enumerable.Aggregatefunc
    - Enumerable.Aggregateseed、func
    - Enumerable.Aggregateseed、func、resultSelector
    - Enumerable.All
    - Enumerable.Any
    - Enumerable.Any
    - Enumerable.AsEnumerable
    - Enumerable.Average
    - Enumerable.Averageセレクタ
    - Enumerable.Cast <Result>
    - Enumerable.Concatsecond
    - Enumerable.Containsvalue
    - Enumerable.Containsvalue、comparer
    - Enumerable.Count
    - Enumerable.Count
    - Enumerable.DefaultIfEmpty
    - Enumerable.DefaultIfEmptydefaultValue
    - Enumerable.Distinct
    - Enumerable.Distinctcomparer
    - Enumerable.ElementAtindex
    - Enumerable.ElementAtOrDefaultindex
    - Enumerable.Empty
    - Enumerable.Except
    - Enumerable.Exceptsecond、comparer
    - Enumerable.First
    - Enumerable.First

- Enumerable.FirstOrDefault
- Enumerable.FirstOrDefault
- Enumerable.GroupBykeySelector
- Enumerable.GroupBykeySelector、resultSelector
- Enumerable.GroupBykeySelector、elementSelector
- Enumerable.GroupBykeySelector、comparer
- Enumerable.GroupBykeySelector、resultSelector、comparer
- Enumerable.GroupBykeySelector、elementSelector、resultSelector
- Enumerable.GroupBykeySelector、elementSelector、comparer
- Enumerable.GroupBykeySelector、elementSelector、resultSelector、comparer
- Enumerable.Intersect
- Enumerable.Intersectsecond、comparer
- Enumerable.Joininner、outerKeySelector、innerKeySelector、resultSelector
- Enumerable.Joininner、outerKeySelector、innerKeySelector、resultSelector、comparer
- Enumerable.Last
- Enumerable.Last
- Enumerable.LastOrDefault
- Enumerable.LastOrDefault
- Enumerable.LongCount
- Enumerable.LongCount
- Enumerable.Max
- Enumerable.Maxセレクタ
- Enumerable.Min
- Enumerable.Minセレクタ
- Enumerable.OfType <TResult>
- Enumerable.OrderBykeySelector
- Enumerable.OrderBykeySelector、comparer
- Enumerable.OrderByDescendingkeySelector
- Enumerable.OrderByDescendingkeySelector、comparer
- Enumerable.Range、カウント
- Enumerable.Repeat、カウント
- Enumerable.Reverse
- Enumerable.Selectセレクタ
- Enumerable.SelectManyセレクタ
- Enumerable.SelectManycollectionSelector、resultSelector
- Enumerable.SequenceEqualsecond
- Enumerable.SequenceEqualsecond、comparer
- Enumerable.Single
- Enumerable.Single
- Enumerable.SingleOrDefault
- Enumerable.SingleOrDefault
- Enumerable.Skipcount
- Enumerable.SkipWhile

- Enumerable.Sum
- Enumerable.Sumセレクタ
- Enumerable.Takecount
- Enumerable.TakeWhile
- orderedEnumerable.ThenBykeySelector
- orderedEnumerable.ThenBykeySelector、comparer
- orderedEnumerable.ThenByDescendingkeySelector
- orderedEnumerable.ThenByDescendingkeySelector、comparer
- Enumerable.ToArray
- Enumerable.ToDictionarykeySelector
- Enumerable.ToDictionarykeySelector、elementSelector
- Enumerable.ToDictionarykeySelector、comparer
- Enumerable.ToDictionarykeySelector、elementSelector、comparer
- Enumerable.ToList
- Enumerable.ToLookupkeySelector
- Enumerable.ToLookupkeySelector、elementSelector
- Enumerable.ToLookupkeySelector、comparer
- Enumerable.ToLookupkeySelector、elementSelector、comparer
- Enumerable.Union
- Enumerable.Unionsecond、comparer
- Enumerable.Where
- Enumerable.Zipsecond、resultSelector

LINQクエリをするには、 `System.Linq`をインポートするがあります。

Method Syntaxはよりでがありますが、Query Syntaxはよりシンプルでいれたものになります。Queryでされたすべてのクエリは、コンパイラによってにされるため、パフォーマンスはじです。

オブジェクトは、されるまでされないため、パフォーマンスのなしにまたはすることができます。

Examples

どこで

されたがであるのサブセットをします。

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

メソッド

```
// Select all trees with name of length 3
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

クエリ

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

の

Selectをすると、IEnumerableをしているすべてのデータのすべてのにをできます。

のリストののののをします。

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

ラムダの

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

O
B
B
E
H
M

.NET Fiddleのライブデモ

LINQクエリの

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

メソッドの

くのLINQは`IEnumerable<TSource>`し、 `IEnumerable<TResult>`もします。パラメータ`TSource`と`TResult`は、のメソッドとそれにされたにじて、じをするとしないがあります。

これのいくつかのがあります

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)
```

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

のメソッドチェーニングでは、するにセットをするがありますが、LINQは、でEnumerableとEnumeratorをするyield return [MSDN]をして、 をします。 LINQでするプロセスは、に、なものをすることによってされるまで、のセットのためのなイテレータをしています。

これにより、これらのをにさせることができます。ここでは、あるがののにすることができます。このスタイルのコードは、1つのステートメントでくのシーケンスベースのをするためにできます。

たとえば、 `Select` 、 `Where` 、および`OrderBy`をみわせて、1つのステートメントでシーケンスを、フィルタリング、ソートすることができます。

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
        .Select(n => n * 2)    // Multiply each number by 2
        .Where(n => n != 6)    // Keep all the results, except for 6
        .OrderBy(n => n);      // Sort in ascending order
```

2
4
8

.NET Fiddleのライブデモ

な`IEnumerable<T>`をしてすはすべて、のステートメントでとしてできます。このなプログラミングスタイルはで、のメソッドをするときにはするがあります。

とりし

`Enumerable`の`Range`と`Repeat`メソッドは、なシーケンスをするためにでき`Enumerable`。

`Enumerable.Range()`は、とカウントがえられたのシーケンスをします。

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

.NET Fiddleのライブデモ

りす

`Enumerable.Repeat()`は、となりしをしてりしのシーケンスをし`Enumerable.Repeat()`。

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

.NET Fiddleのライブデモ

スキップテイク

Skipメソッドは、ソースコレクションのからいくつかのをいたコレクションをします。されるのは、としてえられたです。にされたよりないのがコレクションにある、のコレクションがされます。

Takeメソッドは、ソースコレクションのからいくつかのをむコレクションをします。まれるのはとしてえられたです。にされたよりないのがコレクションにある、されるコレクションにはソース·コレクションとじがまれます。

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo        = values.Skip(2);         // { 3, 2, 1 }
var takeThree      = values.Take(3);         // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero       = values.Take(0);         // An IEnumerable<int> with 0 items
```

.NET Fiddleのライブデモ

Skipと**Take**は、のページけをうためににされます。

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

> LINQ to Entitiesは、けされたクエリのスキップしかサポートしていません。あなたがメッセージでサポートをしますせずにスキップしようとすると、「『をスキップ』はエンティティへのLINQでソートされたのためにサポートされています。『のOrderBy』メソッドのにびされなければならない『スキップ』。」

に、FirstOrDefault、Last、LastOrDefault、Single、および**SingleOrDefault**

6つのメソッドはすべて、シーケンスののをし、のにかかわらずびすことができます。

`predicate`とするの、または`predicate`がされていないは、ソースシーケンスのにじて、のようにします。

- シーケンスのの、またはされた`predicate`するのをします。
- シーケンスにがまれていないは、「シーケンスにがありません」というメッセージとともに

`InvalidOperationException`がスローされます。
- シーケンスにされた`predicate`とするがまれていない、 `InvalidOperationException`がスローされ、「シーケンスにするがまれていません」というメッセージがされます。

```
// Returns "a":
new[] { "a" }.First();

// Returns "a":
new[] { "a", "b" }.First();

// Returns "b":
new[] { "a", "b" }.First(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].First();
```

.NET Fiddleのライブデモ

FirstOrDefault

- シーケンスのの、またはされた`predicate`するのをします。
- シーケンスにがまれていない、またはされた`predicate`にするがないは、 `default(T)`をしてシーケンス.タイプのデフォルトをします。

```
// Returns "a":
new[] { "a" }.FirstOrDefault();

// Returns "a":
new[] { "a", "b" }.FirstOrDefault();

// Returns "b":
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].FirstOrDefault();
```

.NET Fiddleのライブデモ

- シーケンスのの、またはされた`predicate`するのをします。

- シーケンスにがまれていないは、「シーケンスにがありません」というメッセージとともに`InvalidOperationException`がスローされます。
- シーケンスにされた`predicate`とするがまれていない、`InvalidOperationException`がスローされ、「シーケンスにするがまれていません」というメッセージがされます。

```
// Returns "a":
new[] { "a" }.Last();

// Returns "b":
new[] { "a", "b" }.Last();

// Returns "a":
new[] { "a", "b" }.Last(x => x.Equals("a"));

// Returns "be":
new[] { "ba", "be" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].Last();
```

LastOrDefault

- シーケンスのの、またはされた`predicate`するのをします。
- シーケンスにがまれていない、またはされた`predicate`にするがないは、`default(T)`をしてシーケンス.タイプのデフォルトをします。

```
// Returns "a":
new[] { "a" }.LastOrDefault();

// Returns "b":
new[] { "a", "b" }.LastOrDefault();

// Returns "a":
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));

 // Returns "be":
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].LastOrDefault();
```

シングル

- シーケンスにに1つのがまれている、またはされた`predicate`とするが1つだけまれている、そのがされます。

- シーケンスにがまれていない、またはされた`predicate`にするがないは、「シーケンスにがまれていません」というメッセージとともに`InvalidOperationException`がスローされます。
- シーケンスにのがまれている、またはのがされた`predicate`とするは、「シーケンスにのがまれています」というメッセージとともに`InvalidOperationException`がスローされます。
- シーケンスにに1つのがまれているかどうかをするために、で2つのをするがあります。

```
// Returns "a":
new[] { "a" }.Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();
```

SingleOrDefault

- シーケンスにに1つのがまれている、またはされた`predicate`とするが1つだけまれている、そのがされます。
- シーケンスにがまれていない、またはされた`predicate`とするがないは、 `default(T)`がされます。
- シーケンスにのがまれている、またはのがされた`predicate`とするは、「シーケンスにのがまれています」というメッセージとともに`InvalidOperationException`がスローされます。
- シーケンスに、された`predicate`とするがまれていないは、 `default(T)`をしてシーケンス·タイプのデフォルトをします。
- シーケンスにに1つのがまれているかどうかをするために、で2つのをするがあります。

```
// Returns "a":
new[] { "a" }.SingleOrDefault();

// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();
```

```
// Returns null:
new string[0].SingleOrDefault();
```

- `FirstOrDefault` 、 `LastOrDefault`または`SingleOrDefault`をしてシ―ケンスにがまれているかどうかをできますが、 `Any`または`Count`ががくなります。これは、シ―ケンスの//の`default(T)`がじように`default(T)`になるがあるため、これらの3つのメソッドのいずれかからの`default(T)`りがシ―ケンスがであることをしないため`default(T)`

- コ―ドのにもしたをします。えば、`Single`あなたのにするコレクションののがあることをするがあるのみ-それの`First` 、シ―ケンスにのがする、 `Single`はをスロ―します。もちろんこれは "* OrDefault" - じにもてはまります。

- について1つのアイテム `Single` 、または1つまたはゼロ `SingleOrDefault` のアイテムのみがクエリによってされることをすることはしばしばですが、これらのメソッドはともコレクションのよりくの、しばしばをとしますクエリに2のがないことをします。これは、えば、のマッチをつけたにできる`First`メソッドのとはなります。

Exceptメソッドは、のコレクションにまれているが2のコレクションにまれていないアイテムのセットをします。デフォルトの`IEqualityComparer`は、2つのセットのをするためにされます。`IEqualityComparer`をとしてけれるオ―バ―ロ―ドがあります。

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

1
4

.NET Fiddleのライブデモ

この`.Except(second)`のにまれるを`second` 、すなわち2び30および5はにまれていない`first`のアレイおよびスキップされます。

`Except Distinct`しますつまり、りしをします。えば

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

1
4

.NET Fiddleのライブデモ

この、1と4は1だけされます。

IEquatableIEquatableか、 IEqualityComparerをすると、なるメソッドをしてをできます。GetHashCodeメソッドもオ―バ―ライドして、 IEquatableにっての$_{object}$ののハッシュコ―ドをすようにするがあることにしてください。

IEquatableをした

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
    }
}
```

ハヌカ

.NET Fiddleのライブデモ

SelectManyシーケンスのシーケンスをする

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

`SelectMany()`するか、シーケンスのシーケンスをしていますが、そのを1つのいシーケンスとしたいにします。

LINQクエリ

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

コレクションのコレクションがあり、にコレクションとコレクションのデータをできるようにしたいは、 `SelectMany`でもです。

なクラスをしましょう

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

のコレクションがあるとしましょう。

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        }
    },
```

```
    new BlogPost()
    {
        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    }
};
```

このコメントにけられている`BlogPost Id`とともにコメントの`Content`をします。そうするために、な`SelectMany`オ―バ―ロ―ドをすることができます。

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId =
post.Id, CommentContent = comment.Content });
```

たちの`commentsWithIds`はのようになります

```
{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}
```

SelectMany

SelectMany linqメソッドは、 `IEnumerable<IEnumerable<T>>`を`IEnumerable<T>` "します。ソ―ス`IEnumerable`まれる`IEnumerable`インスタンスのTはすべて、の`IEnumerable`されます。

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

をシ―ケンスにするセレクタをすると、はそれらのシ―ケンスのが1つずつされます。

`Select()`とはなり、ののはののとじであるはありません。

よりな

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"}
}},
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} }}
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

ボブ
ジャック
ジム
ジョン

.NET Fiddleのライブデモ

すべて

`All`は、コレクションのすべてのがにするかどうかをチェックするためにされます。
をしてください *.Any*

1.のパラメータ

All のパラメータではできません。

2.パラメータとしてのラムダ

All コレクションのすべてのがラムダをたすは`true`し、そうでないは`false`します。

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.All(i => i < 10); // true
bool result = numbers.All(i => i >= 3); // false
```

3.のコレクション

すべて コレクションがでラムダがされているは`true`します。

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

`All`は、としないをつけるとすぐにコレクションのをします。つまり、コレクションがずしもにされるわけではありません。にしないのをつけるためににされます。

タイプにクエリをする/タイプするキャストエレメント

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

`OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumerable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumerable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumerable<IFoo> with all four items
```

`Where`う

```
var foos = collection.Where(item => item is Foo); // result: IEnumerable<IFoo> with item0 and
item1
var bars = collection.Where(item => item is Bar); // result: IEnumerable<IFoo> with item2 and
item3
```

`Cast`をする

```
var bars = collection.Cast<Bar>();               // throws InvalidCastException on the 1st
item
var foos = collection.Cast<Foo>();               // throws InvalidCastException on the 3rd
item
var foosAndBars = collection.Cast<IFoo>();       // OK
```

2つのコレクションをして、デフォルトのをしてなるコレクションをします。

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

.NET Fiddleのライブデモ

ジョイン

は、のキーをしてデータをするなるリストまたはテーブルをするためにされます。

LINQでは、SQLとに、ののがサポートされています。
、、、、。

の2つのリストは、のでされています。

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                        f => f,
                        s => s,
                        (f, s) => new { f, s });

// Result: {"a","a"}
//         {"c","c"}
```

```
var leftOuterJoin = from f in first
                    join s in second on f equals s into temp
                    from t in temp.DefaultIfEmpty()
                    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                    from s in second.Where(x => x == f).DefaultIfEmpty()
                    select new { First = f, Second = s};

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}


// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                    f => f,
                                    s => s,
                                    (f, s) => new { First = f, Second = s })
                                .SelectMany(temp => temp.Second.DefaultIfEmpty(),
                                    (f, s) => new { First = f.First, Second = s });
```

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};
```

```
// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

クロス

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

な

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
//         {null,"d"}
```

の

のはなデ―タをっているため、になるLINQのにすることができますが、では、するがあるをつテ―ブルがあります。

のでは、には1つのクラス`Region`しかされていませんが、じキ―をする2つのなるテ―ブルをしますこのでは、 `first`と`second`はキ―`ID`されてい`ID`。

のデ―タをえてみましょう。

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

データをしますつまり、データをりみます。

```
// Left data
var first = new List<Region>()
                 { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
                 {
                    new Region(1, "Eastern"),  new Region(2, "Western"),
                    new Region(3, "Northern"), new Region(4, "Southern")
                 };
```

このでは、 `first`のがまれていないので、 `second`のにしたいことがわかります。に、はのようになります。

```
// do the inner join
var result = from f in first
             join s in second on f.ID equals s.ID
             select new { f.ID, s.RegionDescription };


 // Result: {1,"Eastern"}
 //         {3, Northern}
 //         {4,"Southern"}
```

この、のオブジェクトがされましたが、これはありませんが、すでになクラスをしています。これをすることができます `select new { f.ID, s.RegionDescription };` `select new Region(f.ID, s.RegionDescription);`とうことができ`select new Region(f.ID, s.RegionDescription);`じデータをしますが、のオブジェクトとのをする`Region`のオブジェクトをします。

.NETのバイブルでのライブデモ

な

`IEnumerable`からのをします。は、デフォルトのをしてされます。

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

カスタムデータをするには、 `IEquatable<T>`インターフェイスをし、 `GetHashCode`メソッドと`Equals`メソッドをするがあります。または、をオーバーライドすることもできます。

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

グループ1つまたはのフィールド

々はいくつかのフィルムモデルをっているとします

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

カテゴリのプロパティ

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
}
```

カテゴリとのグループ

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

さまざまなLinqメソッドでRangeをう

Linqクエリとんでクラスをして、ループをLinqの1つのライナーにできます。

これにする

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

あなたはこれをうことができます

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

このでは、100のがされ、それらもされます

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

クエリのけ - OrderByThenByOrderByDescendingThenByDescending

```
string[] names= { "mark", "steve", "adam" };
```

クエリの

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

メソッドの

```
var sortedNames = names.OrderBy(name => name);
```

sortedNamesには、「adam」、「mark」、「steve」のにがまれています。

クエリの

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

メソッドの

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNamesには、 "steve"、 "mark"、 "adam"

のフィ―ルドでする

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil" , LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 15}
};
```

クエリの

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

メソッドの

```
 sortedPeople = people.OrderBy(person => person.LastName)
                      .ThenBy(person => person.FirstName)
                      .ThenByDescending(person => person.Age);
```

```
1. Adam Ackerman 29
2. Adam Ackerman 15
```

```
3. Phil Collins  28
4. Steve Collins 30
```

LINQは、コレクションまたはをするのにいにです。

たとえば、のサンプルデ―タがあるとします。

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true  },
    new Student { Name = "Bob",   Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true  },
    new Student { Name = "Greg",  Grade = 90, HasSnack = false },
    new Student { Name = "Joe",   Grade = 59, HasSnack = false }
}
```

LINQをしてこのデ―タを「」することができます。たとえば、スナックをっているすべてのをするには

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

または、90ののをし、な`Student`オブジェクトではなく、のみをす

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

LINQは、じをする2つのでされ、ほぼじパフォ―マンスをちますが、になったきをしています。ののは、 クエリとばれます 。ただし、のでは、 メソッドのをしています 。のとじデ―タがされますが、クエリがきまれるはなります。

```
var topStudentNames = classroom.Students
                              .Where(s => s.Grade >= 90)
                              .Select(s => s.Name);
```

GroupBy

GroupByは、の`IEnumerable<T>`コレクションをなるグル―プにするなです。

な

こののでは、との2つのグル―プになります。

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items
```

```
foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.Write(item); // 135792468  (first odd then even)
    }
}
```

よりな

としてにのリストをグループしましょう。まず、との2つのプロパティをつPersonオブジェクトをします。

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

に、々なとののサンプルリストをします。

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

に、LINQクエリをして、のリストをにグループします。

```
var query = people.GroupBy(x => x.Age);
```

そうすることで、グループのをし、グループののリストをることができます。

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

これにより、のようながられます。

```
20
Mouse
30
Neo
Trinity
40
```

```
Morpheus
Dozer
Smith
```

.NET Fiddleのライブデモでふことができます

どれか

`Any`は、コレクションのがにするかどうかをチェックするためにされます。
.All 、 *Any*、FirstOrDefaultベストプラクティスもしてください。

1.のパラメータ

Any コレクションにが`true`は`true`し、コレクションがのは`false`します。

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

2.パラメータとしてのラムダ

Any ラムダのをたすがコレクションに1つ`true`は`true`します。

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

3.のコレクション

Any コレクションがでラムダがされているは`false`します。

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

`Any`は、にするをつけるとすぐにコレクションのをします。つまり、コレクションがずしもにされるわけではありません。にするのをつけるためににされます。

.NET Fiddleのライブデモ

ToDictionary

`ToDictionary()` LINQメソッドは、された`IEnumerable<T>`ソースについて`Dictionary<TKey, TElement>`コレクションをするためにできます。

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

このでは、 `ToDictionary`されるのは、のキーをす`Func<TSource, TKey>`です。

これは、のをするなです。

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

2のパラメータを`Func<TSource, TElement>`の`ToDictionary`メソッドにして、エントリにする`Value`をすこともできます。

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

キーをするためにされる`IComparer`をすることもできます。これは、キーがで、とをしないにです。

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsenstiveName = users.ToDictionary(x => x.Name,
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsenstiveName["john"];
var user2 = usersByCaseInsenstiveName["JOHN"];
user1 == user2; // Returns true
```

`ToDictionary`メソッドでは、すべてのキーがであるがあります。するキーはしません。する、がスローされます`ArgumentException: An item with the same key has already been added.`じキーをつのがあることがわかっているシナリオがあるは、わりに`ToLookup`をするがよいでしょう。

`Aggregate`アキュムレータをシーケンスにします。

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- のステップで`prevSum = 1`
- 2の`prevSum = prevSum(at the first step) + 2`
- iのステップ`prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);
// joinedString = "Hello World !"
```

`Aggregate`2のオーバーロードはまた、アキュムレータである`seed`パラメータをける。これは、コレ

クションをすることなく、コレクションののをするためにできます。

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

したい`items`のコレクションについて

1. `.Count`
2. の
3. をする

`Aggregate`をうと、のようにすることができます

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },
                (accumelative,item) =>
                new {
                    Total = accumelative.Total + 1,
                    Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),
                    FourthItems = (accumelative.Total + 1)%4 == 0 ?
                        new List<int>(accumelative.FourthItems) { item } :
                        accumelative.FourthItems
                });
// Result:
// Total = 12
// Even = 6
// FourthItems = [4, 8, 12]
```

シ―ドとしてをすると、プロパティがみりであるため、オブジェクトをしいオブジェクトでインスタンスするがあることにしてください。カスタムクラスをすると、にをりてることができ、`new`はありませんの`seed`パラメ―タ

Linqクエリでをするキ―ワ―ドをする

linqのでをするには、 **let**キ―ワ―ドをします。これは、、サブクエリのをするためにされます。たとえば、のようになります。

```
 int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

 var aboveAverages = from number in numbers
                     let average = numbers.Average()
                     let nSquared = Math.Pow(number,2)
                     where nSquared > average
                     select number;

 Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

 foreach (int n in aboveAverages)
 {
   Console.WriteLine("Query result includes number {0} with square of {1}.", n,
Math.Pow(n,2));
 }
```

のは4.5です。

クエリには、9の2で3がまれます。
クエリ—には、4のをむ4がまれます。
クエリには、5のをむ5がまれます。
クエリには、6のをむ6がまれます。
せには7のの7がまれます。
せには、8のをむ8がまれます。
クエリには、9のがまれます。

デモをる

SkipWhile

`SkipWhile()`は、にになるまでをするためにされますこれはほとんどのです

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

DefaultIfEmpty

DefaultIfEmptyは、シ—ケンスにがまれていない、デフォルトのをすためにされます。このは、そののまたはそののユ—ザ—インスタンスです。

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
     .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
        .DefaultIfEmpty().First(); // returns null;
```

の

`DefaultIfEmpty`すると、のLinq Joinはがつからないにデフォルトのオブジェクトをすことができます。したがって、SQLのとしてします。

```
var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };
```

```
foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

ouput:

Num = 99         Char = c
Num = 100        Char = d
Num = 5          Char = ?
Num = 20         Char = ?
Num = 102        Char = ?
Num = 105        Char = i
```

`DefaultIfEmpty`がデフォルトをせずにされ、しいシーケンスにするがない、プロパティにアクセスするにそのオブジェクトが`null`ないことをするがあります。それのは、 `NullReferenceException`がします。

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
    {
        new { Value = 1 },
        new { Value = 2 },
        new { Value = 3 },
        new { Value = 4 },
    };

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
                      {
                          Left = l,
                          // 5 will not have a matching object in the right so result
                          // will be equal to null.
                          // To avoid an error use:
                          //    -  C# 6.0 or above - ?.
                          //    -  Under           - result == null ? 0 : result.Value
                          Right = result?.Value
                      }).ToList();
```

SequenceEqual

`SequenceEqual`は、2つの`IEnumerable<T>`シーケンスをいにするためにされます。

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

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

カウントとロングカウント

`Count`は、 `IEnumerable<T>`ののをします。 `Count`は、カウントするをフィルタリングできるオプションのパラメ―タもしています。

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount`じようにします`Count`が、のりのがある`long`してカウントするためにされている`IEnumerable<T>`よりいシ―ケンス`int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater
than 100
```

クエリをにする

LINQはをするため、にはをまず、されたときにをすクエリオブジェクトをつことができます。したがって、フロ―についてクエリをにし、したらすることができます。

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int
count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
```

きでフィルタをすることができます。

```
    if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
        query = query.Where(v => search.Years.Contains(v.Year));

    if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Makes.Contains(v.Make));
    }

    if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Models.Contains(v.Model));
    }

    if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
        decimal minCylinders = 0;
        decimal maxCylinders = 0;
        switch (search.Cylinders) {
            case "2-4":
                maxCylinders = 4;
```

```
                break;
            case "5-6":
                minCylinders = 5;
                maxCylinders = 6;
                break;
            case "8":
                minCylinders = 8;
                maxCylinders = 8;
                break;
            case "10+":
                minCylinders = 10;
                break;
        }
        if (minCylinders > 0) {
            query = query.Where(v => v.Cylinders >= minCylinders);
        }
        if (maxCylinders > 0) {
            query = query.Where(v => v.Cylinders <= maxCylinders);
        }
    }
```

についてクエリにソートをできます。

```
    switch (search.SortingColumn.ToLower()) {
        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
```

クエリは、のポイントからするようにできます。

```
    query = query.Skip(start - 1);
```

のレコードをすようにされています。

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
}
```

クエリオブジェクトを`ToList`、`foreach`ループ、または`ToList`や`ToArray`などののをするLINQメソッドのいずれかをしてをできます。

```
SearchModel sm;
```

```
// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

ジップ

Zipメソッドは2つのコレクションにします。これは、について2つののをにペアにします。 Funcインスタンスでは、 Zipをして2つのCコレクションのをペアでします。シリーズのサイズがなる、きなシリーズのなはされます。

「Cin a Nutshell」というをにると、

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

3 = 3
5 = 5
7 = 7

デモをる

をつGroupJoin

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };
```

ElementAtおよびElementAtOrDefault

ElementAtはインデックスnのをします。 nがのにないは、 ArgumentOutOfRangeExceptionスローします。

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

ElementAtOrDefaultはインデックスnのをします。 nがのにないは、 default(T)します。

```
int[] numbers  = { 1, 2, 3, 4, 5 };
```

```
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

ElementAtとElementAtOrDefaultは、ソースがIList<T>にされ、そのようなにはのインデックスがされます。

ElementAt、されたインデックスがIList<T>サイズよりきい、リストはArgumentOutOfRangeExceptionスローするがありますただし、にはされません。

Linq

は、シーケンスののまたはすべてがをたすにブールをします。このでは、これらのをできるなLINQ to Objectsシナリオをいくつかていきます。 LINQでできる3つのがあります。

All - シーケンスのすべてのがをたすかどうかをするためにされます。えば

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

Any - シーケンスのがをたすかどうかをするためにされます。えば

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

Contains - シーケンスにされたがまれているかどうかをするためにさContains。 えば

```
//for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
string[] query={"Tom","grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
stringValue.Contains("h");
```

のシーケンスにする

エンティティCustomer 、 Purchaseおよ びPurchaseItemをのようにえます。

```
public class Customer
```

```
{
   public string Id { get; set } // A unique Id that identifies customer
   public string Name  {get; set; }
}

public class Purchase
{
   public string Id { get; set }
   public string CustomerId {get; set; }
   public string Description { get; set; }
}

public class PurchaseItem
{
   public string Id { get; set }
   public string PurchaseId {get; set; }
   public string Detail { get; set; }
}
```

のエンティティのサンプルデータをのようにえてみましょう。

```
var customers = new List<Customer>()
 {
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },

    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
 };

 var purchases = new List<Purchase>()
 {
     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase2"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[1].Id,
         Description = "Customer2-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[1].Id,
         Description = "Customer2-Purchase2"
     }
  };
```

```
var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};
```

、linqクエリのでしてください

```
var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId     // second join
             select new
             {
                c.Name, p.Description, pi.Detail
             };
```

のクエリのをするには

```
foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

クエリのはのようになります。

1、1 - 1、1 - 1

1、1 - 2、2 - 1

1、1 - 2、2 - 2

2、2 - 2、3 - 1

.NET Fiddleのライブデモ

のキーにする

```
  PropertyInfo[] stringProps = typeof (string).GetProperties();//string properties
  PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties();//stringbuilder
properties

    var query =
        from s in stringProps
        join b in builderProps
            on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
        select new
        {
            s.Name,
            s.PropertyType,
            StringToken = s.MetadataToken,
            StringBuilderToken = b.MetadataToken
        };
```

のjoinは、すべてのプロパティがしいにのみオブジェクトがしいとみなされるため、じプロパティをむがあることにしてください。それの、クエリはコンパイルされません。

Funcで selector - のランキングをするためにします。

Selectメソッドのオーバーロードでは、 Selectれているコレクションののアイテムのindexもされselect。 これらは、そのいくつかのです。

アイテムの「」をする

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

グループのアイテムのランクをする

```
var rankInGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .SelectMany(group => group.OrderBy(item => item.Property2)
                                                      .ThenByDescending(item => item.Property3)
                                                      .Select((item, index) => new
                                                      {
                                                          Item = item,
                                                          RankInGroup = index
                                                      })).ToList();
```

グループのランキングをするOracleでdense_rankとしてもられている

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
```

```
                            .Select((group, index) => new
                            {
                                Items = group,
                                Rank = index
                            })
                            .SelectMany(v => v.Items, (s, i) => new
                            {
                                Item = i,
                                DenseRank = s.Rank
                            }).ToList();
```

これをテストするには、をできます

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

そしてデータ

```
List<SomeObject> collection = new List<SomeObject>
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}
};
```

TakeWhile

`TakeWhile`は、があるり、シーケンスからをします

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

`Enumerable.Sum`メソッドは、のをします。

コレクションのがの、をすることができます。

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

のがなのは、ラムダをしてするがあるをできます。

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Sumメソッドは、のタイプでできます。

- Int32
- Int64
- シングル
- ダブル
-

コレクションにnullがまれているは、null-coalescingをして、nullのデフォルトをできます。

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

げる

> ToLookupは、インデックスをにするデータをします。これはメソッドです。これは、foreachループをしてけまたはできるILookupインスタンスをします。エントリは、キーのグループにまとめられます。 - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",",lookup[3]));
//output: one,two
```

もうつの

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",",lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(",",lookup[1]));
//output: 1,3,5,7
```

IEnumerableののLinqをする

Linqのきなの1つは、それがとてもにできることです。が`IEnumerable<T>`であるメソッドをするだけです。

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

このでは、`IEnumerable<T>`のをサイズのリストにします。のリストにはアイテムのりのがまれています。メソッドがされるオブジェクトが、`this`キーワードをしてのとしての`source` どのようにされるかにしてください。`yield`キーワードは、`IEnumerable<T>`ののをするにされます yieldキーワード。

このは、あなたのコードでのようにされます

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

のループでは、サブリストは`{2, 3, 4}`、2の`{5, 6}`ます。

カスタムLinQメソッドは、のLinQメソッドとみわせることもできます。えば

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)         // generate a list
                       .Where(x => x%2 == 0) // filter the list or do something other
                       .Batch(3)             // call our extension method
                       .ToList()             // call other standard methods
```

このクエリは、サイズが`{0, 2, 4}, {6, 8, 10}, {12}`バッチでグループされたをします`{0, 2, 4}, {6, 8, 10}, {12}`

`using MyNamespace;`するがあること`using MyNamespace;`メソッドにアクセスできるようにします。

ネストされたループのわりにSelectManyをする

2つのリスト

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

すべてのをしたいは、のようなネストされたループをできます。

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

SelectManyをすると、

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Any and FirstOrDefault - ベストプラクティス

は`Any`と`FirstOrDefault`がをしているのかについてはしません。については、 AnyおよびFirst、FirstOrDefault、Last、LastOrDefault、SingleおよびSingleOrDefaultをしてください。

コードでよくられるパターンは、けるべきです。

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

このようににくことができます

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

2のをすると、コレクションは1だけされ、のコレクションとじがられます。じえを`Single`もできます。

GroupByとカウント

サンプルクラスをってみましょう

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

さて、トランザクションのリストをえてみましょう

```
var transactions = new List<Transaction>
{
   new Transaction { Category = "Saving Account", Amount = 56, Date =
DateTime.Today.AddDays(1) },
   new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-
10) },
   new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1)
},
   new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
   new Transaction { Category = "Current Account", Amount = 100, Date =
DateTime.Today.AddDays(5) },
};
```

とカウントのカテゴリをするは、GroupByをのようにできます。

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
                           .Select(t => new
                           {
                               Category = t.Key,
                               Count = t.Count(),
                               Amount = t.Sum(ta => ta.Amount),
                           }).ToList();

Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
            row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

あるいは、これを1ステップですることもできます。

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
        var transactionArray = t as Transaction[] ?? t.ToArray();
        return new
        {
            Category = key,
            Count = transactionArray.Length,
            Amount = transactionArray.Sum(ta => ta.Amount),
        };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

ののクエリのはじになります

カテゴリアカウントをめる、66、2

カテゴリクレジットカード、71、カウント2

カテゴリ、100、1

.NET Fiddleのライブデモ

- シーケンスののをします。
- がない、 `ArgumentNullException: source is null.`スローされます`ArgumentNullException: source is null.`

```
// Create an array.
int[] array = { 1, 2, 3, 4 };                          //Output:
// Call reverse extension method on the array.         //4
var reverse = array.Reverse();                         //3
// Write contents of array to screen.                  //2
foreach (int value in reverse)                         //1
    Console.WriteLine(value);
```

ライブコードの

Remeberは、LINQのチェーンにじて`Reverse()`がなることがあります。

```
        //Create List of chars
        List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

        //Reversing the list then taking the two first elements
        IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

        //Taking 2 elements and then reversing only thos two
        IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

        //reverseFirst output: 6, 5
        //reverseLast output:  2, 1
```

ライブコードの

*Reverse*は、すべてをバッファリングしてにしますが、ではありませんが、どちらもOrderByではありません。

LINQ-to-Objectsには、バッファリングReverse、OrderBy、GroupByなどとバッファリングWhere、Take、Skipなどがあります。

バッファリングしない

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

ライブコードの

このメソッドは、にリストをするとがするがあります。

Enumerableの

IEnumerable <T>インターフェイスは、すべてののインターフェイスであり、LINQをするなです。そのに、それはシーケンスをします。

このなインターフェイスは、 Collection <T> 、 Array 、 List <T> 、 Dictionary <TKey、TValue> クラス 、およびHashSet <T>など、すべてのコレクションにされます。

シーケンスをすことにえて、IEnumerable <T>からするクラスはIEnumerator <T>をするがあります。は、のイテレータをし、これらの2つのされたインタフェースとアイデアは、「な」のソースです。

""はなフレーズです。はにするためのであり、マテリアライズされたオブジェクトをしません。たとえば、ソートするとき、はソートするフィールドのをすることができますが、 `.OrderBy()` をでするとソートのみをるIEnumerable <T>がされます。オブジェクトをするびしをすると、をするときのように、としてられていますたとえば`.ToList()`。 プロセスは、シリーズをしてオブジェクト、フィルタ、などをすために、 どのようになをします。

がされると、オブジェクトのがわれます。これは、 のさ シリーズのサイズにする やのさシリーズのサイズにしてどれくらいのスペースをするがあるかされる。

IEnumerable <T>からするのクラスをするには、なになるにじてしにすることができます。に、のジェネリックコレクションの1つをすることがです。つまり、としてされたをたずにIEnumerable <T>インターフェイスをすることもです。

たとえば、フィボナッチをになるシーケンスとしてします。 `Where`のびしはに`IEnumerable`し、のいずれかがマテリアライズされるようにできることをするまではありません。

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }

    //Enumerable called typically from linq
```

```
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

2のフィッティングfibMod612のさは、フィボナッチのをするびしをったにもかかわらず、`.First()`をって1つのしかられなかったため、のさはOnけアルゴリズムのにされるがあった。これは、われわれのが1つのしかめていないため、をするがないからです。しなければならなかった`.First() .Take(5)`わりに`.Take(5)`をすると、は5つのをし、5つのを`.First()`するがあります。のセットをしてからの5つのをることがとすると、のは、とくのスペースをできます。

OrderBy

されたでコレクションをオーダーします。

が 、 またはのは、 でまります 。つまり、にのがられます。ゼロはのよりもきくなります1を。

charでると、このメソッドは**char**の*ascii*をしてコレクションをソートします2を。

をべえると、OrderByメソッドはCultureInfoをべることによってそれらをしますが、アルファベットののa、b、c ...からまっています。

このようなはとばれ、のはにするがありますOrderByDescendingを。

1

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

2

```
 char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
 IEnumerable<char> ascending = letters.OrderBy(x => x);
 // returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', '{' }
```

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob
```

OrderByDescending

されたでコレクションをオーダーします。

が、またはのは、からします。つまり、ゼロとそれのよりものがにられます1を。

charですると、このメソッドは**char**の*ascii*をしてコレクションをソートします2を。

をべえると、OrderByメソッドはCultureInfoをべることによってそれらをしますが、アルファベットz、y、x、...ののからまっています。

このようなはとばれ、のはにするがありますOrderBy。

1

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

2

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }
```

3

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
```

```
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

2つのコレクションをマージしますをしません。

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

MSDN

された`IEqualityComparer<T>`をしてシーケンスにされたがまれるかどうかをします。

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true
only for 4,5
```

ユーザーオブジェクトの

```
public class Person
{
   public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki"},
    new Person { Name = "Gilad"},
    new Person { Name = "Phil"},
    new Person { Name = "John"}
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

`Enumerable.Contains(value, comparer)`オーバーロードの

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
```

```
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

Containsスマートなは、の**if**を**Contains**びしにきえることです。

だからこれをするわりに

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

これをう

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

オンラインでLINQクエリをむ https://riptutorial.com/ja/csharp/topic/68/linqクエリ

40: Microsoft.Exchange.WebServices

Examples

したユーザーののをする

に、コンストラクタがたちのためにサービスにする`ExchangeManager`オブジェクトをしましょう。`GetOofSettings`メソッドもあり、されたメールアドレスの`OofSettings`オブジェクトをします。

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password =
WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return
redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

これをのようにのでびすことができます

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

のユーザーののをする

のクラスをして、Exchangeにし、のユーザーのUpdateUserOofのをUpdateUserOofできます。

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

ユーザーをのようにします。

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage,
```

```
externalMessage, theUser);
```

の`html`タグをしてメッセージをフォーマットすることができます。

オンラインでMicrosoft.Exchange.WebServicesをむ
https://riptutorial.com/ja/csharp/topic/4863/microsoft-exchange-webservices

41: Nullable

- `Nullable<int> i = 10;`
- int j = 11;
- int k = null;
- DateOfBirth = DateTime.Now;
- = 1.0m;
- ブール IsAvailable = true;
- チャー= 'a';
- タイプ variableName

Nullableは、のすべてのと`null`をすことができ`null`。

`T?` `Nullable<T>`です。

null `System.ValueType`なはには`System.ValueType`オブジェクトなので、ボックスおよびボックスすることができます。また、 `null`オブジェクトの`null`はオブジェクトの`null`とじではなく、なるフラグです。

nullなオブジェクトboxingの、nullは`null`にされ、 `null`のはnullなにされます。

```
DateTime? dt = null;
var o = (object)dt;
var result = (o == null); // is true

DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var dt2 = (DateTime)dt; // correct cause o contains DateTime value
```

2のルールはしいがなコードにつながります

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

いで

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

Examples

nullableをする

`null`の

```
Nullable<int> i = null;
```

または

```
int? i = null;
```

または

```
var i = (int?)null;
```

nullのの

```
Nullable<int> i = 0;
```

または

```
int? i = 0;
```

Nullableにがあるかどうかをする

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

どちらがじです

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

nullなのをする

えられたnullな`int`

```
int? i = 10;
```

デフォルトがなは、 null `GetValueOrDefault`をしてりてるか、にnullable int `HasValue`チェックします

。

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

のはにではありません。にiがnullの、 System.InvalidOperationExceptionがスローされます。に、がされていなければ、 Use of unassigned local variable 'i'します。

```
int j = i.Value;
```

nullなからデフォルトをする

.GetValueOrDefault()メソッドは、 .HasValueプロパティがfalseのでもをしますをスローするValueプロパティとはなります。

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

```
0
1
```

ジェネリックパラメータがヌルなかどうかをチェックする

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

ヌルのデフォルトはnullです。

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
```

```
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

ヌル

となるNullableのな

nullなはすべてです。 nullなはです。

リフレクション /コードのにするコードをするときに、 Nullable.GetUnderlyingTypeメソッドのをにできるようにするいくつかのがあります。

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsNullable())
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsNullable(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
```

```
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS。 `NullableTypesCache`はのようにされています。

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type,
Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

オンラインでNullableをむ https://riptutorial.com/ja/csharp/topic/1240/nullable

42: NullReferenceException

Examples

NullReferenceExceptionがされました

NullReferenceExceptionは、オブジェクトのメンバープロパティ、メソッド、フィールドまたはイベントにアクセスしようとするとスローされますが、nullです。

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

このようなをデバッグするには、にです。がスローされたでは、すべてのをべるだけ.'または'['、まれに('。

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

のはどこからたのですかどちらか

- myGarageはnull
- myGarage.CarCollectionがnull
- currentIndexがnull
- myGarage.CarCollection[currentIndex.Value]はnull
- theCarInTheStreetはnull

デバッグモードでは、これらののそれぞれにマウスカーソルをくだけで、nullがつかります。それで、あなたがしなければならないのは、それががないをすることです。は、あなたのメソッドのににします。

それをインスタンス/するのをれましたか

```
myGarage.CarCollection = new Car[10];
```

オブジェクトがnullのは、かうことをやろうとしていますか

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

あるいは、かがあなたにヌルをえたとしたら、

```
if (theCarInTheStreet == null)
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

いずれのでも、メソッドはしてNullReferenceExceptionをスローしないようにしてください。そうであれば、それはあなたがかをすることをれたことをします。

オンラインでNullReferenceExceptionをむ
https://riptutorial.com/ja/csharp/topic/2702/nullreferenceexception

43: Onののためのアルゴリズム

き

プログラミングをするのでは、としてすべきでいがありました。これらのの1つは、またはのコレクションをのだけさせることでした。ここではそれをうなをあなたとします。

Examples

えられたシフトでをさせるなメソッドの

は、シフトがであるときににし、がであるときににすることをしたい。

```
    public static void Main()
    {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int shiftCount = 1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = 15;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -35;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
    }

    private static void Rotate<T>(ref T[] array, int shiftCount)
    {
        T[] backupArray= new T[array.Length];

        for (int index = 0; index < array.Length; index++)
        {
            backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
        }

        array = backupArray;
    }
```

このコーードでなことは、にしいインデックスがつかるです。

index + array.Length + shiftCountarray.Lengtharray.Length

それについてもうししくりたいです

shiftCountarray.Length - >シフトをのさにしますさ10のでは、シフト1または11はじものです。-1と-11についてもです 。

array.Length +shiftCountarray.Length - >これはのためにわれ、のインデックスにはならず、のまでさせます。インデックス0と-1のさが10のでは、の-1になり、のインデックスのは9になりません。10 +-110= 9

index + array.Length +shiftCountarray.Length - >しいインデックスをするために、インデックスにロ―テ―ションをするので、ここであまりしません。 0 + 10 +-110= 9

index + array.Length +shiftCountarray.Lengtharray.Length - > 2のは、しいインデックスがのにはてこないことをしていますが、のにをします。これはのためです。なぜなら、さ10のではインデックス9がなく、1ではのにあるインデックス10にり、のインデックスは0になりません。 + 10 +11010 = 0

オンラインでOnののためのアルゴリズムをむ https://riptutorial.com/ja/csharp/topic/9770/o-n-ののためのアルゴリズム

44: ObservableCollection

Examples

ObservableCollectionをする

ObservableCollectionは、 List<T>ようなTのコレクションです。 つまり、 Tのオブジェクトをしています。

ドキュメントから、たちはそれをんでいます

> ObservableCollectionは、アイテムの、、またはリストのリフレッシュにをするデータコレクションをします。

のコレクションとのないは、 ObservableCollection INotifyCollectionChangedおよびINotifyPropertyChangedのインターフェイスをし、しいオブジェクトのまたはおよびコレクションのクリアにイベントをすぐにさせること INotifyPropertyChanged。

これは、オブジェクトがなコレクションにされたりされたりすると、UIがにされるため、なコードをすることなくアプリケーションのUIとバックエンドをするににです。

それをするためのは、

```
using System.Collections.ObjectModel
```

たとえば、 stringのコレクションののインスタンスをするか

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

またはデータでたされたインスタンス

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

すべてのIListコレクションとに、インデックスは0 IList.Itemプロパティ からまります。

オンラインでObservableCollection をむ
https://riptutorial.com/ja/csharp/topic/7351/observablecollection--t-

45: Stacktracesをみ、する

き

スタックトレースは、プログラムをデバッグするににちます。プログラムがをスローするとスタックトレースがし、プログラムがすることがあります。

Examples

WindowsフォームでのなNullReferenceExceptionのスタックトレース

をスローするさなコードをしましょう

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

これをすると、のとスタックトレースがされます。

```
System.NullReferenceException: "Object reference not set to an instance of an object."
   at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

スタックトレースはそういうものですが、このはたちのにはです。

スタックトレースのにはのがあります。

> F\ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.csの
> WindowsFormsApplication1.Form1.button1_Clickオブジェクト、EventArgs e29

これがもなです。これは、がしたなをしています。Form1.csの29。
だから、これがあなたのをめるです。

2は

> System.Windows.Forms.Control.OnClickEventArgs e

これは`button1_Click`をびした`button1_Click`です。これで、エラーがした`button1_Click`が`System.Windows.Forms.Control.OnClick`からびされたことが`button1_Click`。

たちはこれをけることができます。 3は

System.Windows.Forms.Button.OnClickEventArgs e

これは`System.windows.Forms.Control.OnClick`をびしたコードです。

スタックトレースは、コードでExceptionがするまでびされたのリストです。これをすることで、コードがにするまでどのコードがされたかをることができます。

スタックトレースには.Netシステムからのびしがまれています。がになったのかをるために、、すべてのMicrosoftの`System.Windows.Forms`コードにうはなく、のアプリケーションにするコードだけがです。

だから、なぜこれは "スタックトレース"とばれていますか
なぜなら、プログラムがメソッドをびすたびに、プログラムはそのメソッドがどこにあったのかをするからです。それは "スタック"とばれるデータをち、のをダンプします。
メソッドのがすると、メソッドをびすのをスタックでべ、そこからします。

そこでスタックは、しいメソッドをびすに、したをコンピュータにらせます。

しかし、デバッグのけともなります。がをしたときにしたみをするように、プログラマーはスタックをって、プログラムがするのステップをすることができます。

オンラインでStacktracesをみ、するをむ https://riptutorial.com/ja/csharp/topic/8923/stacktracesをみ-する

46: String.Format

き

`Format`メソッドは、オブジェクトをのにするをするためにされる`System.String`クラスのオーバーロードのセットです。このは、.NET Frameworkの`String.Format` 、さまざまな`WriteLine`メソッド、およびそののメソッドにできます。

- string.Format、paramsオブジェクト[] args
- string.FormatIFormatProviderプロバイダ、、paramsオブジェクト[] args
- $ "string {text} blablabla" // C6

パラメーター

パラメータ	
フォーマット	。argsをにするをします。
args	にされるのオブジェクト。これは`params`をするので、のカンマりリストまたはのオブジェクトのどちらかをできます。
プロバイダ	オブジェクトをにフォーマットするのまり。なには、CultureInfo.InvariantCultureおよびCultureInfo.CurrentCultureがまれます。

ノート

- `String.Format()`は、をスローせずに`null`をし`null`。
- `args`パラメータを1つ、2つ、または3つのオブジェクトパラメータできえるオーバーロードがあります。

Examples

String.Formatがフレームワークにめまれている

に`String.Format`をできるはいくつかあります。は、`string format, params object[] args`でオーバーロードをすことです。

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

よりいバージョンにきえることができます

```
Console.WriteLine("{0} - {1}", name, value);
```

`String.Format`などののメソッドもできます。

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

カスタムをする

`NumberFormatInfo`は、とののにできます。

```
// invariantResult is "1,234,567.89"
var invarianResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);

// NumberFormatInfo is one of classes that implement IFormatProvider
var customProvider = new NumberFormatInfo
{
    NumberDecimalSeparator = "_NS_", // will be used instead of ','
    NumberGroupSeparator = "_GS_", // will be used instead of '.'
};

// customResult is "1_GS_234_GS_567_NS_89"
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

カスタムフォーマットプロバイダをする

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
        {
            return null;
        }

        if (format == "Reverse")
        {
            return String.Join("", arg.ToString().Reverse());
        }

        return arg.ToString();
    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}
```

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

```
-> dlroW olleH <-
```

I、パッドをスペースでする

の2のは、のさをします。2のをまたはにすることによって、ストリングのアラインメントをすることができる。

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

```
// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

"c"またはは、ををすにします。

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

デフォルトは2です。c1、c2、c3などをしてをします。

```
string.Format("{0:C1}", 112.236677) //$112.2
string.Format("{0:C3}", 112.236677) //$112.237
string.Format("{0:C4}", 112.236677) //$112.2367
string.Format("{0:C9}", 112.236677) //$112.236677000
```

1. カスタム`CultureInfo`シンボルをするには、 `CultureInfo`インスタンスをします。

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); //$112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); //112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); //□ 112.24
```

2. のをとしてします。 `NumberFormatInfo`を`NumberFormatInfo`て、をカスタマイズし`NumberFormatInfo`。

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
```

```
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

の

CurrencyPositivePatternをのとするためにCurrencyNegativePatternののため。

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); //$112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); //112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); //$ 112.24
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

ネガティブパターンのはポジティブパターンとじです。よりくのユースケースはのリンクをしてください。

カスタムり

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

C6.0

6.0

C6.0、 `String.Format`わりにをすることができます。

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

こんにちはJohn Doe

これにするは、C6.0のトピック「 」をしてください 。

String.Formatのにをエスケープする

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);
```

```
//Outputs "{I am in brackets!} I am outside of bracket"
```

の

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.Write(String.Format("{0:dd}",date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"),"{0:dddd}",date);
```

6.0

```
Console.Write($"{date:ddd}");
```

```
06
Лхагва
06
```

		サンプル	
d		`{0:d}`	201476
DD	ゼロデパッド	`{0:dd}`	06
ddd	の	`{0:ddd}`	した
dddd	の	`{0:dddd}`	
D	い	`{0:D}`	201676
f	なと、い	`{0:f}`	201676630
ff	2の、2	`{0:ff}`	20
fff	2の、3	`{0:fff}`	201
ffff	2の、4	`{0:ffff}`	2016
F	な	`{0:F}`	2016766:30:14 PM
g	デフォルトのと	`{0:g}`	2014766:30 PM
gg		`{0:gg}`	
hh	2、12H	`{0:hh}`	06
HH	2、24	`{0:HH}`	18

		サンプル	
M	と	`{0:M}`	76
mm	、ゼロパッド	`{0:mm}`	30
MM	、ゼロパッド	`{0:MM}`	07
うーん	3の	`{0:MMM}`	7
んー	の	`{0:MMMM}`	7
SS		`{0:ss}`	14
r	RFC1123の	`{0:r}`	、20167618:30:14 GMT
s	ソートな	`{0:s}`	2016-07-06T183014
t		`{0:t}`	6:30 PM
T	い	`{0:T}`	6:30:14 PM
tt		`{0:tt}`	PM
あなた	ユニバーサルソートな	`{0:u}`	2016-07-06 183014Z
U	ユニバーサルGMT	`{0:U}`	2016769:30:14 AM
Y	と	`{0:Y}`	20167
yy	2の	`{0:yy}`	16
yyyy	4の	`{0:yyyy}`	2016
グーグー	2のタイムゾーンオフセット	`{0:zz}`	+09
zzz	フルタイムゾーンオフセット	`{0:zzz}`	+0900

ToString

ToStringメソッドは、すべてのオブジェクトにします。これはToStringメソッドをつObjectからしたすべてのによるものです。オブジェクトクラスのToStringメソッドはをします。のはコンソールに "User"をします。

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

```
...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

ただし、クラスUserは、されるをするためにToStringをオーバーライドすることもできます。のコードは、 "Id5、NameUser1"をコンソールにします。

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

ToStringとの

String.Format()メソッドはデータをとしてするにはですが、にのようにのオブジェクトをうには、しなものになることがあります。

```
String.Format("{0:C}", money);  // yields "$42.00"
```

よりなアプローチは、CのすべてのオブジェクトでできるToString()メソッドをにすることです。じおよびカスタムをすべてサポートしますが、が1つしかないため、なパラメータマッピングはありません。

```
money.ToString("C");  // yields "$42.00"
```

との

このアプローチは、いくつかのシナリオではかもしれませんが、 ToString()アプローチは、String.Format()メソッドのように、またはのパディングをすることにしてされています。

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

ToString()メソッドでこのじをするには、 PadLeft()またはPadRight()ようなのメソッドをするがあります。

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

オンラインでString.Formatをむ https://riptutorial.com/ja/csharp/topic/79/string-format

47: StringBuilder

Examples

StringBuilderのと

StringBuilderは、のとはなり、なのをします。くの、にしたをするがありますが、のオブジェクトはできません。つまり、がされるたびに、しいオブジェクトをし、コピーしてからりてするがあります。

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

のでは、 myStringははは"Apples"というしかっていません。しかし、が「のきな」をすると、クラスはにをするがありますか

- myStringのさとするしいにしいのしいをします。
- myStringすべてのをしいのにコピーし、しいをのにコピーします。
- メモリにしいオブジェクトをし、それをmyStringりてしmyString。

のの、これはです。しかし、ループのようにくのをするがあるはどうなりますか

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

コピーとオブジェクトのがりされるため、プログラムのパフォーマンスがにします。わりにStringBuilderをすることで、これをできます。

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

じループがされると、プログラムののパフォーマンスとは、のをするよりもににになります。StringBuilderのにすには、にStringBuilder ToString()メソッドをびすだけです。

しかし、これはStringBuilderがつのではありません。をさらにするために、パフォーマンスをさせるそののプロパティをできます。

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

StringBuilderさをにっていれば、そのサイズをにすることができます。これにより、でっているのサイズをするがなくなります。

```
sb.Append('k', 2000);
```

のために`StringBuilder`をするのはよりもはるかにですが、のをもするだけであれば、さらににできます。

のがしたら、 `StringBuilder` `ToString()`メソッドをして`string`にでき`string`。 これは、`StringBuilder`クラスが`string`からしないため、しばしばです。

たとえば、 `StringBuilder`をして`string`をするをにします。

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

として、 `StringBuilder`は、のくのをパフォーマンスをにいてうがあるに、のわりにするがあります。

StringBuilderをしてのレコードからをする

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
       customerNamesCsv
            .Append(record.LastName)
            .Append(',')
            .Append(record.FirstName)
            .Append(Environment.NewLine);
    }

    return customerNamesCsv.ToString();
}
```

オンラインでStringBuilderをむ https://riptutorial.com/ja/csharp/topic/4675/stringbuilder

48: System.DirectoryServices.Protocols.LdapConnec

Examples

されたSSL LDAP、SSLがDNSとしません

サーバとのいくつかのをします。 LDAPv3をしていますが、にできます。

```
// Authentication, and the name of the server.
private const string LDAPUser =
"cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

には、LdapDirectoryIdentifierサーバーとNetworkCredentialsの3つのでをします。

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
        secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//    connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},",
TargetServer));
```

LDAPサーバーをします。たとえば、すべてのobjectClassにしてuseridでかをします。コンパウンドをすためにobjectClassがします。アンパサンドは、2つのクエリーのブール「and」です。

```
 SearchRequest searchRequest = new SearchRequest(
        CompanyDN,
        string.Format((&(objectClass=*)(uid={0})), uid),
        SearchScope.Subtree,
        null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

スーパーシンプルLDAP

LDAPv3をしていますが、にできます。これはのされていないLDAPv3 LdapConnectionのです。

```
private const string TargetServer = "ldap.example.com";
```

には、LdapDirectoryIdentifierサーバーとNetworkCredentialsの3つのでをします。

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

をするには、このようなものは、スミス

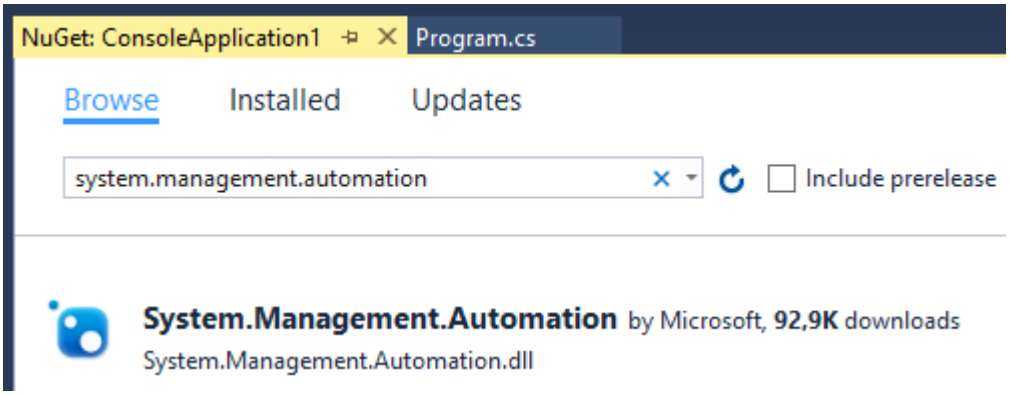
```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

オンラインでSystem.DirectoryServices.Protocols.LdapConnectionをむ
https://riptutorial.com/ja/csharp/topic/5177/system-directoryservices-protocols-ldapconnection

49: System.Management.Automation

*System.Management.Automation*は、Windows PowerShellのルートです。

System.Management.AutomationはMicrosoftのライブラリであり、NuGetパッケージマネージャまたはパッケージマネージャコンソールからVisual Studioプロジェクトにできます。



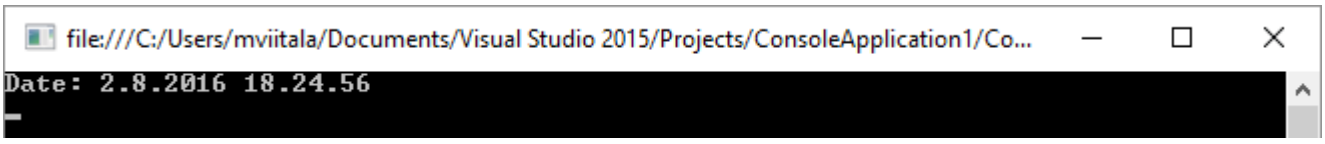

```
PM> Install-Package System.Management.Automation
```

Examples

シンプルなパイプラインをびす

のとをします。

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



オンラインでSystem.Management.Automationをむ

https://riptutorial.com/ja/csharp/topic/4988/system-management-automation

50: T4コード

- **T4**
- `<#@...#>` //テンプレート、アセンブリ、、およびテンプレートがするをむプロパティの
- `Plain Text` //されたファイルにしてループスルーできるテキストをする
- `<#=...#>` //スクリプトの
- `<#+...#>` //スクリプトレットをする
- `<#...#>` //テキストブロックをする

Examples

コードの

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

オンラインでT4コードをむ https://riptutorial.com/ja/csharp/topic/4824/t4コード

51: Windows Communication Foundation

き

Windows Communication FoundationWCFは、サービスアプリケーションをするためのフレームワークです。 WCFをすると、あるエンドポイントからのエンドポイントにメッセージとしてデータをできます。サービスエンドポイントは、IISによってホストされるになサービスのであるもあれば、アプリケーションでホストされているサービスであるもあります。メッセージは、XMLとしてされるのやのようになものでも、バイナリデータのストリームのようななものでもかまいません。

Examples

めのサンプル

このサービスは、メタデータとしてされるサービスでされるをします。

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

サービスは、のコードにすように、なをしてします。

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

このサービスは、のにすように、ファイルWeb.configをしてされた、サービスとするためのエンドポイントをします。

```
<services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
        <!-- ICalculator is exposed at the base address provided by
         host: http://localhost/servicemodelsamples/service.svc.  -->
       <endpoint address=""
              binding="wsHttpBinding"
              contract="StackOverflow.ServiceModel.Samples.ICalculator" />
```

```
      ...
    </service>
</services>
```

フレームワークはデフォルトでメタデータをしません。そのため、サービスは ServiceMetadataBehaviorをオンにし、 http//localhost/servicemodelsamples/service.svc/mexでメタデータエクスチェンジMEXエンドポイントをします。のでこれがされています。

```
<system.serviceModel>
  <services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is explosed at
       http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
   attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

クライアントは、ServiceModelメタデータユーティリティツールSvcutil.exeによってされたクライアントクラスをして、のコントラクトタイプをしてします。

きプロキシをするには、クライアントディレクトリのSDKコマンドプロンプトからのコマンドをします。

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

サービスとに、クライアントはファイルApp.configをして、するエンドポイントをします。クライアントのエンドポイントは、のにすように、サービスエンドポイントのアドレス、バインド、およびでされています。

```
<client>
     <endpoint
         address="http://localhost/servicemodelsamples/service.svc"
         binding="wsHttpBinding"
         contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

クライアントはクライアントをインスタンスし、のコードにすようにきインターフェイスをしてサービスとのをします。

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

オンラインでWindows Communication Foundationをむ
https://riptutorial.com/ja/csharp/topic/10760/windows-communication-foundation

52: Windowsフォームアプリケーションでの MessageBoxをする

き

に、MessageBoxがであるかをるがあります...

MessageBoxコントロールは、されたテキストのメッセージをし、カスタムイメージ、タイトル、およびボタンセットこれらのボタンセットは、ユーザーがなyes / noのものをできますをすることによってカスタマイズできます。

のMessageBoxをすることで、されたDLLをするか、クラスをむファイルをコピーするだけで、しいアプリケーションでMessageBoxコントロールをできます。

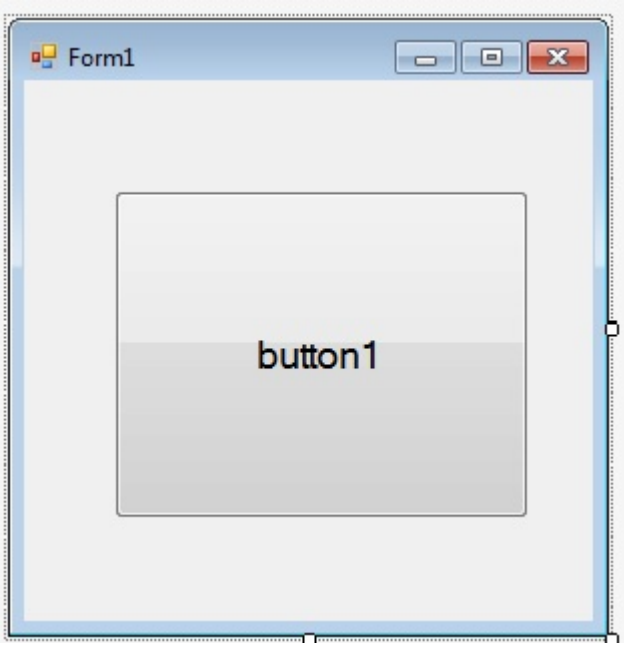
- 'DialogResult result = DialogResult.No; // DialogResultは、にダイアログによってされます。

Examples

の**MessageBox**コントロールの。

のMessageBoxコントロールをするには、のガイドにってください。

1. Visual Studioのインスタンスをく VS 2008/2010/2012/2015/2017
2. のツールバーにして、「ファイル」->「プロジェクト」->「Windows Formsアプリケーション」->プロジェクトをし、「OK」をクリックします。
3. ロードされたら、ツールボックスのをからフォームにボタンコントロールをドラッグアンドドロップします。



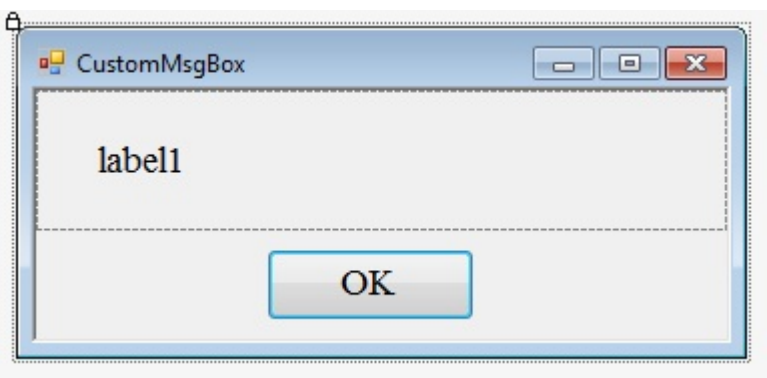
4. ボタンをダブルクリックすると、によってにクリックイベントハンドラがされます。

5. フォームのコードをして、のようにしますフォームをクリックして[コードの]をクリックします。

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
           CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. ソリューションエクスプローラ - >プロジェクトをクリック - > - > Windowsフォームをし、を "CustomMsgBox.cs"にします。

7. ボタンラベルコントロールをツールボックスからフォームにドラッグしますフォームをしたは、のようになります。



8. しくしたフォームにのコードをいてください。

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}
```

9. F5キーをすだけでプログラムをできます。おめでとう、あなたはなコントロールをりました。

のしたMessageBoxコントロールをのWindowsフォームアプリケーションでする

の.csファイルをつけるには、Visual Studioのインスタンスでプロジェクトをクリックし、ファイルエクスプローラでフォルダをくをクリックします。

1. Visual Studio - >のプロジェクトWindowsフォーム - >ソリューションエクスプローラ - >プロジェクト - >クリック - > - >アイテム - >の.csファイルをします。

2. は、コントロールをするためににうことがあります。コードにusingステートメントをして、アセンブリがそのをできるようにします。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
.
.
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. メッセージボックスをするには、をしてください...

   CustomMsgBox.Show "メッセージボックスのメッセージ..."、 "MSG"、 "OK";

オンラインでWindowsフォームアプリケーションでのMessageBoxをするをむ
https://riptutorial.com/ja/csharp/topic/9788/windowsフォームアプリケーションでのmessageboxをする

53: XDocumentとSystem.Xml.Linq

Examples

XMLをする

は、のXMLドキュメントをすることです。

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

コード

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

XMLファイルの

XMLファイルをするには`XDocument`、あなたはのにファイルをロードする`XDocument`、メモリにそれをし、それをし、のファイルをきします。よくあるいは、メモリのXMLをし、ディスクのファイルがされることをすることです。

えられたXMLファイル

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

バナナのをにしたい

1. ディスクのファイルへのパスをるがあります。
2. `XDocument.Load` 1つのオーバーロードはURIファイルパスをけります。
3. xmlファイルはをするため、とをしてクエリをするがあります。
4. ヌルのにするためにC6をするLinqクエリ。すべて.がをしない、このクエリではnullセットをすがあります。 C6よりには、のステップでこれをい、でnullをチェックしてください。は、バナナをむ`<Fruit>`です。には`IEnumerable<XElement>`。 そのため、のステップで`FirstOfDefault()`されます。
5. はつかったFruitからFruitColorをします。ここでは、ただつしかないとします。あるいは、のものだけをにします。
6. nullでないは、FruitColorを「Brown」にします。
7. に、 `XDocument`をし、のファイルをディスクにきします。

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
    Elements(ns + "FruitName")?.
    Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
}

// 7.
xdoc.Save(xmlFilePath);
```

ファイルはのようになります。

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Brown</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

なをしてXMLをする

ゴール

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

コード

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

オンラインでXDocumentとSystem.Xml.Linqをむ
https://riptutorial.com/ja/csharp/topic/1866/xdocumentとsystem-xml-linq

54: XmlDocumentとSystem.Xml

Examples

なXMLの

```
public static void Main()
{
    var xml  = new XmlDocument();
    var root = xml.CreateElement("element");
        // Creates an attribute, so the element will now be "<element attribute='value' />"
        root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
            day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

XMLからのみみ

XMLファイルの

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
    </Sample>
```

このXMLファイルからのみみ

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

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();

        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

XmlDocumentとXDocumentと

Xmlファイルとするにはいくつかのがあります。

1. XML
2. XDocument
3. XmlReader / XmlWriter

LINQ to XMLのに、、などをするようなXMLのためにXMLDocumentがされました。 LINQ to XMLは、XDocumentをじのものにします。はXMLDocumentよりはるかにで、のコーードしかとしません。

また、XDocumentはXmlDocumentのようにです。 XmlDoucumentは、XMLドキュメントをクエリするためのくていソリューションです。

XmlDocumentクラスと**XDocument**クラスクラスのをいくつかします。

XMLファイルをみむ

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

XmlDocumentをする

XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
 XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

XMLのノードの**InnerText**をする

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

のファイルの

はずxmlをにしてください。

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

XMLからのリトリーブ

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

すべてのからをし、attribute = somethingのをします。

XmlDocument

```
List<string> valueList = new List<string>();
    foreach (XmlNode n in nodelist)
    {
        if(n.Attributes["type"].InnerText == "City")
        {
            valueList.Add(n.Attributes["type"].InnerText);
        }
    }
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

ノードをする

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to safe the document*/
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new
XElement("SecondLevelNode"));

 /*After a change make sure to safe the document*/
_doc.Save(fileName);
```

オンラインでXmlDocumentとSystem.Xmlをむ
https://riptutorial.com/ja/csharp/topic/1528/xmldocumentとsystem-xml

55: XMLドキュメントのコメント

いくつかのは、xmlコメントからテキストをするがあります。にも、なはありません 。

しかし、このにできるいくつかの々のプロジェクトがあります

- のお
- Docu
- NDoc
- DocFX

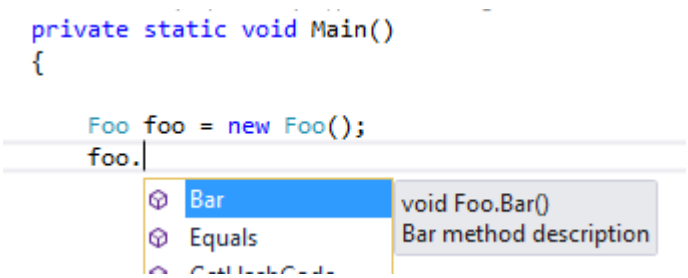
Examples

なメソッド

ドキュメントコメントは、それらがするメソッドまたはクラスのすぐにされます。らは3つのスラッシュでまる`///` 、およびメタがXMLですることができます。

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

}
```

タグのは、Visual StudioなどのツールでintelliSenseなどのサービスをするためにできます。



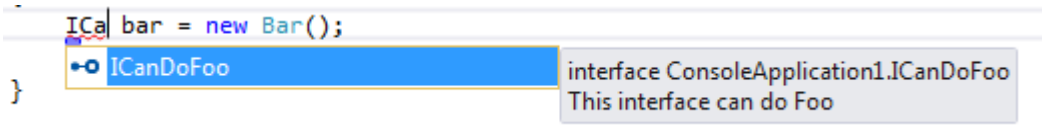
Microsoftのなドキュメントタグのリストもしてください。

インタフェースとクラスのドキュメントのコメント

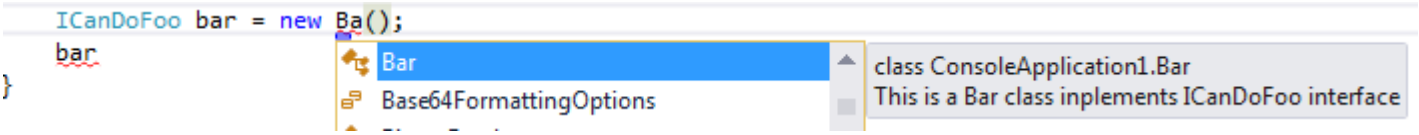
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
```

```
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

インターフェイスの



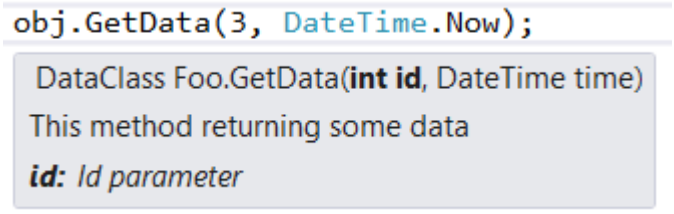
クラス



メソッドのドキュメンテーションはparamでコメントし、をす

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
   // ...
}
```

IntelliSenseは、パラメータのをします。



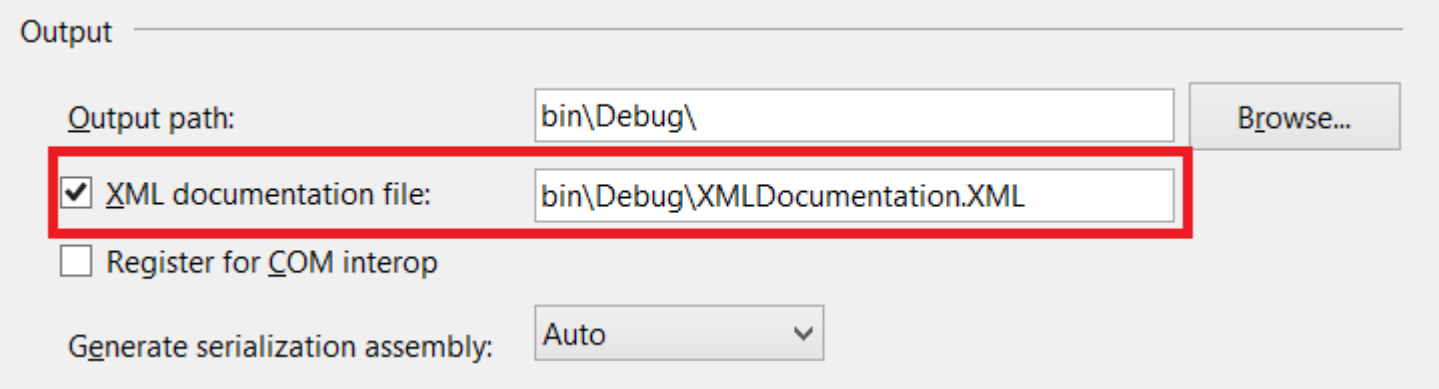
ヒントIntellisenseがVisual Studioにされないは、のかっこまたはカンマをしてからもうしてください。

ドキュメントのコメントからXMLをする

コードのドキュメンテーションコメントからXMLドキュメントファイルをするには、 `csc.exe` Cコンパイラで`/doc`オプションをします。

Visual Studio 2013/2015で、 プロジェクト -> プロパティ -> ビルド -> で、 `XML documentation`

`file`チェックボックスをオンにします。



プロジェクトをビルドすると、プロジェクトにする `XMLDocumentation.dll -> XMLDocumentation.xml` でXMLファイルがコンパイラによってされます。

のプロジェクトでアセンブリをするは、XMLファイルがされているDLLとじディレクトリにあることをしてください。

この

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}


/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

ビルドにこのXMLをします

```
<?xml version="1.0"?>
<doc>
    <assembly>
```

```
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>
```

ドキュメントののクラスの

`<see>`タグは、のクラスにリンクするためにできます。それはされるクラスのをむべき`cref`メンバーをんでいます。 Visual Studioはこのタグをするにintellisenseをし、されるクラスのをするときにそのようながされます。

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

Visual Studio Intellisenseのポップアップでは、そのようなもテキストにきでされます。

ジェネリッククラスをするには、のようなものをします。

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
/// </summary>
public class SomeGenericClass<T>
{
}
```

オンラインでXMLドキュメントのコメントをむ https://riptutorial.com/ja/csharp/topic/740/xmlドキュメントのコメント

56: アクションフィルター

Examples

カスタムアクションフィルタ

さまざまなでカスタムアクションフィルタをします。ロギング、またはアクションにデータをデータベースにするためのカスタムアクションフィルタがあります。また、データベースからデータをし、それをアプリケーションのグローバルとしてすることもできます。

カスタムアクションフィルタをするには、のタスクをするがあります。

1. クラスをする
2. ActionFilterAttributeクラスからする

ののなくとも1つをオーバーライドします。

OnActionExecuting - このメソッドは、コントローラアクションがされるにびされます。

OnActionExecuted - このメソッドは、コントローラアクションがされたにびされます。

OnResultExecuting - このメソッドは、コントローラのアクションがされるにびされます。

OnResultExecuted - このメソッドは、コントローラのアクションがされたにびされます。

のリストにすように、フィルタをできます。

```
using System;

using System.Diagnostics;

using System.Web.Mvc;



namespace WebApplication1
{

    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }

        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            var controllerName = filterContext.RouteData.Values["controller"];
            var actionName = filterContext.RouteData.Values["action"];
```

```
                var message = String.Format("{0} controller:{1} action:{2}",
"onactionexecuting", controllerName, actionName);
                Debug.WriteLine(message, "Action Filter Log");
                base.OnActionExecuting(filterContext);
            }
        }
    }
```

オンラインでアクションフィルターをむ https://riptutorial.com/ja/csharp/topic/1505/アクションフィルター

57: アクセス

アクセスがされている、

- クラスはデフォルトで`internal`
- メソッドはdeault `private`
- gettersとsetterはプロパティのをしますが、デフォルトでは`private`

setterまたはプロパティのgetterのアクセスは、アクセスをするだけで、それをげてはならない
`public string someProperty {get; private set;}`

Examples

パブリック

`public`キーワードは、すべてのコンシューマーができるクラスネストされたクラスをむ、プロパティ、メソッド、またはフィールドをします。

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

プライベート

`private`キーワードは、クラスでのみするプロパティ、メソッド、フィールド、ネストされたクラスをマークします。

```
public class Foo()
{
    private string someProperty { get; set; }

    private class Baz
    {
```

```
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}
```

internalキーワードは、じアセンブリのすべてのコンシューマができるクラスネストされたクラスをむ、プロパティ、メソッド、またはフィールドをします。

```
internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}
```

これは、AssemblyInfo.csファイルにコードをすることで、テストアセンブリがコードにアクセスできるようにすることができます。

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

された

protectedキーワードは、field、methodsプロパティ、およびネストされたクラスを、じクラスおよびクラスでのみするようにマークします。

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

された

`protected internal`キーワードは、じアセンブリまたはのアセンブリのクラスでするフィールド、メソッド、プロパティ、およびネストされたクラスをマークします。

アセンブリ1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty  { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
```

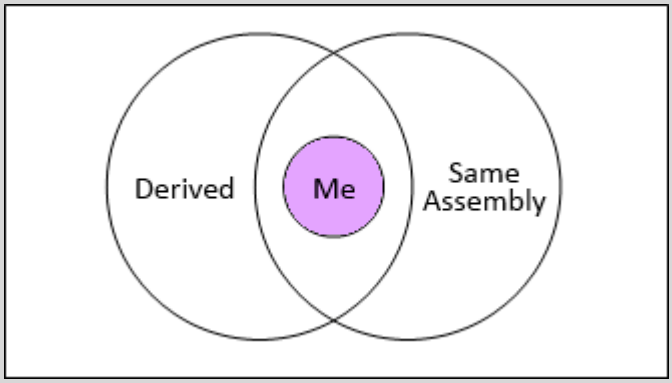
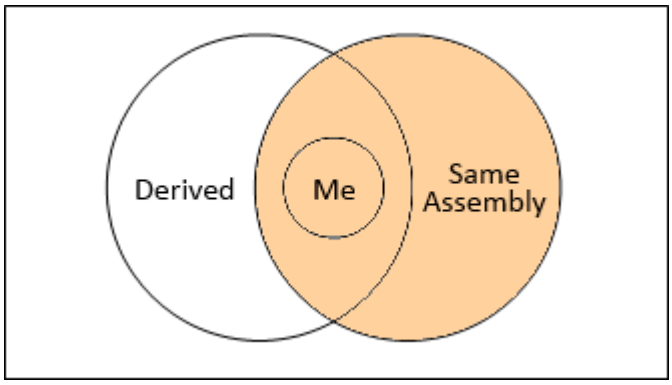
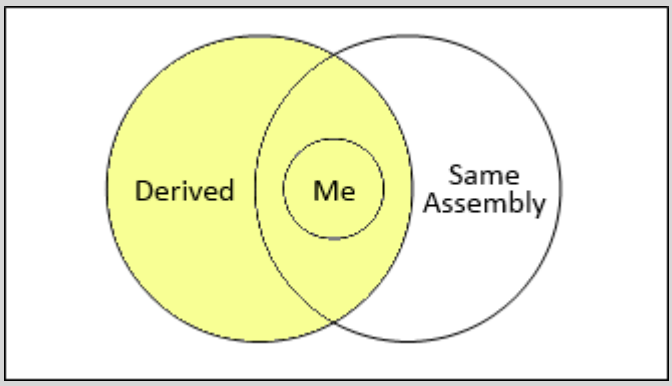
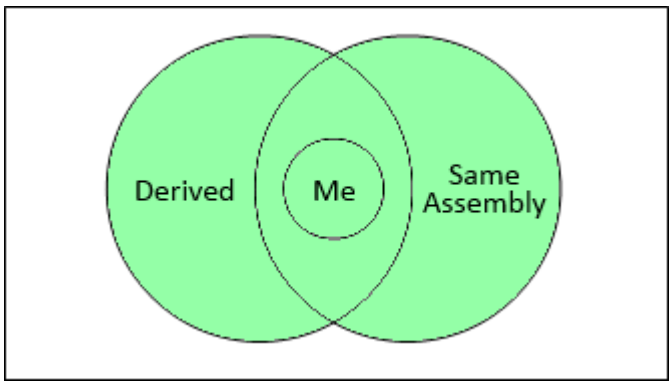
```
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

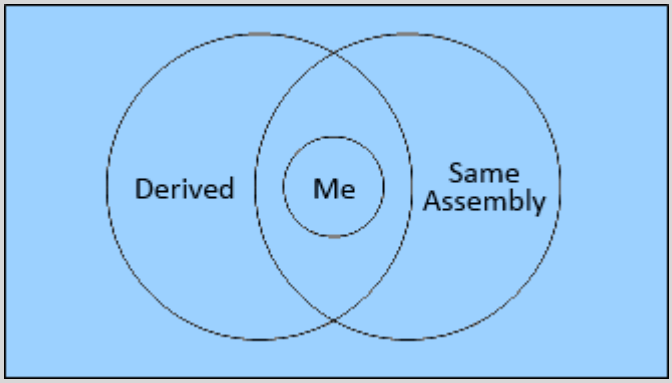
アセンブリ2

```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }

}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

アクセスの

よりなものからアクセスなものまで、すべてのアクセスがあります。

アクセス	
プライベート	
	
された	
された	

アクセス	
パブリック	

あなたのに、よりくのがあります。

オンラインでアクセスをむ https://riptutorial.com/ja/csharp/topic/960/アクセス

58: イテレータ

イテレータは、yieldキーワードをしてまたはコレクションクラスにしてカスタムをするメソッド、getアクセサまたはです

Examples

なイテレータの

イテレータのなは、のにしてらかのをすることです。のは、ののをにコンソールにするをしています。

これは、が`IEnumerable`インターフェイスをし、クライアントが`GetEnumerator()`メソッドをしてのイテレータをできるため`GetEnumerator()`。このメソッドはをします。このは、のにってみりのカーソルです。

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

```
1
2
3
4
5
```

`foreach`をしてじをることもです

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

をしたイテレータの

イテレータはをします。 Cでは、は、 `yield`ステートメントをむメソッド、プロパティ、またはインデクサをすることによってされます。

ほとんどのメソッドは、の`return`をってびしにを`return`ます。これとはに、 `yield`ステートメントをするメソッドでは、があったときにのをびしにすで、それらのをすにローカルをすることができます。これらのりはシーケンスをします。イテレータでされる`yield`には2あります。

- `yield return`はびしにをしますが、はします。コントロールがそれにされるとき、びしはこのからをけます。

- `yield break`は、の`return`とにします。これは、シーケンスのをします。の`return`ステートメントはイテレーター･ブロックではです。

のは、 フィボナッチシーケンス をするためにできるイテレータメソッドをしています。

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

に、このイテレータをして、びしのメソッドによってされるフィボナッチシーケンスのをすることができます。のコードは、フィボナッチシーケンスのの10のをどのようにできるかをしています。

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

```
1
1
2
3
5
8
13
21
34
55
```

オンラインでイテレータをむ https://riptutorial.com/ja/csharp/topic/4243/イテレータ

59: イベント

き

イベントとは、かがしたことマウスクリックなどや、によってはなどこりそうなことのです。

クラスはイベントをすることができ、そのインスタンスオブジェクトはこれらのイベントをさせることがあります。たとえば、Buttonには、ユーザーがクリックしたときにするClickイベントがまれているがあります。

イベントハンドラは、するイベントがしたときにびされるメソッドです。フォームには、ボタンにまれるすべてのボタンのClickedイベントハンドラがまれています。

パラメーター

パラメータ	
EventArgsT	EventArgsからし、イベントパラメータをむ。
イベント	イベントの。
ハンドラー	イベントハンドラの。
SenderObject	イベントをびすオブジェクト。
EventArguments	イベントパラメータをむEventArgsTのインスタンス。

イベントをこすとき

- デリゲートが`null`かどうかをにチェックし`null`。 nullデリゲートは、イベントにサブスクライバがないことをします。サブスクライバのないイベントをさせると、`NullReferenceException`がします。

6.0

- nullをするに、デリゲートeg `EventName` をローカル `eventName` にコピーして、イベントをさせます。これにより、マルチスレッドでのがされます。

っている

```
    if(Changed != null)      // Changed has 1 subscriber at this point
                             // In another thread, that one subscriber decided to unsubscribe
        Changed(this, args); // `Changed` is now null, `NullReferenceException` is thrown.
```

```
    // Cache the "Changed" event as a local. If it is not null, then use
    // the LOCAL variable (handler) to raise the event, NOT the event itself.
    var handler = Changed;
    if(handler != null)
        handler(this, args);
```

6.0

- `if`ステートメントのサブスクライバのヌルチェックではなく、ヌルきをしてメソッドを `EventName?.Invoke(SenderObject, new EventArgsT()); EventName?.Invoke(SenderObject, new EventArgsT());`
- Action <>をしてデリゲートをする、メソッド/イベントハンドラのシグネチャは、イベントのされたデリゲートとじであるがあります。

Examples

イベントのと

イベントの

のをして、の`class`または`struct`イベントをできます。

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

イベントののためのがあり、イベントのプライベートインスタンスをし、 `add`および`set`アクセサをしてパブリックインスタンスをします。はCのプロパティとによくています。コンパイラは、のスレッドがクラスのイベントにイベントハンドラをにおよびできるようにするコードをするため、のをするがあります。

イベントの

6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

6.0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;

    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);

}
```

イベントは、によってのみびすことができます。クライアントは/しかできません。

`EventName?.Invoke`がサポートされていない6.0のCバージョンでは、このにすように、ににイベントをりてて、のスレッドがじものをするにスレッドのをすることをお`EventName?.Invoke`コード。そうしないと、のスレッドがじオブジェクトインスタンスをしているので、`NullReferenceException`がスローされることがあります。 C6.0では、コンパイラはC6のコードにすようなコードをします。

イベント

ィベント

```
public event EventHandler<EventArgsT> EventName;
```

イベントハンドラの

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

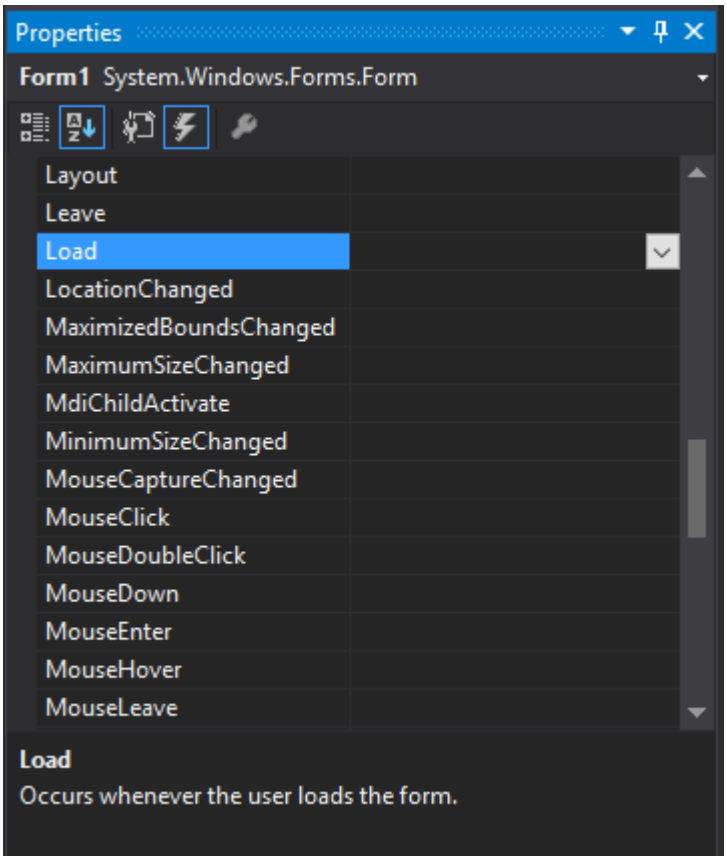
ィベントをする

に

```
EventName += HandlerName;
```

デザィナーをして

1. コントロールのプロパティウィンドウライトニングボルトの[イベント]ボタンをクリックします。
2. イベントをダブルクリックします



3. Visual Studioはイベントコードをします

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

メソッドをびす

```
EventName(SenderObject, EventArguments);
```

イベントハンドラ

イベント

```
public event EventHandler<EventArgsType> EventName;
```

ラムダ=>をしてイベントにするイベントハンドラの

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

デリゲートメソッドをしたイベントハンドラ

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

イベントのパラメータをしないイベントハンドラのとサブスクリプションです。したがって、パ

ラメータをすることなくのをできます。

```
EventName += delegate { /* Handler Logic */ }
```

イベントをびす

```
EventName?.Invoke(SenderObject, EventArguments);
```

イベント

イベントは、 `EventHandler`および`EventHandler<T>`だけでなく、のデリゲートにすることができます。えば

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

これは、の`EventHandler`イベントとにされます。

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

じタイプののイベントをフィールドやローカルとに1つのステートメントにすることはですただし、これはしばしばいえです。

```
public event EventHandler Event1, Event2, Event3;
```

これは、 `EventHandler`の3つの々のイベント `Event1` 、 `Event2` 、および`Event3` をします。
いくつかのコンパイラはインタフェースとクラスでこのをけれるかもしれませんが、Cの*v5.0§13.2.3*はインタフェースをしないインタフェースのをしているため、インタフェースでのはなるコンパイラではできません。

データをむカスタム**EventArg**をする

カスタムイベントは、イベントにするをむカスタムイベントをとします。たとえば、 `MouseDown`イベントや`MouseUp`イベントなどのマウスイベントでされる`MouseEventArgs`には、イベントのにされた`Location`または`Buttons`にするがまれています。

しいイベントをするときに、カスタムイベントargをするには

- `EventArgs`からしたクラスをし、なデータのプロパティをします。
- として、クラスのは`EventArgs`わるがあります。

のでは、クラスの`Price`プロパティの`PriceChangingEventArgs`イベントをします。イベントデータクラスには、 `CurrentPrice`と`NewPrice`がまれています。このイベントは、しいを`Price`プロパティにりてるとにがしていることをらせ、のとしいについてらせるときにします。

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
}
```

がしいをできるようにして、このをすると、そのがプロパティにされます。これをうには、これらのをクラスにすればです。

`NewPrice`のをにします。

```
public int NewPrice { get; set; }
```

`e.NewPrice`をびした、 `OnPriceChanging`をプロパティのとしてするために`Price`のをします。

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

キャンセルなイベントの

キャンセルなイベントは、クラスの`FormClosing`イベントなど、キャンセルなアクションをしようとしているときにさせることができ`Form`。

このようなイベントをするには

- `CancelEventArgs`からしたしいイベントをし、イベントデータのプロパティをします。
- `EventHandler<T>`をしてイベントをし、したしいキャンセルイベントargクラスをします。

のでは、クラスの`Price`プロパティの`PriceChangingEventArgs`イベントをします。イベントデータクラスには、がしいものについてることができる`Value`がまれています。このイベントは、しいを`Price`プロパティにりてるとにがされていることをし、イベントをキャンセルさせるときにします。がイベントをキャンセルした、の`Price`がされます。

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
```

```
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

イベントのプロパティ

クラスがイベントのをきくした、デリゲートごとに1つのフィールドのストレージコストはけれられないがあります。 .NET Frameworkは、これらのケースのイベントプロパティをします。こので、 `EventHandlerList`などののデータをして、イベントデリゲートをできます。

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

このアプローチは、WinFormsなどのGUIフレームワークでくされています。このフレームワークでは、コントロールにはのイベントがまれます。

`EventHandlerList`はスレッドセーフではないので、クラスをのスレッドからすることをするは、ロックステートメントやそののメカニズムをするがありますまたはスレッドのをするストレージを

するがあります。

オンラインでイベントをむ https://riptutorial.com/ja/csharp/topic/64/イベント

60: インターフェイス

Examples

インタフェースの

インタフェースは、それを「する」どのクラスにもメソッドのをするためにされます。インターフェースはキーワード`interface`され、クラスはクラスのろに：`InterfaceName`をすることでそれを"できます。クラスは、インタフェースをコンマでってのインタフェースをできます。
`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

らは`INoiseMaker`をしている`INoiseMaker`、 `cat`と`dog`に`string MakeNoise()`をめるがあり、それなしではコンパイルにします。

のインタフェースの

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
```

```
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

なインターフェイスの

なインタフェースのは、のメソッドをするのインタフェースをするにですが、メソッドをびすためにされるインタフェースにじてなるがですのインタフェースがじメソッドをするはなはありません。のがである。

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }

    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}


GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

は、インタフェースをするはどこからでもびすことはできません。

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
    {
        return "Swinging hard...";
    }
```

```
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

このため、にされたインタフェースのなコードをのプライベートメソッドにれることはです。

もちろん、なインタフェースのは、そのインタフェースににするメソッドにしてのみできます。

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

に、クラスのインタフェースをせずになインタフェースをすると、エラーもします。

ヒント

にインタフェースをすることで、デッドコードをすることもできます。メソッドがもはやなくなり、インタフェースからりかれると、コンパイラはまだするについてをちます。

プログラマーは、のコンテキストにかかわらずがじであることをしています。なでは、びされたときになるをすべきではありません。のとはなり、 `IGolfPlayer.Drive` と `Drive` はなりじことをうがあります。

インターフェイスをする

インターフェースとは、インターフェースのユーザーとそれをするクラスとののののです。インタフェースをえるの1つは、オブジェクトがのをできるというである。

なるタイプのシエイプをするためのインタフェース `IShape` をし、シエイプにがあることをしているので、インタフェースがそのをすようにするメソッドをします。

```
public interface IShape
{
    double ComputeArea();
}
```

の2つのがあります `Rectangle` と `Circle`

```
public class Rectangle : IShape
{
    private double length;
```

```
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

それらのそれぞれは、そののののをっていますが、どちらもです。だからたちのプログラムでは、それらを`IShape`としてることはです。

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}", shape.ComputeArea());
    }
}

// Output:
// Area : 50.00
// Area : 78.54
```

インタフェーースの

の「」としてられるインタフエーースの。つまり、プロパティとメソッドをしますが、されていません。

だからクラスのインターフェイスとはって

- インスタンスできません
- をつことはできません
- メソッドのみをめることができます* プロパティとイベントはにメソッドです
- インタフェースをすることを「する」といいます。
- あなたは1つのクラスからすることができますが、のインターフェイスをすることができます

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

- "I"はインタフェースにされるです。
- はセミコロン ";"にきえられます。
- にはメソッドでもあるため、プロパティもされています

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

。

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

これは、WinFormsやWPFなどのUIフレームワークでするににです。これは、クラスからしてユーザーコントロールをし、なるコントロールタイプにしてをするがわれてしまうためです。る

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}
```

```
public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

されたは、どちらも "テキスト"というをんでいますが、プロパティがなります。また、2つのなるクラスにのがあるため、クラスをすることはできません。インター―フェイスをすると、

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

すぐMyButtonとMyTextBlockはがあります。

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");


    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;


    /* the solution is to check the type first.
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors


}
```

による「」メンバー

あなたがにしないメンバーがすぎると、インターフェイスがあなたのクラスをするとき、あなたはそれをいませんかさて、はをましたな

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

、このようなクラスをします。

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

すべてのメンバーはされています。

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

えはしません。したがって、どちらもされるべきではありませんが、にメンバーをとしてすれば、コンパイラはエラーをスローします

ソリューションはなをすることです

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

だから、あなたはにじてメンバーをしました。メンバーをすることはありません。

```
var obj = new MyObjectWithMessages();
```

```
/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

にしていても、メンバーにアクセスしたいは、オブジェクトをインターフェースにキャストするだけでみます。

```
((IMessageService)obj).OnMessageRecieve();
```

IComparable インタフェースのとして

インターフェイスは、あなたがにそれらのようにえるまでにえることがあります。 IComparableとIComparable<T>は、インターフェイスがなぜたちにつのかのらしいです。

オンラインストアのプログラムでは、できるさまざまなアイテムがあるとしましょう。には、ID、があります。

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity

}
```

たちはItem List<Item>にしています。たちのプログラムでは、リストをIDでソートしたいのです。のソートアルゴリズムをSort()わりに、 List<T>にあるSort()メソッドをわりにできます。しかし、たちのItemクラスはのところ、リストをソートするをList<T>がするはありません。ここにIComparableインターフェイスがってきます。

しくするためCompareTo、 CompareToパラメータが「よりきい」である、パラメータはのゼロ「」である、それらがしいをし、のべきです。

```
Item apple = new Item();
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Itemのインターフェイスのをにします。

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity

}
```

サーフェスレベルでは、の`CompareTo`メソッドはにIDのをしますが、にはのことはですか

`List<Item>`オブジェクトにして`Sort()`をびすと、オブジェクトをするをするがあるときに`List`はに`Item`の`CompareTo`メソッドをびします。さらに、 `List<T>`ほかに、 2つのなる`Item`いにするをしているので、2つのオブジェクトをするをとするものは、 `Item`とします。

オンラインでインターフェイスをむ https://riptutorial.com/ja/csharp/topic/2208/インターフェイス

61: インデクサー

- publicりthis [IndexType index] {get {...} set {...}}

インデクサーは、のようながインデックスをつオブジェクトのプロパティにアクセスすることをにします。

- クラス、、またはインタフェースでできます。
- オーバーロードすることができます。
- のパラメータをできます。
- にアクセスしてをするためにできます。
- ののインデックスをできます。

Examples

シンプルなインデクサ

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

デモをる

2つのとインタフェースをつIndexer

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}
```

```
class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

インデクサーをオーバーロードしてSparseArrayをする

インデクサーをオーバーロードすることで、のようにえ、じるクラスをできますが、そうではありません。 O1メソッドをしてし、インデックス100のにアクセスできますが、そのにのサイズをします。 SparseArrayクラス

```
class SparseArray
    {
        Dictionary<int, string> array = new Dictionary<int, string>();

        public string this[int i]
        {
            get
            {
                if(!array.ContainsKey(i))
                {
                    return null;
                }
                return array[i];
            }
            set
            {
                if(!array.ContainsKey(i))
                    array.Add(i, value);
            }
        }
    }
```

オンラインでインデクサーをむ https://riptutorial.com/ja/csharp/topic/1660/インデクサー

62: オーバーフロー

Examples

オーバーフロー

にできるがあります。そしてそのをえると、それはのにループバックします。 `int`は`2147483647`

```
int x = int.MaxValue;                //MaxValue is 2147483647
x = unchecked(x + 1);                //make operation explicitly unchecked so that the example
also works when the check for arithmetic overflow/underflow is enabled in the project settings

Console.WriteLine(x);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

こののの、データBigIntegerをつSystem.Numericsをします。については、のリンクをしてください。https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx

のオーバーフロー

にオーバーフローもします。のでは、xは`int` 、1はデフォルトで`int`です。したがって、は`int`です。は`int`ます。それはオーバーフローします。

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1;                     //It will be overflown
Console.WriteLine(y);               //Will print -2147483648
Console.WriteLine(int.MinValue);    //Same as Min value
```

1Lをすることでそれをぐことができます。は1が`long`なり、 `long`されます

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1L;                    //It will be OK
Console.WriteLine(y);               //Will print 2147483648
```

のコードにオーバーフローがあります

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

のコードではオーバーフローはありません

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

これは、のからへのけによるものです。のコードでは、 `x + x`オーバーフローがし、そのは`long`ます。、 `x + 1L`は`long`なり、その`x`がこのにされます。

オンラインでオーバーフローをむ https://riptutorial.com/ja/csharp/topic/3303/オーバーフロー

63: オブジェクト

- SomeClass sc =しいSomeClass {Property1 =1、Property2 =2、...};
- SomeClass sc = new SomeClassparam1、param2、...{Property1 = value1、Property2 = value2、...}

コンストラクタのカッコは、インスタンスされるがデフォルトのパラメータなしのコンストラクタをできるにのみできます。

Examples

ない

オブジェクトは、オブジェクトをしてすぐにいくつかのプロパティをするがあるにですが、なコンストラクタではです。クラスがあるとします

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

をしてクラスのしいインスタンスをするには

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

これは

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

の

オブジェクトは、コンパイラによってされるであるをするのです。

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

そのため、オブジェクトは、のオブジェクトのどのにがあるかをするなをするため、LINQクエリでくされています。

```
var albumTitles = from a in albums
                  select new
                  {
```

```
                    Title = a.Title,
                    Artist = a.Band
                };
```

デフォルトのコンストラクタでの

にじて、オブジェクトをコンストラクタとみわせてをすることができます。たとえば、のようにされたクラスをる

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

これは、まず`Book(int)`コンストラクタで`Book`をインスタンスし、にイニシャライザのプロパティをします。これはのようになります。

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

オンラインでオブジェクトをむ https://riptutorial.com/ja/csharp/topic/738/オブジェクト

64: オブジェクトプログラミングC

き

このトピックでは、OOPアプローチについてプログラムをくをえてください。しかし、オブジェクトプログラミングのパラダイムをえようとはしません。クラス、プロパティ、、、インタフェースなどのトピックについてします。

Examples

クラス

クラスのスケルトンは

<>

[]オプション

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited
class][,][[Interface Name 1],[Interface Name 2],...]
{
    //Your code
}
```

をできないはしないでください。のとして、のようなクラスをえてみましょう。

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

このクラスでは`int`の`i`をし、デフォルトのprivate アクセスとpublicアクセスをつ`getMyValue()`メソッドをします。

オンラインでオブジェクトプログラミングCをむ https://riptutorial.com/ja/csharp/topic/9856/オブジェクトプログラミングc-

65: ガイド

き

GUIDまたはUUIDは、「グローバル」または「ユニバーサル」のです。これは、リソースをするためにされる128ビットのです。

`Guid`は、 *UUIDのUniversally Unique Identifiers*ともばれるグローバル です。

それらは128ビットのです。いランダムアルゴリズムによってされた、それらはすべてのなによってでであるとえることができるので、にくのな`Guid` のすべてののセルについて10＾18の`Guid` があります。

`Guid`は、データベースのキーとしてもにされます。らのは、ほとんどであることがされているしいIDをするためにデータベースをびすがないことです。

Examples

Guidのの

Guidのは、みみの`ToString`メソッドをしてできます

```
string myGuidString = myGuid.ToString();
```

にじて、 `ToString`びしにタイプをして、Guidをすることもできます。

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None           "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens        "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces         "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses    "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex            "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

ガイドをする

これらは、Guidのインスタンスをするもなです。

-

のGUID `00000000-0000-0000-0000-000000000000` の

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- しいGuidをする

```
Guid g = Guid.NewGuid();
```

- のをつガイドをする

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

nullなGUIDをする

のとに、GUIDにはNULLをとることができるnullなもあります。

```
Guid? myGuidVar = null;
```

これは、テーブルのがNULLであるがあるに、データベースからデータをするににです。

オンラインでガイドをむ https://riptutorial.com/ja/csharp/topic/1153/ガイド

66: キーワード

き

キーワードは、あらかじめされたみで、コンパイラにとってなをちます。 `@`をけずにプログラムのとしてうことはできません。えば`@if`キーワードなではありません`if`。

Cには、それぞれがなをつみの "キーワード"またはのコレクションがあります。これらのは、 `@`にけないり、、メソッド、クラスなどのとしてすることはできません。

- abstract
- as
- base
- bool
- break
- byte
- case
- catch
- char
- checked
- class
- const
- continue
- decimal
- default
- delegate
- do
- double
- else
- enum
- event
- explicit
- extern
- false
- finally
- fixed
- float
- for
- foreach
- goto
- if
- implicit
- in
- int
- interface
- internal
- is
- lock
- long
- namespace
- new

- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using
- using ステートメント
- virtual
- void
- volatile
- when
- while

これらとはに、Cはコードでのをするためにいくつかのキーワードもします。それらをコンテキストキーワードといいます。コンテキストキーワードはとしてすることができ、のに@けるはありません。

- add
- alias
- ascending
- async
- await
- descending
- dynamic
- from
- get
- global
- group
- into

- join
- let
- nameof
- orderby
- partial
- remove
- select
- set
- value
- var
- where
- yield

Examples

stackalloc

`stackalloc`キーワードは、スタックにメモリのをし、そのメモリのへのポインタをします。スタックにりてられたメモリは、されたスコープがするとにされます。

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
ptr[2] = 232;
...
```

でないでされます。

Cのすべてのポインタとに、みりとりてのをするはありません。りてられたメモリのをえてみると、メモリののにアクセスするがあり、アクセスのがするがあります。

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

スタックにりてられたメモリは、されたスコープがするとにされます。これは、stackallocでされたメモリをすことはにけ、スコープのをえてすることはできません。

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
```

```
     //ptr is now an invalid pointer, using it in any way will have
     //unpredictable results. This is exactly the same as accessing beyond
     //the bounds of the pointer.
     var ptr = Leak();
}
```

stackallocは、のとにのみできます。はではありません 。

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

---

stackallocは、パフォ―マンスのまたはinteropのいずれかにのみするがあります。これは、のによるものです。

- メモリがヒ―プではなくスタックにりてられているためガベ―ジコレクタはです。がになるとすぐにメモリがされます
- ヒ―プではなくスタックにメモリをりてるがい
- デ―タのによるCPUのキャッシュヒットのをめる

の

volatileキ―ワ―ドをフィ―ルドにすると、フィ―ルドのがの々のスレッドによってされるがコンパイラにされます。 volatileキ―ワ―ドのなは、シングルスレッドアクセスのみをしたコンパイラのをすることです。 volatileをすると、フィ―ルドのがなのになり、そのはのであるキャッシングのにはなりません。

でのによるしないをぐために、のスレッドによってされるがあるすべてのをvolatileとしてマ―クすることをおめします。のコ―ドブロックをえてみましょう。

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

のコ―ドブロックでは、コンパイラは、みしx = 5とy = x + 10のとしyいつものようにこのように、それがのステ―トメントをする15ようにしてしまうy = 15 。 しかし、xはにはpublicフィ―ルドであり、 xのはこのフィ―ルドで々にするのスレッドをじてにされるがあります。このされたコ―ドブロックをえてみましょう。フィ―ルドxがvolatileとしてされていることにしてください。

```
public class Example
{
    public volatile int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

、コンパイラはフィールド$_x$みみをし、フィールドののがにされるようにします。これにより、のスレッドがこのフィールドをみきしていても、に$_x$ののがされます。

`volatile`は、 `class`または`struct`のフィールドでのみできます。はではありません 。

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile`はののフィールドにしかできません

- またはのとしてられているジェネリックパラメータ
- `sbyte` 、 `byte` 、 `short` 、 `ushort` 、 `int` 、 `uint` 、 `char` 、 `float` 、 `bool`などのプリミティブ
- `byte` 、 `sbyte` 、 `short` 、 `ushort` 、 `int`または`uint`づいてを`sbyte`
- `IntPtr`と`UIntPtr`

- `volatile`は、lockをしてアクセスをシリアライズせずにのスレッドがアクセスするフィールドにされます。
- `volatile`キーワードは、のフィールドにできます。
- `volatile`キーワードは、32ビットプラットフォームの64ビットプリミティブではしません。のような`Interlocked.Read`と`Interlocked.Exchange`まだこれらのプラットフォームでなマルチスレッドアクセスにするがあります。

fixedステートメントは、メモリを1つのにします。メモリのオブジェクトはですので、ガベージコレクションがです。しかし、でないポインタをメモリアドレスにすると、そのメモリをしてはいけません。

- ガベージコレクタがデータをしないようにfixedをします。

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
```

```
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

でないでされます。

サイズ

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed`は`struct`フィールドでのみできますでないコンテキストでもするがあります。

デフォルト

クラス、インタフェース、デリゲート、、nullableintなどおよびポインタの、 `default(TheType)`は`null`し`null`。

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

との、 `default(TheType)`は`new TheType()`とじをします。

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)`ににでありる`T`くがかどうかをするためにしていないいるため、なパラメータであり`T`えば、またはです。

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

みり

readonlyキーワードはフィールドです。フィールドにreadonlyがまれている、そのフィールドへのはのとして、またはじクラスのコンストラクターでのみできます。

readonlyキーワードはconstキーワードとはなりreadonly。 constフィールドは、フィールドのでのみできます。 readonlyフィールドは、またはコンストラクタでできます。したがって、 readonlyフィールドは、されるコンストラクタにじてなるをつことができreadonly。

readonlyキーワードは、をreadonlyするときによくわれreadonly。

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";

    Person(string name)
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

フィールドをみみにしても、はしません。フィールドがの、オブジェクトのをすることができます。、 *Readonly*はオブジェクトをきしてそのオブジェクトのインスタンスにのみりてるのをぐためにされます。

コンストラクタのでは、readonlyフィールドをりてすることができます

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();
```

```
private void SomeMethod()
{
    car.Speed = 100;
}
```

として

asキーワードは、 キャストにたです。キャストできないは、しasnullではなく、そのより InvalidCastException。

expression as typeはexpression as typeとじexpression is type ? (type)expression : (type)nullでそのas、NULLコンバージョン、ボクシングでのみです。ユーザーのはサポートされていません。わりにのキャストをするがあります。

ののために、コンパイラは、 expressionがしかされず、のチェックをするようなコードをしますのサンプルの2つとなります。

いくつかのをにするためのをするときにちます。にis 、キャストするにすべてのをチェックするのではなく、をキャストしてキャッチするのでisなく、のオプションをユーザーにします。オブジェクトをキャスト/チェックするときには 'as'をすることがベストプラクティスであり、アンボックスペナルティが1つしかしません。 isすること、そしてキャストは2つのアンボクシングのがします。

がののインスタンスであるとされるは、そのがにとってよりであるため、のキャストがされます。

asびすとnullがさasがあるため、にをチェックしてNullReferenceExceptionをします。

```
object something = "Hello";
Console.WriteLine(something as string);        //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);          //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

.NET Fiddleのライブデモ

のようにasないの

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

これは、カスタムクラスでEqualsをオーバーライドするにです。

```
class MyCustomClass
{

    public override bool Equals(object obj)
```

```
    {
        MyCustomClass customObject = obj as MyCustomClass;

        // if it is null it may be really null
        // or it may be of a different type
        if (Object.ReferenceEquals(null, customObject))
        {
            // If it is null then it is not equal to this instance.
            return false;
        }

        // Other equality controls specific to class
    }

}
```

は

オブジェクトがのとがあるかどうか、つまりオブジェクトが`BaseInterface`のインスタンスか、`BaseInterface`からしたかどうかを`BaseInterface`ます。

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

キャストのがオブジェクトをすることであるは、 `as`キーワード'

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True - valid use of 'is'
Console.WriteLine(d is BaseClass);     // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
```

```
    asD.Method(); //prefered method since you incur only one unboxing penalty
}
```

しかし、C7の`pattern matching`では、isをしてをチェックし、にしいをします。 C7のじコード

7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

タイプ

オブジェクトをインスタンスするなく、オブジェクトの`Type`をします。

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

const

`const`は、プログラムのをしてしてしないをすためにされます。そのは、からのをつ`readonly`キーワードとはに、 コンパイルからです。

たとえば、のはしてしないので、にすることができます。

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

これは、コンパイラが`c`をそのでするので、には`return mass * 299792458 * 299792458`をつ`return mass * 299792458 * 299792458`とじです。

その、された`c`はできません。は、コンパイルエラーをします

```
const double c = 299792458;  // Speed of light

c = 500;  //compile-time error
```

には、メソッドとじアクセスのをけることができます。

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const`メンバーは`static`です。ただし、 `static`にすることはできません。

メソッドロ―カルをすることもできます。

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

これらは、されたメソッドにしてにロ―カルであるため、 `private`または`public`キ―ワ―ドのにけることはできません。

`const`ではすべてのをできるわけではありません。されるのは、 `sbyte` 、 `byte` 、 `short` 、 `ushort` 、 `int` 、 `uint` 、 `long` 、 `ulong` 、 `char` 、 `float` 、 `double` 、 `decimal` 、 `bool` 、およびすべての`enum`です。の `TimeSpan`や`Guid` で`const`メンバ―をしようとすると、コンパイルにします。

なされた`string`、はのでできます。のすべてのでは、をできますが、には`null`なければなりません。

`const`はコンパイルにされるため、 `switch`の`case`ラベル、オプションのパラメ―タの、のなどとしてできます。

`const`がなるアセンブリでされるは、バ―ジョンにするがあります。たとえば、アセンブリAが`public const int MaxRetries = 3;`する`public const int MaxRetries = 3;`アセンブリBがそのをする、アセンブリAでで`MaxRetries`のを`5`にしたコンパイルした、アセンブリBがコンパイルされないり、そのはアセンブリBではになりませんAのしいバ―ジョンへの。

そのため、は、プログラムののでされるがある、そのはにするがあるは、そのをしない`const`あなたはかがされるたびに、すべてのアセンブリがコンパイルされることをっているをきます。わりに、にされる`const`わりに`static readonly`をしています。

`namespace`キ―ワ―ドは、コ―ドベ―スのをするのにつです。 Cのは、なフォルダではなく、です。

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
```

```
    }
}
```

Cのは、でることもできます。はとじです

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

してみて

`try` 、 `catch` 、 `finally` 、 `throw`すると、コードのをできます。

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block
that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try
has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}
```

`return`キーワードは`try`ブロックですることができ、 `finally`ブロックはすにされます。えば

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

`connection.Close()`のは、 `connection.Get(query)`がされるにされ`connection.Get(query)`。

する

すぐに、みループののりしにをすfor、foreach、do、while

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

5
6
7
8
9

.NET Fiddleのライブデモ

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

a
b
c
d

.NET Fiddleのライブデモ

ref、out

refキーワードとoutキーワードは、ではなくしのになります。の、これは、のがびしによってできることをします。

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

の、のインスタンスは refなしののようにすることはできませんが、きえることもできます

```
Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now
```

outキーワードとrefキーワードのないは、 refではびしによってがされるがあり、 outはそのびしをびしにすことです。

outパラメータをするには、メソッドとびしメソッドのでにoutキーワードをするがあります。

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number);  // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number);  // number = 34

void AddOneByRef(ref int value)
{
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

.NET Fiddleのライブデモ

outパラメータは、メソッドがるにりてられたをたなければならないので、コンパイルされませんわりにrefをしてコンパイルされます。

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

Generic Modifierとしてキーワードをする

outキーワードは、ジェネリックインターフェイスとデリゲートをするときに、ジェネリックパラメータでもできます。この、 outキーワードは、パラメータがであることをします。

をすると、パラメーター—でされたよりもをすることができます。これにより、バリアントインター—フェイスをするクラスのなと、デリゲート—のながになります。とはでサポート—されていますが、ではサポート—されていません。 - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

チェックされている、チェックされていない

checked uncheckedキーワードとuncheckedキーワードは、がなオーバーフロー—をどのようにするかをします。における「オーバーフロー」checkedし、 uncheckedキーワードがあるに、ターゲット·データ·タイプをすことができるよりもきさがきいでの。

checkedブロックでオーバーフロー—がしたまたはコンパイラがチェックされたをグローバルにするようにされている、ましくないをするがスロー—されます。、 uncheckedれていuncheckedブロックでは、オーバーフロー—はしません。はスロー—されず、はにのにりされます。これはで、つけにくいバグにつながるがあります。

ほとんどのはオーバーフロー—するほどきくないかさいでわれるため、ほとんどの、ブロックをcheckedれたものとしてにするはありません。えば、でをうときやユーザー—をけているときなど、オーバーフロー—をきこすのあるのないにしてをうときはがです。

*checked*も*unchecked checked*もにしません。

ブロックまたはがuncheckedされていuncheckedとされた、そののはエラー—をさせることなくオーバーフロー—することができます。このがましいは、チェックサムのであり、にそのが「ラップアラウンド」できるようになります。

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

uncheckedされていuncheckedもなの1つは、チェックサムのであるobject.GetHashCode()カスタムオーバーライドをすることです。このにするえにキーワードのがされます。オーバーライドされたSystem.Object.GetHashCodeのなアルゴリズムはですか 。

ブロックまたはがcheckedされているとされた、オーバーフロー—をきこすの、 OverflowExceptionが

スローされます。

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

checkedとuncheckedのがブロックとのであるがあります。

チェックされたブロックとチェックされていないブロックは、びされたメソッドにはしません。たとえば、 `Enum.ToObject()` 、 `Convert.ToInt32()` 、およびユーザーのは、カスタムのチェックされた/チェックされていないコンテキストのをけません。

デフォルトのオーバーフローデフォルトチェックとチェックは、 プロジェクトのプロパティまたは***/checked [+ | -]***コマンドラインスイッチでできます。デバッグビルドではチェックされたにデフォルトし、リリースビルドではチェックをすことがです。 *checked unchecked*キーワードと *unchecked*キーワードは、 デフォルトのアプローチがされないにのみされ、をするためにはながです。

`goto`は、ラベルでされたコードののにジャンプするためにできます。

aとしての goto

ラベル

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

.NET Fiddleのライブデモ

ケーステートメント

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
```

```
}
```

.NET Fiddleのライブデモ

これは、Cがフォールスルーケースブロックをサポートしていないため、switchでのをするににです。

の

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

.NET Fiddleのライブデモ

くのとに、のをいて、gotoキーワードのはおめしません。

Cにされるgotoな

- switchのfall-throughの
- マルチレベルブレーク。わりにLINQをすることもできますが、はパフォーマンスがします。
- アンラップされたレベルのオブジェクトをうときのリソースのりて。 Cでは、レベルのオブジェクトはのクラスにラップするがあります。
- 、えばパーサー;コンパイラによってされた/マシンをにします。

enumキーワードは、にすることなく、このクラスがクラスEnumからすることをコンパイラにえます。 Enumは ValueTypeであり、のきのセットですることをしています。

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

オプションで、それぞれのまたはそのをすることもできます。

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

このでは0のをしましたが、これはいです。 enumはな(YourEnumType) 0によってされるデフォルトをにちます。ここでYourEnumTypeはされたenumeです。 0のがされていなければ、 enumはにされたをちません。

enumのデフォルトのになるはintです。となるをbyte、 sbyte、 short、 ushort、 int、 uint、 long、 ulongなどのにできます。は、となるbyteつです

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

また、キャストをしてタイプとのですることもできます。

```
int value = (int)NotableYear.EndOfWwI;
```

これらのから、ライブラリをしているときにenumがかどうかをにするほうがよいでしょう。

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

ベース

baseキーワードは、クラスのメンバーにアクセスするためにされます。これは、メソッドのをびすか、びすべきコンストラクタをするためによくされます。

コンストラクタの

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
```

```
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in
Child's constructor
        Console.WriteLine(message);
    }
}
```

メソッドのをびす

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

のメソッドからをびすために、baseキーワードをすることはです。これは、メソッドびしをベースにびつけます。つまり、しいクラスがメソッドをオーバーライドしても、がびされるため、がです。

```
public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());

        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
```

```
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());

        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}
```

foreach

`foreach`は、のまたは`IEnumerable` imをするコレクションのをするためにされます。

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

これはされます

    "こんにちは"
    "ごはいかがですか"
    "さようなら"

.NET Fiddleのライブデモ

breakキーワードをして`foreach`ループをするか、 continueキーワードをしてのりしにむことができます。

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}
```

```
// Prints: 1, 3, 4,
```

.NET Fiddleのライブデモ

のは、やListなどののコレクションにしてのみされますが、のくのコレクションではされません。

IEnumerableはになコレクションをすためにされますが、 foreachは、コレクションがobject GetEnumerator()メソッドをしていることをし、 bool MoveNext()メソッドとobject Current { get; }プロパティ。

パラメータ

paramsは、メソッドパラメータがのをけることをにします。つまり、0、1つまたはのがそのパラメータにしてされます。

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

このメソッドはintのなリスト、またはintのでびすことができるようになりました。

```
AddAll(5, 10, 15, 20);               // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

paramsはくてもれなければならず、もしされるならば、のがのとなっていても、リストのになければなりません。

---

paramsキーワードをするときにのオーバーロードにしてください。 Cは、よりのオーバーロードとのマッチングをしてから、 paramsオーバーロードをしようとしています。たとえば、の2つのがあるとします。

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }
```

```
    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

に、 `params`オーバーロードをすに、の2のオーバーロードがされます。

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

ブレーク

ループでfor、foreach、do、while、 `break`はものループのをし、そのろのコードにります。また、イテレータがしたことをする`yield`でることもできます。

```
for (var i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

.NET Fiddleのライブデモ

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

breakは、caseまたはdefaultセグメントからするためにswitch-caseコンストラクトでもされます。

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

switchでは、caseのに 'break'キーワードがです。これは、シリーズののケース・ステートメントに「」することをにするいくつかのにする。このをするには、 'goto'をめるか、 'case'をにみねます。

のコードは`0, 1, 2, ..., 9`をえ、のはされません。 `yield break`は、のをしますなるループではありません。

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

.NET Fiddleのライブデモ

のとはなり、Cでのりにラベルをけるはありません。つまり、ネストされたループの、ものループだけがされます。

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
    {
        if (innerItem.ShoudBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
    }
}
```

ここでのループからするは、のようないくつかののうちの1つをできます。

- ループからびすためのgoto。
- ループのりしのにチェックできるのフラグのでは`shouldBreak`。
- コードをリファクタリングして、ものループで`return`をするか、またはネストされたループをにします。

```
bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
```

```
            shouldBreak = true;
        }
    }

    if(shouldBreakNow)
    {
        break; // Break out of outer loop if flag was set to true
    }
}

endAllLooping: // label from where control flow will continue
```

キーワード`abstract`マークされたクラスはインスタンスできません。

メンバをむクラス、またはしていないメンバをするクラスは 、クラスとしてマークするがあります。メンバがまれていなくても、クラスはクラスとしてマークされることがあります。

クラスは、、のをのコンポーネントでするがあるに、クラスとしてされます。

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();       // Allowed due to Cat deriving from Animal
cat.MakeSound();              // will print out "Meov meov"

Animal dog = new Dog();       // Allowed due to Dog deriving from Animal
dog.MakeSound();              // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

キーワード`abstract`マークされたメソッド、プロパティ、またはイベントは、そのメンバのがサブクラスでされることがされることをします。でべたように、メンバはクラスでしかできません。

```
abstract class Animal
{
   public abstract string Name { get; set; }
```

```
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

float、double、decimal

く

`float`は.NETデータ`System.Single`エイリアスです。 IEEE 754をすることができます。このデータは、 `mscorlib.dll`します`mscorlib.dll`は、するときにすべてのCプロジェクトによってにされます。

3.4×10 38 -3.4×10 38

69

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> `float`は、しばしばなめをじることにすべきである。がなアプリケーションでは、のデータをするがあります。

ダブル

`double`は.NETデータ`System.Double`エイリアスです。の64ビットをします。このデータは、Cプロジェクトでにされる`mscorlib.dll`します。

±5.0×10 -324 ±1.7×10 308

1516

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

の

decimalは、.NETデータSystem.Decimalエイリアスです。キーワードは128ビットのデータをします。として、はとがさく、およびにしています。このデータは、Cプロジェクトでにされるmscorlib.dllします。

-7.9×10 28 7.9×10 28

2829

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

uint

なし、またはuintは、ののみをできるデータです。のように、なし32ビットをします。 uintキーワードは、Common Type Systemタイプの`System.UInt32`です。このデータは`mscorlib.dll`にあります`mscorlib.dll`、するときにすべてのCプロジェクトによってにされます。 4バイトのメモリをします。

なしは0から4,294,967,295までののをできます。

どのようにしてなしをしないかの

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Microsoftによると、 uintデータはCLSにしていないため、なりintデータをすることをおめします。

この

`this`キーワードは、クラスオブジェクトののインスタンスをします。そうすることで、クラスフィールドとメソッドのパラメータまたはローカルの2つのをすることができます。

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

キーワードののは、 なコンストラクタのオーバーロードをさせることです。

```
public MyClass(int arg) : this(arg, null)
```

```
{
}
```

インデックス

```
public string this[int idx1, string idx2]
{
    get { /* ... */ }
    set { /* ... */ }
}
```

メソッドをする

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

ロ―カルまたはパラメ―タとのがないは、それをするかどうかのスタイルのであり`this`かどうか、そう`this.MemberOfType`と`MemberOfType`、そのにします。 `base`キ―ワ―ドもてください。

のインスタンスでメソッドをびすは、 `this`ができることにしてください。たとえば、`IEnumerable<>`をしているクラスのでないメソッドのにあり、から`Count`をびすは、のようにしなければなりません。

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

`this`をすることはできません。

にとって

```
for (initializer; condition; iterator)
```

- `for`ル―プは、がわかっているによくされます。
- ル―プをするに、 `initializer`セクションのステ―トメントはだけされます。
- `condition`セクションには、すべてのル―プのわりにル―プがするか、されるべきかをするためにされるブ―ルがまれています。
- `iterator`セクションは、ル―プのにがこるかをします。

のは、 `for`をのにしてする`for`しています。

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

H
e

l
l
o

.NET Fiddleのライブデモ

forステートメントをするすべてのはオプションです。たとえば、のをしてループをします。

```
for( ; ; )
{
    // Your code here
}
```

initializerセクションには、じタイプであるり、のをめることができます。 conditionセクションは、 boolできるのでできます。 iteratorセクションでは、カンマでってのアクションをできます。

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

こんにちは
hello1
hello12

.NET Fiddleのライブデモ

while

whileは、きクエリがfalseにしいか、またはコードがgoto 、 return 、 breakまたはthrowでされるまで、コードブロックをします。

whileキーワードの

while { コードブロック。 }

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

"whileループ1です。"
"whileループ2です。"
"whileループ3です。"
"whileループ4です。"
"whileループ5です。"

.NET Fiddleのライブデモ

whileループは、まれたコードブロックのにがチェックされるため、 **Entry Controlled**です。つまり、がfalseの、whileループはそのをしません。

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

あるでプロビジョニングせずに`while`をすると、ループまたはループになります。なり、これはけるべきですが、なときにはながじることがあります。

このようなループをするには、のようにします。

```
while (true)
{
//...
}
```

Cコンパイラは、のようなループをすることにしてください。

```
while (true)
{
// ...
}
```

または

```
for(;;)
{
// ...
}
```

に

```
{
:label
// ...
goto label;
}
```

whileループは、ブールboolをすさえすれば、どんなになものであっても、をちうることにしてください。ブールをすをむこともできますは `a == x 'などのとじにされるため。えば、

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

リターン

MSDNreturnステートメントは、されているメソッドのをし、びしのメソッドにコントロールをします。オプションのをすこともできます。メソッドがvoidの、returnはできます。

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}


public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

に

`in`キーワードには3つのがあります。

a `foreach`ののとして、またはLINQクエリののとして

```
foreach (var member in sequence)
{
    // ...
}
```

bジェネリックインタフェースとジェネリックデリゲートのでは、のパラメータにするをする

```
public interface IComparer<in T>
{
    // ...
}
```

cLINQクエリのコンテキストでは、クエリされているコレクションをします

```
var query = from x in source select new { x.Name, x.ID, };
```

をして

2があります`using`、キーワードのを`using statement`して`using directive`

1. **using**ステートメント

   `using`キーワードは、`IDisposable`インターフェイスをする`IDisposable`がににされることをします。 usingステートメントにはのトピックがあります

2. **using**ディレクティブ

   `using`ディレクティブには3つのがあります。usingディレクティブのmsdnページをしてください。 usingディレクティブにはのトピックがあります。

シールされた

`sealed`は、クラスにすると、のクラスがしないようにします。

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

`virtual`メソッドまたはプロパティにすると、 `sealed`は、このメソッドプロパティがクラスでオーバーライドされないようにします。

```
public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}

public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}
```

のサイズ

アンマネージのサイズをバイトでするためにされます。

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2(Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

`static`は、メンバーをするためにされます。メンバーは、アクセスするためにインスタンスするはありませんが、にその、つまり`DateTime.Now`してアクセスします。

`static`は、クラス、フィールド、メソッド、プロパティ、、イベント、およびコンストラクタでできます。

クラスのインスタンスには、クラスのすべてのインスタンスフィールドののコピーがまれていますが、フィールドのコピーは1つのみです。

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new A();
        A c = new A();

        Console.WriteLine(A.count); // 3
    }
}
```

`count`は、 `A`クラスのインスタンスのにしい。

は、クラスのコンストラクターをしたり、データをしたり、だけびすのあるコードをするためにもできます。コンストラクタは、クラスがめてされるにびされます。

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

`static class`は`static`キーワードでマークされ、パラメータをするのメソッドのなコンテナとしてできますが、ずしもインスタンスにびけるはありません。クラスの`static`のため、インスタンスすることはできませんが、 `static constructor`むことができます。 `static class`のはのとおりです。

- することはできません
- `Object`のものからできない
- インスタンスコンストラクタではなくコンストラクタをむことができます
-

メンバーのみをむことができます
- されている

コンパイラはフレンドリーであり、クラスにインスタンスメンバーがするかどうかをにらせます。たとえば、とカナダのメトリックをするクラスがあります。

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

クラスがされている

```
public static class Functions
{
  public static int Double(int value)
  {
    return value + value;
  }
}
```

クラスのすべての、プロパティ、またはメンバーもするがあります。クラスのインスタンスはできません。にクラスをすると、にグループされたのバンドルをできます。

、C6 `static`もにすることができる`using`、メンバとメソッドをインポートします。それらはクラスなしですることができます。

い、 `using static`を`using static`せずに

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
Console
    }

}
```

`using static`を`using static`

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
```

```
    }

}
```

クラスはにですが、のがあります

- クラスがびされると、そのクラスはメモリにロードされ、クラスをするAppDomainがアンロードされるまで、ガベージコレクタをしてすることはできません。
- クラスはインタフェースをできません。

int

`int`は、き32ビットのデータである`System.Int32`のエイリアスです。このデータは、にすべてのCプロジェクトによってにされる`mscorlib.dll`あります。

-2,147,483,6482,147,483,647

```
int int1 = -10007;
var int2 = 2132012521;
```

いです

longキーワードは、き64ビットをすためにされます。 `mscorlib.dll`にする`System.Int64`データのエイリアスです`mscorlib.dll` 、するときにすべてのCプロジェクトによってにされます。

いは、にもにもできます。

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

いは、-9,223,372,036,854,775,808から9,223,372,036,854,775,807までののをでき、がの **int**などができるをえるをするがあるでちます。

ulong

なし64ビットにされるキーワード。これは、 `mscorlib.dll`ある`System.UInt64`データをしています`mscorlib.dll`は、するときにすべてのCプロジェクトによってにされます。

018,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 154467440636094513 15UL;
```

`dynamic`キーワードは、 にされたオブジェクトでされます 。 `dynamic`としてされたオブジェクトは、コンパイルのにし、わりににされます。

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

のでは、デシリアライズされたJSONファイルからデータをにみるために、Newtonsoftのライブラリ Json.NETで`dynamic`をしています。

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

キーワードにはいくつかのがあります。それらの1つはメソッドのです。のでは、string `SayHello` メソッドをします。

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

のアプローチは、りのとじようにびすことです。

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

コンパイルエラーはしませんが、に`RuntimeBinderException`がします。これにするは、クラスをしてメソッドをびすことです。

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

、オーバーライド、

とき

`virtual`キーワードをすると、メソッド、プロパティ、インデクサまたはイベントをクラスでオーバーライドし、なをできます。 メンバーはCではデフォルトではではありません

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

メンバをオーバーライドするには、クラスで`override`キーワードをします。 メンバーのはでなければならないことにしてください

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

メンバーのなるいは、びされたときに、されているのメンバーがコンパイルではなくにされることをします。のオブジェクトのもしたクラスのメンバーは、されるインスタンスになります。

するに、オブジェクトはコンパイルに`BaseClass`としてできますが、に`DerivedClass`インスタンスであれば、オーバーライドされたメンバがされます

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

メソッドのオーバーライドはオプションです。

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

しい

`virtual`としてされたメンバーだけがオーバーライドであり、であるため、メンバーをするクラスはしないをくがあります。

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

この、されたメンバは、オブジェクトのについてコンパイルににされます。

- オブジェクトが`BaseClass`のされているにクラスであっても、 `BaseClass`のメソッドがされます
- オブジェクトがのされている`DerivedClass` 、そのの`DerivedClass`されます。

これは、、じタイプがタイプにされたにタイプにメンバ—がされたであり、そのシナリオでコンパイル**CS0108**がされます。

それがだったは、 `new`キ—ワ—ドをしてコンパイラのをしますそして、のにあなたのをらせます。るいはわらず、 `new`キ—ワ—ドはにコンパイラのをします。

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

きのはオプションではありません

C ++とはなり、 `override`キーワードのはオプションではありません 。

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

ので、のはまた、**CS0108**をきこす`B.Foo()`にきない`A.Foo()` `override`はクラスをオーバーライドして、のをこす、するときに`new`なをしてをすると、のをしたいときにします。は、なをくがあるため、してするがあります。

のコードでもエラーがします。

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

クラスはをすることができます

のコードはにですまれですが。

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
        }
    }

    public class B : A
```

```
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

、BおよびそののをつすべてのオブジェクトはをしてFoo()をし、AのはA.Foo()します。

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

メソッドはプライベートにすることはできません

Cコンパイラは、なをぐのにです。 virtualとしてマークされたメソッドはプライベートにすることはできません。プライベートメソッドはからはえないため、きすることもできませんでした。これはコンパイルにします

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

、っている

awaitキーワードは、Visual Studio 2012でサポートされているC5.0リリースのとしてされました。マルチスレッドをにしたタスクライブラリTPLをしています。 asyncキーワードとawaitキーワードは、のようにじでペアでされます。 awaitキーワードは、のタスクがし、そのがされるまで、のメソッドのをするためにされます。 awaitキーワードをするには、 awaitキーワードをするメソッドにasyncキーワードをするがあります。

asyncとvoidはおめしません。はこちらをごください 。

```
public async Task DoSomethingAsync()
{
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
```

```
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

"にたないプロセスをめる..."

** ... 1れ... **

「にたないプロセスは1000ミリかかる。

`Task`または`Task<T>`りメソッドがののみをす、キーワードの`async`および`await`はできます。

それよりむしろ

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

これをうことがましい。

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

5.0

C5.0では、 `catch`と`finally`では`await`をできません。

6.0

C6.0では、 `catch`と`finally`が`await`ます。

チャー

charはのにされた1です。これは、2バイトのメモリをとするビルトインのです。これは、`mscorlib.dll`ある`System.Char`データをしています`mscorlib.dll`は、にすべてのCプロジェクトによってにされます。

これをうにはのがあります。

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

charはに`ushort, int, uint, long, ulong, float, double,`または`decimal`され、そのcharのがされます。

```
ushort u = c;
```

99などをす

ただし、のからcharへのなはありません。わりにそれらをキャストするがあります。

```
ushort u = 99;
 char c = (char)u;
```

ロック

`lock`は、じプロセスの1つのスレッドだけがアクセスできるように、コードブロックにしてスレッドセーフティをします。

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:

Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

ユースケース

にのスレッドでされた、をきこすのあるコードブロックがあるはいつでも。これをぐには、 オブジェクト このでは`_objLock` とともにlockキーワードをできます。

`_objLock`を`null`することはできません。また、コードをするのスレッドは、じオブジェクトインスタンスをするがあります `static`フィールドにするか、のスレッドでじクラスインスタンスをする

コンパイラーから、lockキーワードは、 `Monitor.Enter(_lockObj);`きえられるな`Monitor.Enter(_lockObj);`および`Monitor.Exit(_lockObj);`。 したがって、これらの2つのでコードブロックをんでロックをきえると、じがられます。あなたはCののコードをることができます- ロックの

ヌル

のは、インスタンスへのなかヌルのいずれかをできます。 nullは、なのデフォルトとnullです。

`null`はnullをすキーワードです。

として、ののにnullをりてることができます

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

NULLのないのにはnullをりてることはできません。のりてはすべてです

```
int a = null;
float b = null;
decimal c = null;
```

ヌルを、のようなさまざまなのなインスタンスとしないでください。

- のリスト `new List<int>()`
- の `""`
- のゼロ `0` 、 `0f` 、 `0m`
- ヌル `'\0'`

によっては、かがnullか、の/ defaultオブジェクトかどうかをすることはがあります。これをするにはSystem.String.IsNullOrEmptyStringメソッドをするか、のメソッドをすることができます。

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null
reference. Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
```

```
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

`internal`キーワードは、およびメンバーのアクセスです。またはメンバーは、じアセンブリのファイルでのみアクセスです

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

さまざまなアクセスのいをここでにします

アクセス

パブリック

またはメンバには、じアセンブリののコードまたはアセンブリをするのアセンブリからアクセスできます。

プライベート

またはメンバは、じクラスまたはのコードでのみアクセスできます。

された

またはメンバにアクセスできるのは、じクラスまたはのコード、またはクラスのみです。

またはメンバは、じアセンブリののコードでアクセスできますが、のアセンブリからはアクセスできません。

された

またはメンバには、じアセンブリののコード、またはのアセンブリののクラスからアクセスできます。

アクセスがされていないは、デフォルトのアクセスがされます。だから、たとえそれがされていなくても、にらかののアクセスがあります。

どこで

`where`ではCで2つのをたすことができますなでをし、LINQクエリをフィルタリングします。

なクラスでは、

```
public class Cup<T>
{
    // ...
}
```

Tはパラメ―タとばれます。クラスは、Tになのにをすことができます。

ののをできます。

- の
-
- デフォルトコンストラクタ
- と

の

この、 `struct` sこれには`int` 、 `boolean`などの'プリミティブ'デ―タがまれていますのみをできます

```
public class Cup<T> where T : struct
{
    // ...
}
```

この、クラス･タイプのみをすることができます

```
public class Cup<T> where T : class
{
    // ...
}
```

ハイブリッド/

をデ―タベ―スでなものにすることがましいがあります。これらのは、とにマップされます。すべてののがたされなければならないため`where T : struct or string` これはなではありませんをすることはできません。このをするには、タイプを "...ブ―ル、SByte、バイト、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double、Decimal、DateTime、Char、およびString"のをつ`IConvertible`にします。 "これはにはまれですが、のオブジェクトがIConvertibleをするがあります。

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

デフォルトコンストラクタ

デフォルトのコンストラクタをむだけがされます。これには、デフォルトパラメ―タなしのコンストラクタをむのとクラスがまれます

```
public class Cup<T> where T : new
{
    // ...
}
```

と

のクラスをしたり、のインタフェースをしているだけをすることができます。

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

はのパラメータをすることさえできます

```
public class Cup<T, U> where U : T
{
    // ...
}
```

にはのをできます。

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

のでは、クラスのジェネリックをしていますが、がされているであれば、クラス、、インタフェース、メソッドなど、どこでもをできます。

whereにもLINQすることができます。このは、SQLのWHEREとています。

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
```

```
    // prints 2 1 3 2 0
```

extern

`extern`キーワードは、でされたメソッドをするためにされます。これをDllImportとみわせてすると、Interopサービスをしてアンマネージコードをびすことができます。このは`static`がいてくるでしょう

えば

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);

    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

これは、User32.dllライブラリからインポートされたSetForegroundWindowメソッドをします。

これは、アセンブリエイリアスをするためにもできます。のアセンブリからじコンポーネントのなるバージョンをすることができます。

じをつ2つのアセンブリをするには、のようにコマンドプロンプトでエイリアスをするがあります。

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

これにより、エイリアスGridV1とGridV2がされます。これらのエイリアスをプログラムでするには、externキーワードをしてしてください。えば

```
extern alias GridV1;
extern alias GridV2;
```

ブール

ブール`true`および`false`をするためのキーワード。 boolはSystem.Booleanのエイリアスです。

boolのデフォルトはfalseです。

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

boolがnullをするには、boolとしてするがあります。

boolのデフォルトはです。

```
bool? a // default value is null
```

いつ

whenはC6でされたキーワードで、フィルタリングにされます。

whenキーワードをするに、のごとに1つのcatchをつことができました。キーワードのにより、よりかいがになりました。

whenはにされたcatch、およびにのみwhenのがあるtrue 、 catchがされます。じクラスタイプをつのcatchをつことができwhenがなるwhenなるwhenます。

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }


CatchException(Method1);
CatchException(Method2);
```

チェックされていない

uncheckedキーワードは、コンパイラーがオーバーフロー/アンダーフローをチェックするのをぎます。

えば

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
```

```
}
int1 = unchecked(ConstantMax + 10);
```

`unchecked`キーワードがなければ、2つののどちらもコンパイルされません。

これはいつですか

これは、オーバーフローのチェックにがかかるため、またはオーバーフロー/アンダーフローがましいたとえば、ハッシュコードのになったときに、オーバーフローしないをするのにちます。

`"void"`は、 `System.Void`のエイリアスで`System.Void` 、2つのがあります。

1. りをたないメソッドをします。

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

りのがvoidのメソッドは、そのに`return`キーワードをつことができます。これは、メソッドのをし、びしにフローをすにです。

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. でないコンテキストのなへのポインタをします。

でないコンテキストでは、はポインタ、、のいずれかです。ポインタは`type* identifier`であり、はの `int* myInt` ですが、がな`void* identifier`で`int* myInt`ん。

voidポインターのは、マイクロソフトによってされません。

if、if ... else、if ... else if

`if`は、プログラムのれをするためにされます。 `if`ステートメントは、 `Boolean`のについてするステートメントをします。

のステートメントの、 `braces` {}はオプションですがされています。

```
int a = 4;
if(a % 2 == 0)
```

```
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

`if`は`else`もあり、がfalseとされたにされます

```
int a = 5;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

`if` ... `else if`をすると、のをできます。

```
int a = 9;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
{
     Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

はのでたされている、コントロールはのテストをスキップして... **else**のは、している、の**construct.So**は、テストのがなは、そののわりにジャンプしますことにすることが

Cブールはをします。これは、がをうにです。

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

`someOtherBooleanMethodWithSideEffects`がにされるというはありません。

のがのものをすることが「」であることをするにもです。えば

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

をりす、はにです。

```
if (someCollection.Count > 0 && someCollection != null) {
```

someCollectionがnull、 NullReferenceExceptionがスローされnull。

う

doは、きクエリがfalseにしくなるまで、コードのブロックをします。 do-whileループは、 goto 、 return 、 breakまたはthrowによってすることもできます。

doキーワードのはのとおりです。

do { コードブロック; } while ;

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

「Doはループ1にあります。
「Doはループ2にあります。
「Doはループ3にあります。
「Doはループ4にあります。
「Doはループ5にあります。

whileループとはなり、do-whileループは**Exit Controlled**です。つまり、がめてしたでも、do-whileループはなくとも1ステートメントをします。

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

オペレーター

みみ をむのは、 operatorキーワードとpublicおよびstaticをしてオーバーロードすることができます。

には、2、の3つのがあります。

およびは、とじのなくとも1つのパラメータをとし、また、あるものはのをとします。

は、みタイプとのでするがあります。

```
public struct Vector32
{

    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{X}, {Y}}}";

}

public struct Vector64
{

    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}, {Y}}}";
```

```
}
```

```
var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2);  // {102, 103}
Console.WriteLine(vector1 - vector2);  // {-72, -25}
```

structは、、のやののなど、するのさなグループをカプセルするためにされるです。

クラスは、はです。

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }

        public void Display(string name)
        {
            WriteLine(name + ": " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

にはコンストラクタ、、フィールド、メソッド、プロパティ、インデクサ、、イベント、ネストされたがまれることがありますが、そのようなメンバがいくつかなは、

をいつうべきか、そしてクラスをいつうべきかについてのMSからのいくつかの

する

のインスタンスがさく、にである、またはにのオブジェクトにめまれているは、クラスのわりにをします。

ける

がのすべてのをたないり、をします。

- これはにのをし、プリミティブint、doubleなどにています。
- インスタンスサイズは16バイトです。
- それはです。
- にボックスにれるはありません。

スイッチ

switchステートメントは、のリストからするスイッチセクションをするコントロールステートメントです。 switchステートメントには、1つまたはのスイッチセクションがまれます。スイッチセクションには、1つのcaseラベルのに1つのステートメントがきます。のラベルにするがまれていないは、 defaultセクションにがりdefaultする。ケースフォールスルーは、Cではにはサポートされていません。ただし、1つのcaseラベルがのcase 、コードをむのcaseブロックのコードにいます。これにより、じでのcaseラベルをグループできます。のでは、 monthが12のcase 、 caseラベル12 1と2がグループされているため、 case case 2のコードがされます。 caseブロックがでない case 、のcaseラベルのにbreakがするがあります。そうでない、コンパイラはエラーにフラグをけます。

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
```

```
    default:
        Console.WriteLine("Incorrect month index");
        break;
}
```

caseはコンパイルにられているえば1 、 "str" 、 Enum.A でしかラベルけできないので、 variableはなcaseラベルではありませんが、 constまたはEnumはanyリテラル。

インタフェーース

interfaceは、メソッド、プロパティ、およびイベントのシグネチャがまれています。クラスは、メンバーののみがインターフェイスにまれるため、メンバーをします。

インタフェースは、 interfaceキーワードをしてされinterface 。

```
interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;

    public Product(decimal price)
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}
```

でない

unsafeキーワードは、やメソッド、またはインラインブロックのにできます。

このキーワードのは、のブロックにしてCのでないサブセットをできるようにすることです。でないサブセットには、ポインタ、スタックりて、Cのようななどのがまれています。

でないコードはではないため、そのはされません。でないコードをコンパイルするには、スイッチをCコンパイラにすがあります。さらに、CLRでは、のアセンブリにながです。

これらのにもかかわらず、でないコードは、いくつかのをよりにするえば、インデックスけか、よりにするえば、いくつかのアンマネージドライブラリをつinteropなをっています。

になとして

```
// compile with /unsafe
class UnsafeTest
{
```

```
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p; // the '*' dereferences the pointer.
      //Since we passed in "the address of i", this becomes "i *= i"
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
      Console.WriteLine(i); // Output: 25
   }
}
```

ポインタをってしているときに、でアドレスするのではなく、メモリのをすることができます。ガ―ベジコレクタがをかしてしまうため、メモリのをぐためにfixedキ―ワ―ドをするがあることにしてくださいそうしないと、エラ―CS0212がするがあります。 ""されたにはきむことができないので、のポインタとじをしすのポインタをつがあることがよくあります。

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++;      //move the pointer to the next memory space.
                      // NOTE that the pointer will move 4 bytes since "p" is an
                      // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;"
        }
    }
}
```

```
1
4
9
```

```
16
25
36
49
64
81
100
```

`unsafe`は、Cランタイムライブラリの_allocaのようなスタックにメモリをりてるstackallocのもします。のをして、 `stackalloc`をのようにすることができます

```
unsafe void Main()
{
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

はとじです

`implicit`キーワードは、のオーバーロードにされます。たとえば、にじてに`double`にされる`Fraction`クラスをし、 `int`からにすることができます

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
```

```
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

`true`と`false`キーワードには2つのがあります。

1. リテラルブール

```
var myTrueBool = true;
var myFalseBool = false;
```

2. になるのある

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Falseのオーバーロードは、 `Nullable`がされるのC2.0ではでした。
`true`にをかけるは、 `false`にもをかけるがあります。

`string`は、テキストのをできる.NETデータ`System.String`エイリアスです。

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

のはUTF-16でエンコードされます。つまり、に2バイトのがです。

ushort

16ビットののをするためにされる。 `ushort`は`System.UInt16`エイリアスで、2バイトのメモリを`ushort`ます。

なは`0`に`65535`。

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

sbyte

8ビットきをするためにされる。 `sbyte`は`System.SByte`エイリアスで、1バイトのメモリを`sbyte`し`System.SByte`。 なしののは、 `byte`し`byte`。

は`-127`に`127` りのをするためにされます。

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

var

ユーザーがをしたかのようにくけされたにされたローカル。のとはなり、コンパイラは、このがりてられているについてこれがすのをします。

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

のタイプのとはなり、このキーワードをつはされたときにするがあります。これは、にされたをす**var**キーワードがです。

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

varキーワードをして、しいデータをにすることもできます。これらのしいデータはとしてられています 。これらのメソッドは、ユーザーがののオブジェクトをにするなく、のプロパティをできるため、にです。

なタイプ

```
var a = new { number = 1, text = "hi" };
```

をす*LINQ*クエリ

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName  { get; set; }
}

public void GetDogsWithBreedNames()
```

```
{
    var db = new DogDataContext(ConnectString);
    var result = from d in db.Dogs
             join b in db.Breeds on d.BreedId equals b.BreedId
             select new
                    {
                        DogName = d.Name,
                        BreedName = b.BreedName
                    };

    DoStuff(result);
}
```

foreachではvarキーワードをできます

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

デリゲート

デリゲートは、メソッドへのをすです。メソッドをとしてのメソッドにすためにされます。

は、メソッド、インスタンスメソッド、メソッド、またはラムダをできます。

```
class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
```

```
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}
```

デリゲートにメソッドをりてる、メソッドはパラメータとじりのをたなければならないことにすることがです。これは、メソッドのシグネチャをするパラメータだけが「の」メソッドのオーバーロードとはなります。

ィベントはデリゲートのにされます。

イベント

eventによって、はパターンをできます。

な

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}

public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

MSDNリファレンス

な

キーワード`partial`は、クラス、、またはインタフェースのにして、をのファイルにすることができます。これはされたコードにしいをみむのにです。

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

クラスはののファイルにできます。ただし、すべてのはじとじアセンブリのになければなりません。

メソッドは`partial`キーワードをして`partial`とすることもできます。この、1つのファイルにはメソッドのみがまれ、のファイルにはがまれます。

> なメソッドは、のあるでされたシグネチャと、そののでされたそのをっています。なメソッドにより、クラスデザイナーは、イベントハンドラーとに、がするかどうかをめるメソッドフックをすることができます。がをしない、コンパイラはコンパイルにをします。なには、のがされます。
>
> - ののはするがあります。
> - このメソッドはvoidをすがあります。
> - アクセスはできません。なメソッドはにプライベートです。
>
> - MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

メソッドをむもするがあります。

オンラインでキーワードをむ https://riptutorial.com/ja/csharp/topic/26/キーワード

67: キャッシング

Examples

MemoryCache

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

オンラインでキャッシングをむ https://riptutorial.com/ja/csharp/topic/4383/キャッシング

68: コード

1. Contract.RequiresCondition、userMessage

   Contract.RequiresCondition、userMessage

   Contract.Result <T>

   Contract.Ensures

   Contract.Invariants

.NETは、System.DiagnosticsにあるContractsクラスをしてDesign by Contractアイデアをサポートし、.NET 4.0でされました。コードコントラクトAPIには、コードのチェックとランタイムチェックのクラスがまれており、メソッドの、、をすることができます。は、メソッドがされるにパラメータがたさなければならない、メソッドのにされる、メソッドのにしないをですることをします。

コードがなのはなぜですか

アプリケーションがされているときのアプリケーションのにするは、すべてのとのです。トラッキングはくのでできます。えば -

- アプリケーションでトレースをし、アプリケーションのにアプリケーションのをすることができます
- アプリケーションのは、イベントログをできます。イベントビューアをしてメッセージをできます
- ののにPerformance Monitoringをし、アプリケーションからライブデータをきむことができます。

コードコントラクトは、アプリケーションののとになるアプローチをします。、、およびメソッドののけをりて、メソッドびし、コードからされたすべてをするのではなく、メソッドへのりがすべてしいことをしてください。

Examples

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();
```

```
        public void Add(Payment payment)
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

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

        public Point()
        {
        }

        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
```

```
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }

            Console.WriteLine("X = {0}", this.X);
            Console.WriteLine("Y = {0}", this.Y);
        }

        [ContractInvariantMethod]
        private void ValidateCoordinates()
        {
            Contract.Invariant(this.X >= 0);
            Contract.Invariant(this.Y >= 0);
        }
    }
}
```

インタフェースでのの

```
[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer
ID cannot be null!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password
cannot be null!!");
        }
    }
```

```
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID),"Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID),
"Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }

    }

    private string m_custID, m_PWD;

    public string CustomerID
    {
        get
        {
            return m_custID;
        }
        set
        {
            m_custID = value;
        }
    }

    public string Password
    {
        get
        {
            return m_PWD;
        }
        set
        {
            m_PWD = value;
        }
    }
}
```

のコードでは、 `[ContractClass]`というをつ`IValidation`というインターフェースをしました。このは、インタフェースのコントラクトをしたクラスのアドレスをります。 `ValidationContract`クラスは、インタフェースでされたプロパティをし、 `Contract.Requires<T>`をしてnullをチェックします。 `T`はクラスです。

getアクセサに`[Pure]`をけました。なは、メソッドまたはプロパティが`IValidation`インターフェイスがされているクラスのインスタンスをしないことをします。

オンラインでコードをむ https://riptutorial.com/ja/csharp/topic/4241/コード

69: コードとアサーション

Examples

をチェックするアサーションはにでなければなりません

アサーションは、パラメータのテストをするのではなく、プログラムフローがcorectであることをするためにされます。つまり、のでコードにするのをうことができます。いえれば、`Debug.Assert`われたテストは、 にテストされたが`true`としなければなりません 。

Debug.AssertはDEBUGビルドでのみされます。 RELEASEビルドからされます。これは、テストにえて、コードやのきえとしてではなく、デバッグツールとなすがあります。

えば、これはいアサーションです

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

ここでは、RetrieveSystemConfigurationがなをし、してnullをさないとできるので、assertはいです。

もうつのいがあります

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

まず、RetrieveUserDataがなをすとします。に、Ageプロパティをするに、ユーザーのがにであるというにであるべきをします。

これはassertのいです

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

アサーションはのためのものではありません。なぜなら、このアサーションがにであるとするのはっているからです。そのためには、メソッドをするがあります。の、がのであることもするがあります。

オンラインでコードとアサーションをむ https://riptutorial.com/ja/csharp/topic/4349/コードとアサーション

70: コメントと

Examples

コメント

プロジェクトのコメントをすると、のをするながられ、コードをしたりしたりするときに、のをにすることをすべきです。

コードにコメントをするには2りのがあります。

のコメント

//にかれたテキストはコメントとしてわれます。

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

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

のまたはられたコメント

/*と*/のテキストはコメントとしてわれます。

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

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

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

```
}
```

リージョンはりたたみなコードブロックで、コードのとにちます。

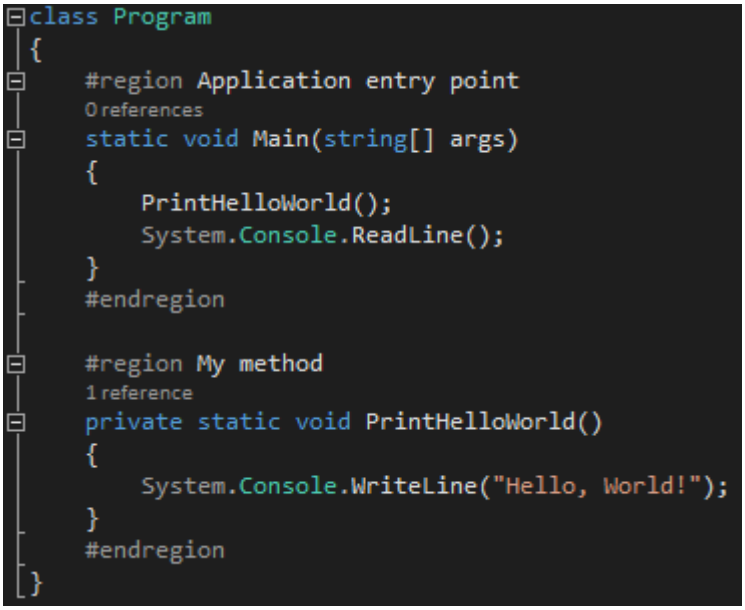
StyleCopのルールSA1124 DoNotUseRegionsはのをします。 Cにはクラスやそののがまれているため、はひどくされたコードのです。

リージョンはのでできます。

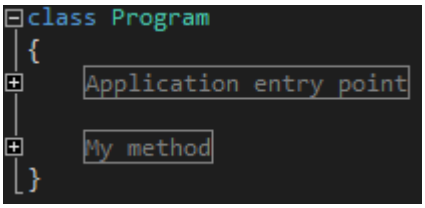
```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

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

のコードをIDEでると、+と - のをしてコードをりたたんですることができます。

された



りたたまれた

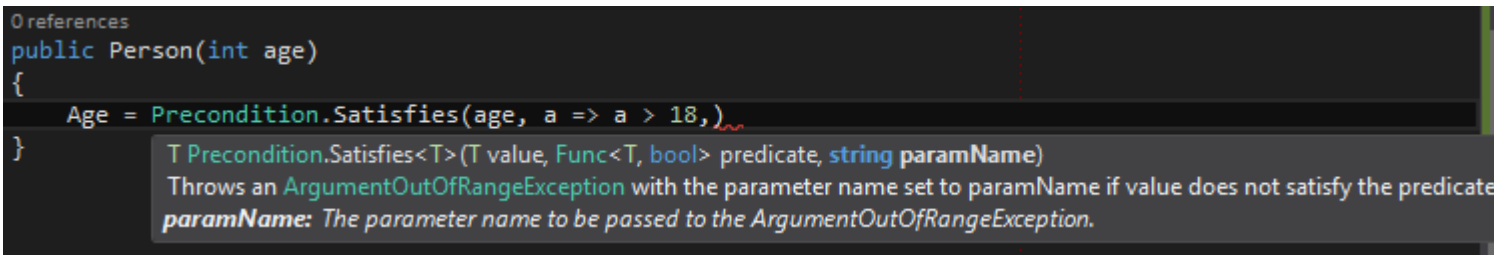


ドキュメントのコメント

XMLドキュメンテーションコメントは、ツールでにできるAPIドキュメントをするためにできます。

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

        return value;
    }
}
```

IntelliSenseによってドキュメントがにされます



オンラインでコメントとをむ https://riptutorial.com/ja/csharp/topic/5346/コメントと

71: コレクション

このなをしてオブジェクトをするのは、が`System.Collections.IEnumerable`と`Add`メソッドをすることです。コレクションとばれますが、オブジェクトはコレクションであるはありません。

Examples

コレクションの

コレクションをでする

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

コレクションは、`Add()`びしのです。のコードはのものとです

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

をけるために、はをしてにわれることにしてください。

`Add()`メソッドでのパラメータをするのは、コンマりのをでみます。

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

これはのようになります。

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

C6インデックスイニシャライザ

C6、インデクサをつコレクションは、するインデックスをでし、そのにとそれにくをすることでできます。

の

をったこのの

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

これはのようになります。

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

C6よりにこれをうコレクションのはのとおりです。

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

これは、

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

したがって、しいでは、されたオブジェクトのインデクサーをして`Add()`メソッドをするわりにをりてるため、にきないがあります。つまり、しいではされているインデクサのみがであり、それをつオブジェクトであればします。

```
public class IndexableClass
{
    public int this[int index]
    {
        set
        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }
}

var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}
```

これはします

```
10 was assigned to index 0
20 was assigned to index 1
```

カスタムクラスのコレクション

クラスがコレクションをサポートするようにするには、 `IEnumerable`インターフェイスをし、なくとも1つの`Add`メソッドをたなければなりません。 C6、 `IEnumerable`をしているコレクションは、メソッドをしてカスタム`Add`メソッドでできます。

```
class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
}
```

パラメータをつコレクション

のパラメータとパラメータをさせることができます

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

このはです

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool"  , 35663, 35732, 12312, 75685},
    {"Bruce"    , 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

オブジェクトイニシャライザでのコレクションの

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms`はコレクションのプロパティです。 `Tag`オブジェクトがオブジェクトイニシャライザをしてされるとき、 `Synonyms`はコレクションイニシャライザですることもできます。

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

コレクションプロパティはみりで、コレクションのをサポートします。このされたをえてみましょう `Synonyms`プロパティはプライベートセッターをっています

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }

    public IList<string> Synonyms { get; private set; }
}
```

のようにしい`Tag`オブジェクトをできます

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

これは、コレクションのが`Add()`へのびしよりもなにすぎないためにします。ここではしいリストはされません。コンパイラは、のオブジェクトに`Add()`をびすだけです。

オンラインでコレクションをむ https://riptutorial.com/ja/csharp/topic/21/コレクション

72: コンストラクタとファイナライザ

き

コンストラクター—は、そのクラスのインスタンスがされるときにびされる、クラスのメソッドです。なは、しいオブジェクトをかつしたにすることです。

デストラクタ/ファイナライザは、クラスのインスタンスがされたときにびされるクラスのメソッドです。 Cでは、に/されることはほとんどありません。

Cにはデストラクタがにはなく、C ++スタイルのデストラクタをするFinalizersがあります。デストラクタをすると、びすことのできない`Object.Finalize()`メソッドがオ—バ—ライドされます。

のをつのとはなり、オブジェクトはスコープにたときにはびされませんが、のではガベ—ジコレクタがされたときにびされます 。したがって、のですることはされていません。

ファイナライザはMarshalクラスをしてポインタが、P /びしシステムコール、またはなブロックでされるのポインタをして のみアンマネ—ジリソ—スをクリ—ンアップするためのをわなければなりません。リソ—スをクリ—ンアップするには、IDisposable、Disposeパターン、 `using`ステ—トメントをし`using`ください。

しい いつデストラクタをすればよいですか

Examples

デフォルトコンストラクタ

がコンストラクタなしでされている

```
public class Animal
{
}
```

コンパイラはのようなデフォルトのコンストラクタをします。

```
public class Animal
{
    public Animal() {}
}
```

ののコンストラクタのは、デフォルトコンストラクタのをします。タイプがのようにされている

```
public class Animal
{
    public Animal(string name) {}
```

```
}
```

`Animal`は、されたコンストラクタをびすことによってのみできます。

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

2のでは、コンパイラはエラーメッセージをします。

> 'Animal'には0のをるコンストラクタがまれていません

クラスにパラメータのないコンストラクタとパラメータをけるコンストラクタのをたせたいは、のコンストラクタをにすることでそれをうことができます。

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

クラスがパラメータのないコンストラクタをたないのクラスをする、コンパイラはデフォルトコンストラクタをできません。えば、たちが`Creature`クラスをっていれば

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

その、 `Animal`としてされ`class Animal : Creature {}`コンパイルされないであろう。

のコンストラクタからコンストラクタをびす

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
}

var dog = new Animal();      // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

コンストラクタ

コンストラクタは、のいずれかのメンバがにされるとき、クラスメンバがびされるとき、またはメソッドがびされるときにびされます。コンストラクタはスレッドセーフです。コンストラクタは、にのでされます。

- をします。つまり、じクラスのなるインスタンスでされるです。
- シングルトンをする

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

の
がされました
がされました

デモをる

のびしがメソッドの、コンストラクター—はインスタンスコンストラクター—なしでびされます。メソッドはインスタンスのにアクセスできないため、これはありません。

```
Animal.Yawn();
```

これはされます

の
ハーン

コンストラクタとコンストラクタの もしてください。

シングルトンの

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

クラスのコンストラクタをびす

クラスのコンストラクタは、クラスのコンストラクタがされるにびされます。たとえば、 `Mammal` が`Animal`する、 `Mammal`インスタンスをするときに、 `Animal`のコンストラクタにまれるコードがにびされます。

クラスが、びされるクラスのコンストラクタをにしていない、コンパイラはパラメータのないコンストラクタをします。

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

この、 `new Mammal("George the Cat")`をびして`Mammal`インスタンスすると、

のがまれます。
ジョージザキャットはです。

デモをる

クラスののコンストラクタをびすには、コンストラクタのシグネチャとボディのに：`base(args)`します。

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

`new Mammal("George the Cat")`をびすとのようにされます

ジョージ･ザ･キャットがまれる。
ジョージザキャットはです。

デモをる

クラスのファイナライザ

オブジェクトグラフがファイナライズされると、そのはのです。たとえば、のコードですように、スーパータイプはベースタイプのにファイナライズされます。

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

シングルトンコンストラクタパターン

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

コンストラクターはプライベートなので、コードをすることで`SingletonClass`のしいインスタンスをすることはできません。 `SingletonClass`のインスタンスにアクセスするのは、プロパティ`SingletonClass.Instance`をすることです。

`Instance`プロパティは、Cコンパイラがするコンストラクタによってりてられます。 .NETランタイムは、コンストラクター—がで1され、 `Instance`がにみられるにされることをします。したがって、すべてのとのは、ランタイムによってされます。

スタティックコンストラクタがした、 `Singleton`クラスはAppDomainのにできなくなることにしてください。

また、コンストラクター—は、 `Instance`のアクセスにされるはありません。むしろ、それはそののあるでされます。これにより、がにこるがじます。の、JITは、 `Instance`するメソッドのコンパイルではなくにコンストラクタをびすことがよくあります。これはパフォ—マンスのです。

シングルトンパター—ンをするのについては、「 シングルトン 」ペ—ジをしてください。

コンストラクタをにびす

コンストラクタはにののののにびされますが、にびすことができるとなことがあり、 `RuntimeHelpers`クラスはヘルパ—をします

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

すべてのフィ—ルドイニシャライザなどは、コンストラクタだけでなく、されます。

な UIアプリケ—ションのスプラッシュでをするか、スタティックコンストラクター—がテストでしないようにします。

コンストラクタでメソッドをびす

CのC ++とはなり、クラスコンストラクタからメソッドをびすことができますこれはC ++でもですが、のはくべきことです。えば

```
abstract class Base
{
    protected Base()
    {
        _obj = CreateAnother();
    }

    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }
```

```
    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

あなたがC ++のバックグラウンドからたのであればこれはくべきことですが、クラスのコンストラクタはにクラスのメソッドテ―ブルをています

クラスはまだにはされていないかもしれませんそのコンストラクタはクラスのコンストラクタのにされます。このテクニックはですこのためのStyleCopもあります。、これはいとみなされます。

なコンストラクタ

コンストラクタがされているがである、コンストラクタはののみわせごとに1びされます。

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

これはされます

    System.Object
    System.String

See alsoもジェネリックのコンストラクタはどのようにしますか

コンストラクタの

コンストラクター―がをスロ―したは、されません。このは、AppDomainのはできません。のこれのは、のをむ`TypeInitializationException`をさせます。

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }
```

```
    public static void Yawn() {}
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

これはされます

> スタティックctor
>
> System.TypeInitializationException 'Animal'のがをスローしました。 --->
> System.Exception 'System.Exception'のがスローされました。

[...]

> System.TypeInitializationException 'Animal'のがをスローしました。 --->
> System.Exception 'System.Exception'のがスローされました。

のコンストラクタはだけされ、はされていることがわかります。

コンストラクタとプロパティの

クラスのコンストラクタのまたはに 、プロパティのをしますか

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
```

```
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

のでは、クラスのコンストラクタまたはクラスコンストラクタのに`TestProperty`を`1`にするがありますか

のようにインスタンスにプロパティをりてる

```
var testInstance = new TestClass() {TestProperty = 1};
```

コンストラクタのにされます。ただし、C6.0のクラスのプロパティでプロパティをのようにします。

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

コンストラクタがされるにされます。

の2つのを1つのでみわせると、

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

static void Main(string[] args)
{
    var testInstance = new TestClass() { TestProperty = 1 };
    Console.WriteLine(testInstance.TestProperty); //resulting in 1
}
```

```
"Or shall this be executed"
"1"
```

`TestProperty`は、に`2`としてりてられ、に`TestClass`コンストラクタがされ、

```
"Or shall this be executed"
```

そしてその`TestProperty`ようにりてられる`1`によるに`new TestClass() { TestProperty = 1 }`の`TestProperty`によって`Console.WriteLine(testInstance.TestProperty)`であることが

```
"1"
```

オンラインでコンストラクタとファイナライザをむ https://riptutorial.com/ja/csharp/topic/25/コンストラクタとファイナライザ

73: ジェネリックス

- `public void SomeMethod <T> () { }`
- `public void SomeMethod<T, V>() { }`
- `public T SomeMethod<T>(IEnumerable<T> sequence) { ... }`
- `public void SomeMethod<T>() where T : new() { }`
- `public void SomeMethod<T, V>() where T : new() where V : struct { }`
- `public void SomeMethod<T>() where T: IDisposable { }`
- `public void SomeMethod<T>() where T: Foo { }`
- `public class MyClass<T> { public T Data {get; set; } }`

パラメーター

パラメーター	
T、V	ジェネリックのプレースホルダの

らのなセマンティクスもにコンパイルされ、されますのCでされたジェネリックCでジェネリックがダウンし、にすべてのはサポートされているCILを 。

これは、、Cではジェネリックをし、されたときにオブジェクトをしたり、オブジェクトがジェネリックのインスタンスであるかどうかをチェックすることができます。これは、コンパイルにジェネリックのがされるとはです。これは、にのなジェネリックがのジェネリックになり、のジェネリックのをさらにインスタンスするためになメタデータがわれるジェネリックのテンプレートアプローチとはです。

しかし、ジェネリックにするときはがです。ジェネリックのはコンパイルにされ、とパラメータのをバックティックとジェネリックパラメータのできえます。したがって、 `Dictionary<TKey, Tvalue>`は`Dictionary`2`にされる。

Examples

パラメータクラス

```
class MyGenericClass<T1, T2, T3, ...>
{
    // Do something with the type parameters.
}
```

```
var x = new MyGenericClass<int, char, bool>();
```

パラメータのとして

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

パラメータメソッド

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

びし

コンパイラがにをできるので、にをえるはありません。

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

しかし、あいまいさがあるは、ジェネリックメソッドをtype arguemntsでびすがあります。

```
MyGenericMethod<int, int, string>(x,y,z);
```

タイプパラメータインタフェース

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

パラメータのとして

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

のメソッド

なをメソッドにすとき、するのは、、にできます。すべてのジェネリックをできるは、でることはオプションです。

のなをえてみましょう。 1つのパラメータと1つのジェネリックパラメータがあります。それらのにはにながあります。ジェネリックパラメータへのとしてされるは、パラメータにされるのコンパイルとじでなければなりません。

```
void M<T>(T obj)
{
}
```

これらの2つのびしはです。

```
M<object>(new object());
M(new object());
```

これらの2つのびしもです。

```
M<string>("");
M("");
```

これらの3つのびしもです。

```
M<object>("");
M((object) "");
M("" as object);
```

なくとも1つのができないは、それらのすべてをするがあることにしてください。

のなをえてみましょう。のジェネリックは、のとじです。しかし、2のジェネリックのにはそのようなはありません。したがって、コンパイラは、このメソッドへのびしで2ジェネリックのをするがありません。

```
void X<T1, T2>(T1 obj)
{
}
```

これはもうしません

```
X("");
```

これは、どちらもしません。コンパイラは、1または2ジェネリックパラメータをしているかどうかわかりませんどちらも`object`としてです。

```
X<object>("");
```

たちは、のようにをタイプするがあります

```
X<string, object>("");
```

のクラスとインタフェース

は、パラメータがのインタフェースまたはクラスをするようにすることができます。

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
```

```
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

のの

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

は、とじでします。つまり、のとしてのインタフェースをできますが、クラスは1つのみです。

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

もう1つのルールは、クラスをのとしてし、にインターフェイスをするがあるということです。

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

のインスタンスがするためには、されたはすべてにたされなければなりません。 2つのセットをするはありません。

クラスと

それぞれの`class`または`struct`をして、をまたはにするがあるかどうかをすることができます。これらのがされているは、のすべてのたとえばのまたは`new()` をリストするにする があります。

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

タイプ**new-keyword**

`new()`をすることによって、のデフォルトのコンストラクタをするパラメータをすることができます。

```
class Foo
{
    public Foo () { }
}

class Bar
{
    public Bar (string s) { ... }
}
```

```
class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty
constructor.
```

`Create()`への2のびしでは、のメッセ―ジとともにコンパイルエラ―がします。

> 'Bar'は、ジェネリックまたはメソッド 'Factory'でパラメ―ター― 'T'としてするために、パブリックパラメ―タ―なしコンストラクタ―をつでなければなりません

パラメ―タをつコンストラクタのはなく、パラメ―タのないコンストラクタのみがサポ―トされています。

クラス

は、がコンストラクタではしないというかられることができます

```
class Tuple<T1,T2>
{
   public Tuple(T1 value1, T2 value2)
   {
   }
}

var x = new Tuple(2, "two");              // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

パラメ―タをにせずにインスタンスをするのは、のようなコンパイルエラ―をきこします。

> ジェネリック 'Tuple <T1、T2>'をうには2つのがです

なは、クラスにヘルパ―メソッドをすることです。

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
         return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two");  // This WILL work...
```

パラメ―タを

typeofはパラメータにします。

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

なパラメータ

ジェネリックメソッドのパラメータをにするがあるがあります。のの、コンパイラは、されたメソッドパラメータからすべてのパラメータをすることができません。

1つのケースは、パラメータがないです。

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
}

SomeMethod(); // doesn't compile
SomeMethod<int, bool>(); // compiles
```

2のケースは、パラメータの1つまたはそれがメソッドパラメータのではないです。

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

インタフェースをタイプとしてするメソッドの。

これは、クラスAnimalのEatメソッドでジェネリックのTFoodをするのです

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}
```

```
public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

Eatメソッドはのようにびすことができます

```
var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore
```

この、をしようとすると

```
sheep.Eat(lion);
```

オブジェクトライオンがインタフェースIFoodをしていないため、これはです。のびしをみるとコンパイラエラーがします " 'Carnivore'は、またはメソッド 'Animal.EatTFood'のパラメータ 'TFood' としてできません ' 「Carnivore」から「IFood」まで。

IEnumerable<T>がのIEnumerable<T1>サブタイプであるのはいつですか TがT1サブタイプである。 IEnumerableはそのTパラメータでです。 つまり、 IEnumerableのサブタイプのはT とじになります。

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>
```

えられたパラメータをつジェネリックのインスタンスは、パラメータがないじジェネリックににです。

IEnumerable は T sをするが、それらをしないため、このがりつ。 Dogをするオブジェクトは、Animalするとにできます。

タイプのパラメーターは、 としてのみするがあるため、 outキーワードをしoutされます 。

```
interface IEnumerable<out T> { /* ... */ }
```

としてされたパラメータは、としてはれません。

```
interface Bad<out T>
{
    void SetT(T t);  // type error
}
```

ここにながあります

```
using NUnit.Framework;

namespace ToyStore
{
   enum Taste { Bitter, Sweet };

   interface IWidget
   {
      int Weight { get; }
   }

   interface IFactory<out TWidget>
       where TWidget : IWidget
   {
      TWidget Create();
   }

   class Toy : IWidget
   {
      public int Weight { get; set; }
      public Taste Taste { get; set; }
   }

   class ToyFactory : IFactory<Toy>
   {
      public const int StandardWeight = 100;
      public const Taste StandardTaste = Taste.Sweet;
```

```
      public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste };
}
   }

   [TestFixture]
   public class GivenAToyFactory
   {
      [Test]
      public static void WhenUsingToyFactoryToMakeWidgets()
      {
         var toyFactory = new ToyFactory();

         //// Without out keyword, note the verbose explicit cast:
         // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

         // covariance: concrete being assigned to abstract (shiny and new)
         IFactory<IWidget> widgetFactory = toyFactory;
         IWidget anotherToy = widgetFactory.Create();
         Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
         Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); //
concrete contract
      }
   }
}
```

コントラスト

IComparer<T>がなるIComparer<T1>サブタイプであるときはいつですか T1がTサブタイプである。IComparerはそのTパラメーターでです。つまり、 IComparerのサブタイプのはT とになります。

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

えられたパラメーターをつジェネリックのインスタンスは、よりのパラメーターをつじジェネリックににです。

このは、 IComparer T sをするが、それらをしないためにされる。の2つのAnimalをできるオブジェクトをして、2つのDogをすることができます。

contravariantのパラメーターは、 inキーワードをしてされます 。これは、パラメーターがとしてのみされるがあるためです。

```
interface IComparer<in T> { /* ... */ }
```

としてされたパラメーターは、としてれないかもしれません。

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

IList<T>はしてなるIList<T1>サブタイプではありません。 IListはパラメ―タ―でです。

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

あなたがリストにをれて、リストからをることができますので、リストにはサブタイプのはありません。

IListがIListである、 ったサブタイプのをのリストにすることができます。

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

IListがである、えられたリストからったサブタイプのをすることができます。

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() };  // if this were allowed...
Dog dog = dogs[1];  // ... then this would be allowed, which is bad!
```

パラメ―タは、 inとoutキ―ワ―ドのをすることによってされます。

```
interface IList<T> { /* ... */ }
```

バリアントインタフェ―ス

インタフェ―スにはパラメ―タがあります。

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

クラスとは

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
```

```
}
```

ジェネリックメソッドもしない

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
    {
        // ...
    }
}
```

のは、じインタフェースののをしています

```
interface IFoo<in T1, out T2, T3>
//  T1 : Contravariant type
//  T2 : Covariant type
//  T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

は、バリアントパラメータをつことができます。

```
delegate void Action<in T>(T t);    // T is an input
delegate T Func<out T>();           // T is an output
delegate T2 Func<in T1, out T2>();  // T1 is an input, T2 is an output
```

これは、 Liskov Substitution Principle Liskov Substitution Principle につい ています。これは、にDは、のにメソッドBよりもかれたとえることができるとべています。

- DはBとしいまたはよりくのをします。
- Dは、Bとのなするパラメータタイプをつ

したがって、のりてはすべてです。

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

パラメータとりとしてのバリアント

がとしてする、そのはである。プロデューサー$_T$ Sはのようなものである$_T$。

```
interface IReturnCovariant<out T>
{
```

```
    IEnumerable<T> GetTs();
}
```

contravariantがとしてれる、はです。`T` Sはかかるようである`T`。

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

がとしてする、するはである。プロデューサー`T` Sはかかるようである`T`。

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

がとしてれる、はである。`T` Sは、のようである`T`。

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

なののチェック

ジェネリッククラスまたはメソッドのロジックが、ジェネリックをつのをチェックするがあるは、`EqualityComparer<TType>.Default` プロパティをします。

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
}
```

`TBar`が`IEquatale<TBar>` インタフェースをしているかどうかをチェックし、そうであれば`IEquatable<TBar>.Equals(TBar other)`メソッドをびすため、このアプローチはに`Object.Equals()`メソッドをびすよりもれています。これにより、ののボクシング/ボックスをできます。

```
    /// <summary>
    /// Converts a data type to another data type.
    /// </summary>
    public static class Cast
    {
        /// <summary>
        /// Converts input to Type of default value or given as typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
```

```
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (input == null || input == DBNull.Value) return result;
                if (typeof (T).IsEnum)
                {
                    result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
                }
                else
                {
                    result = (T) Convert.ChangeType(input, typeof (T));
                }
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }

        /// <summary>
        /// Converts input to Type of typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input)
        {
            return To(input, default(T));
        }



    }
```

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);


// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

ジェネリックキャスティングのリーダー

```
    /// <summary>
    /// Read configuration values from app.config and convert to specified types
    /// </summary>
    public static class ConfigurationReader
    {
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>AppSettings value against key is returned in Type of default value or
given as typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (ConfigurationManager.AppSettings[strKey] != null)
                    result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <returns>AppSettings value against key is returned in Type given as typeparam
T</returns>
        public static T GetConfigKeyValue<T>(string strKey)
        {
            return GetConfigKeyValue(strKey, default(T));
        }

    }
```

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

オンラインでジェネリックスをむ https://riptutorial.com/ja/csharp/topic/27/ジェネリックス

74: シングルトン

Examples

にされたシングルトン

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

このはスレッドセーフです。この、 `instance`オブジェクトはコンストラクタでされるためです。CLRは、すべてのコンストラクターがスレッドセーフでされるようにします。

の`instance`はスレッドセーフなではないため、 `readonly`はのをします。

レイジーでスレッドセーフなシングルトンDouble Checked Lockingを

このシングルスレッドのスレッドセーフバージョンは、 `static`がスレッドセーフであることがされていないバージョンの.NETではでした。よりなバージョンのフレームワークでは、のパターンでいをこしやすいので、はにされたシングルトンがされます。

```
public sealed class ThreadSafeSingleton
{
   private static volatile ThreadSafeSingleton instance;
   private static object lockObject = new Object();

   private ThreadSafeSingleton()
   {
   }

   public static ThreadSafeSingleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (lockObject)
            {
               if (instance == null)
               {
                  instance = new ThreadSafeSingleton();
               }
            }
         }

         return instance;
      }
   }
}
```

`if (instance == null)`チェックは、ロックをするに1、そのに1、2われます。このは、のヌルチェックをわなくてもスレッドセーフになります。しかし、これは、インスタンスがされるたびにロックがされ、パフォーマンスがするになることをします。のヌルチェックは、でないりロックがされないようにされます。 2のヌルチェックでは、ロックをするのスレッドだけがインスタンスをすることをします。のスレッドは、インスタンスをしてにスキップすることができます。

スレッドセーフなシングルトン**Lazy**を

.Net 4.0レイジーはスレッドセーフなオブジェクトのをしているので、このをしてシングルトンをることができます。

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

`Lazy<T>`をすると、オブジェクトがびしコードのどこかでされているにのみインスタンスされます。

ないはのようになります

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

.NET Fiddleのライブデモ

のある、スレッドセーフなシングルトン**.NET 3.5**、

.NET 3.5では、 `Lazy<T>`クラスをたないため、のパターンをします。

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {
    }

    public static Singleton Instance
```

```
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

これはJon Skeetのブログからインスピレーションをています。

`Nested`クラスはネストされ、プライベートなので、 `Sigleton`クラスののメンバーにアクセスすることによってえば、public readonlyプロパティなど、シングルトンインスタンスのインスタンスはトリガーされません。

Singletonインスタンスがになったときにする

ほとんどのでは、アプリケーションがもはやそのオブジェクトをとしなくても、アプリケーションがするまで`LazySingleton`オブジェクトをインスタンスしてしています。これにするは、`IDisposable`をし、のようにオブジェクトインスタンスをnullにすることです。

```
public class LazySingleton : IDisposable
{
    private static volatile Lazy<LazySingleton> _instance;
    private static volatile int _instanceCount = 0;
    private bool _alreadyDisposed = false;

public static LazySingleton Instance
{
    get
    {
        if (_instance == null)
            _instance = new Lazy<LazySingleton>(() => new LazySingleton());
        _instanceCount++;
        return _instance.Value;
    }
}

private LazySingleton() { }

// Public implementation of Dispose pattern callable by consumers.
public void Dispose()
{
    if (--_instanceCount == 0) // No more references to this object.
    {
       Dispose(true);
```

```
        GC.SuppressFinalize(this);
    }
}

// Protected implementation of Dispose pattern.
protected virtual void Dispose(bool disposing)
{
    if (_alreadyDisposed) return;

    if (disposing)
    {
        _instance = null; // Allow GC to dispose of this instance.
        // Free any other managed objects here.
    }

    // Free any unmanaged objects here.
    _alreadyDisposed = true;
}
```

のコードは、アプリケーションがするのインスタンスをしますが、コンシューマがにオブジェクトにして`Dispose()`をびすにります。これがこるというやそれをするはないので、インスタンスがされるというもありません。しかし、このクラスがでされているは、`Dispose()`メソッドがするたびにびされることをするがです。をにします。

```
public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}
```

このはスレッドセーフではありません 。

オンラインでシングルトンをむ https://riptutorial.com/ja/csharp/topic/1192/シングルトン

75: ステートメントの

き

IDisposableオブジェクトのしいをするなをします。

- {いて} {}をって
- IDisposable disposable = new MyDisposable{}をして

usingステートメントのオブジェクトは、 IDisposableインターフェイスをするがあります。

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

IDisposableのよりなは、 MSDNのドキュメントをしてください 。

Examples

ステートメントのの

usingは、なtry-finallyブロックをとせずにリソースがクリーンアップされることをするためのです。つまり、コードがはるかにクリーンになり、されていないリソースがれることはありません。

IDisposableインターフェイス FileStreamのクラスStreamが.NETでうをするIDisposableにしては、Disposeクリーンアップパターンをします。

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;

        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
```

```
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

usingはなtry-finallyすことでusingします

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

じようにfinallyブロックはに、エラーやリターンになくusing 、にびすDispose()しても、エラーがしたには、

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}
```

Disposeはコードフローになくびされるため、 IDisposableをするときにDisposeをスローしないようにすることをおめします。さもなければ、のはしいによってオーバーライドされ、デバッグのとなってしまいます。

ブロックをしてる

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

try..finallyののため、 usingブロックがされ、 returnはりにします。りは、 finallyブロックがされてがtry..finallyされるfinallyされます。のはのとおりです。

1. tryボディをする
2. りをしてキャッシュする
3. finallyブロックをする

4. キャッシュされたりをします。

ただし、`disposable`は、でされたをんでいるため、すことはできません - するをしてください。

1つのブロックでのusingステートメント

ネストされたののレベルをせずに、のネストされた`using`ステートメントをすることはです。えば

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

のとして、のようにします。

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

これはのとまったくじです。

ネストされた`using`ステートメントは、Microsoft Code Analysisルールを CS2002 この をトリガーし、をすることがあります。リンクされたえでしたように、ステートメント`using`してネストするのはにです。

`using`ステートメントのがじのは、それらをコンマでり、を1だけできますこれはしいことですが。

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

これは、にがあるにもできます。

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

のでは、 `var`キーワードはできません 。コンパイルエラーがします。されたになるのがある、コンマりのさえしません。

Gotchaあなたがしているリソースをす

`db`をすにそれをするので、はいえです。

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

これにより、さらにないがするがあります。

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

これはですが、LINQのがし、でとなる`DBContext`がすでに`DBContext`されているにのみされるがあります。

つまり、`using`するにされません。このをするの1つは、 `using`をしますが、をするメソッドをびすことによって、をすぐにすることです。たとえば、 `ToList()` 、 `ToArray()`などです。バージョンのEntity Frameworkをしているは、 `ToListAsync()`や`ToArrayAsync()`などの`async`カウンタをできます。

にそのをします。

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

しかし、 `ToList()`や`ToArray()`びすことで、がにされることにすることがです。つまり、したのがりしをわなくてもメモリにロードされます。

ステートメントのは**null**セーフです

`IDisposable`オブジェクトで`null`をチェックするはありません。 `using`はをスローせず、 `Dispose()`はびされません。

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
```

```
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

GotchaDisposeメソッドのをしてのエラーをマスクする

のコードブロックをえてみましょう。

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

コンソールに「をできませんでした」とされることがされますが、には「にできませんでした」とされます。のがスローされたでもDisposeメソッドがびされるためです。

このなことは、オブジェクトがされるのをぐのエラーをマスクし、デバッグするのをよりにするがあるのでするがあります。

とデータベースの

`using`キーワードは、ステートメントでされたリソースがステートメントのスコープにのみすることをします。ステートメントでされたすべてのリソースは、 `IDisposable`インターフェイスをするがあります。

これらは、 `IDisposable`インターフェイスをしているをする、がにじられるだけでなく、 `using`ステートメントがになったにそのリソースがされることをできるため、にです。

の`IDisposable`データクラス

のうちくは、 `IDisposable`インターフェイスをし、 `using`ステートメントのなとなるデータクラス

です。

- `SqlConnection`、 `SqlCommand`、 `SqlDataReader`など
- `OleDbConnection`、 `OleDbCommand`、 `OleDbDataReader`など
- `MySqlConnection`、 `MySqlCommand`、 `MySqlDbDataReader`など
- `DbContext`

これらはすべて、Cをしてデータにアクセスするためににされ、データのアプリケーションをするににします。じ`FooConnection`、 `FooCommand`、 `FooDataReader`クラスをする、されていないのくのクラスは、じようにすることができます。

ADO.NETのアクセスパターン

ADO.NETをしてデータにアクセスするにできるなパターンは、のようになります。

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

         // Add your parameters here if necessary

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

または、なをしていてがなは、じがされます。

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

          // Add parameters here

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

DataContextでのステートメントの

Entity Frameworkなどのくの`DbContext`ようなクラスのでデータベースとやりとりするためにされるクラスをします。これらのコンテキストはに`IDisposable`インターフェイスもしており、なりステートメントを`using`これをするがあります。

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

Disposeをしてカスタムスコープをする

ユースケースによっては、 `using`をし`using`カスタムスコープをすることができます。たとえば、のカルチャでコードをするために、のクラスをすることができます。

```
public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
```

このクラスをして、のカルチャでされるコードブロックをできます。

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");

using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM
```

する`CultureContext`インスタンスはしないので、をしません。

このテクニックは、ASP.NET MVCの`BeginForm` ヘルパーによってされます。

コンテキストでのコードの

のコンテキストでしたいコード ルーチン があるは、をできます。

のは、いているSSLでするをしています。こののは、クライアントコードにしないライブラリまたはフレームワークにあります。

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
    }
}
```

SSLでかをしたいが、すべてのSSLのをしたくないクライアントコード。これで、SSLトンネルでなをうことができますたとえば、キーをするなど。

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

このルーチンは、のようにします。

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

これをするには、 `using()`がです`using()` `try..finally`ブロックをいてのであるため、クライアントコード `ExchangeSymmetricKey` がいてリソースをにすることなくすることをできます。 `using()` `using()`ないと、ルーチンがコンテキストのをってそれらのリソースをできないかどうかはしてかりません。

オンラインでステートメントのをむ https://riptutorial.com/ja/csharp/topic/38/ステートメントの

76: ストップウォッチ

- stopWatch.Start - ストップウォッチをします。
- stopWatch.Stop - ストップウォッチをします。
- stopWatch.Elapsed - のでされたをします。

ストップウォッチはベンチマークプログラムでよくタイムコードにされ、なるコードセグメントがどのくらいにされるかをします。

Examples

ストップウォッチのインスタンスの

ストップウオッチのインスタンスは、をいくつかのでできます。は、すべてのがにされます。これは、2つののをするできるをする。

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

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

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

`Stopwach`は`System.Diagnostics`ありますので、 `System.Diagnostics`を`using System.Diagnostics;`してするがあり`using System.Diagnostics;`あなたのファイルに。

IsHighResolution

- IsHighResolutionプロパティは、タイマーがのパフォーマンスカウンタにづいているのか、DateTimeクラスにづいているのかをします。
- このフィールドはみりです。

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

long frequency = Stopwatch.Frequency;
```

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

https://dotnetfiddle.net/ckrWUo

Stopwatchクラスでされるタイマーは、システムのハードウェアとオペレーティングシステムによってなります。ストップウォッチタイマーがのパフォーマンスカウンタにづいている、IsHighResolutionはです。それの、IsHighResolutionはfalseで、ストップウォッチタイマーはシステムタイマーにづいています。

StopwatchのTicksはマシン/ OSにしているので、2のシステムでStopwatchティックのをでしてしてしてしてしないでください。したがって、ストップウォッチダニをDateTime / TimeSpanティックとじにすることはしてありません。

システムにしないをするには、StopwatchのElapsedまたはElapsedMillisecondsプロパティーをしてください。これはにStopwatch.Frequencyのティックをしています。

ストップウォッチは、よりであるためタイミングプロセスにはにDatetimeをし、のパフォーマンスカウンタをできないはDateimeをします。

ソース

オンラインでストップウォッチをむ https://riptutorial.com/ja/csharp/topic/3676/ストップウォッチ

77: ストリーム

Examples

ストリームの

ストリームは、データをするためのレベルのをするオブジェクトです。それらはデータコンテナとしてはしません。

するデータは、バイト `byte []` です。みみときみのはすべてバイト`WriteByte()` `WriteByte()`。

、などをうはありません。これはストリームをににしますが、テキストをしたいだけのにはいがではありません。ストリームは、のデータをするににちます。

たちは、なるタイプのStreamをして、そこからきむ/みむのあるつまりバッキングストアをするがあります。たとえば、ソースがファイルの、 `FileStream`をするがあります。

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

に、バッキングストアがメモリのは、 `MemoryStream`がされます。

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes("C:\\file.txt");

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
   // do stuff here...
}
```

に、 `System.Net.Sockets.NetworkStream`はネットワークアクセスにされます。

すべてのストリームは、クラス`System.IO.Stream`からしています。ストリームからデータをみきすることはできません。 .NET Frameworkには、ネイティブタイプとレベルストリームインターフェイスをし、ストリームをするヘルパクラス `StreamReader` 、 `StreamWriter` 、 `BinaryReader` 、`BinaryWriter`などが`BinaryWriter`れています。

`StreamReader`と`StreamWriter`を`StreamReader` 、ストリームのみりときみをうことができます。これらをじるときにはがです。デフォルトでは、じられたストリームもじられ、それのにはできなくなります。このデフォルトのは、 `bool leaveOpen`パラメータをち、そのを`true`したコンストラクタを

してできtrue。

StreamWriter

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.Write(NextLine);
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

StreamReader

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();     This will close ms and when we try to use ms later it will cause an
exception
    sw.Flush();     //You can send the remaining data to stream. Closing will do this
automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Stream、 StreamReader、 StreamWriter、などをIDisposable々はびすことができ、インターフェースをDispose()これらのクラスのオブジェクトのメソッドを。

オンラインでストリームをむ https://riptutorial.com/ja/csharp/topic/3114/ストリーム

78: スレッディング

スレッドは、のとはしてできるプログラムのです。のスレッドとにタスクをできます。 マルチスレッドは、のをにうことができるように、プログラムがをできるようにするです。

たとえば、スレッドをしてタイマーやカウンタをバックグラウンドでするとに、フォアグラウンドでのタスクをすることができます。

マルチスレッド・アプリケーションは、ユーザーのにするがく、がのにじてスレッドをできるため、スケーラビリティもです。

デフォルトでは、Cプログラムにはスレッドメインプログラムスレッドが1つあります。ただし、プライマリスレッドとしてコードをするために、セカンダリスレッドをしてすることができます。このようなスレッドはワーカースレッドとばれます。

スレッドのをするために、CLRはスレッドスケジューラとばれるオペレーティングシステムにをします。スレッドスケジューラは、すべてのスレッドにながりてられていることをします。ブロックまたはロックされているスレッドがCPUのをしていないこともチェックします。

.NET Framework `System.Threading`はスレッドのをにします。 System.Threadingは、いくつかのクラスとインタフェースをすることによって、マルチスレッドをにします。のスレッドにとクラスをすることはとして、スレッドのコレクション、タイマークラスなどをするもします。また、データへのアクセスをすることで、そのサポートをします。

`Thread`は`System.Threading`のメインクラスです。そののクラスには、 `AutoResetEvent` 、 `Interlocked` 、 `Monitor` 、 `Mutex` 、および`ThreadPool`ます。

`System.Threading`にするのには、 `ThreadStart` 、 `TimerCallback` 、 `TimerCallback`などがあり `WaitCallback` 。

`System.Threading`のには、 `ThreadPriority` 、 `ThreadState` 、および`EventResetMode`まれ `System.Threading` 。

.NET Framework 4およびそれのバージョンでは、 `System.Threading.Tasks.Parallel`および `System.Threading.Tasks.Task`クラス、Parallel LINQPLINQ、 `System.Collections.Concurrent`しいコレクションクラスをして、マルチスレッドプログラミングがよりでになりました `System.Collections.Concurrent`、およびしいタスクベースのプログラミングモデル。

Examples

シンプルなスレッドのデモ

```
class Program
{
    static void Main(string[] args)
```

```
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

タスクをしたなスレッドのデモ

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
```

```
    }
}
```

なタスク**Parallism**

```
    private static void explicitTaskParallism()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
        Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

        // Start the task.
        taskA.Start();
        taskB.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);
        taskA.Wait();
        taskB.Wait();
        Console.Read();
    }
```

なタスクの

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
            () => b.DoSomeOtherWork()
            );

      }
```

2のスレッドのと

いをしているは、にそれらをコンピュータのなるスレッドでできます。これをうために、しいスレッドをし、のメソッドをしします。

```
using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
```

```
    }
}
```

パラメータでスレッドをする

using System.Threading;

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

プロセッサごとに1つのスレッドをする

`Environment.ProcessorCount`のマシンのプロセッサーのをします。

CLRはスレッドをプロセッサにスケジューリングします。これは、には、なるプロセッサのスレッド、のプロセッサのすべてのスレッド、またはのみわせをします。

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }

    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

にデータをみきするのをける

によっては、スレッドがにデータをしたいがあります。これがこるときには、コードをしてっているのあるをロックすることがです。 2つのスレッドカウントのなをにします。

はなったコードです

```
using System.Threading;

class MainClass
```

```
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

1,2,3,4,5をえるのではなく、1,1,2,2,3をえます。

このをするには、countのをロックするがあります。これにより、のなるスレッドがにみきをうことができなくなります。ロックとキーをすることで、スレッドがにデータにアクセスするのをぐことができます。

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
```

```
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value
of count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}
```

Parallel.ForEachループ

スピードアップしたいforeachループがあり、のがにならないは、のようにしてforeachループにできます。

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

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

デッドロック2つのスレッドがおいをっている

デッドロックは、2つのスレッドがおいがするのをっているとき、またはにつようなでリソースをするときにします。

おいをつ2つのスレッドのなシナリオは、WindowsフォームGUIスレッドがワーカースレッドをち、ワーカースレッドがGUIスレッドによってされているオブジェクトをびそうとするときです。このコードでは、button1をクリックするとプログラムがハングアップすることにしてください。

```
private void button1_Click(object sender, EventArgs e)
```

```
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

`workerthread.Join()`は、workerthreadがするまでびしスレッドをブロックするびしです。`textBox1.Invoke(invoke_delegate)`は、GUIスレッドがinvoke_delegateをするまでびしスレッドをブロックするびしですが、GUIスレッドがびしスレッドがするのをっているは、このびしによってデッドロックがします。

これをするには、わりにテキストボックスをびすノンブロッキングのをできます

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

ただし、にされるテキストボックスにするコードをするがあるは、このがします。その、びしのとしてしますが、GUIスレッドでされることにしてください。

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}
```

または、しいスレッドをして、そのスレッドがGUIスレッドでするようにして、workerthreadがするようにします。

```
private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
```

```
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}
```

のデッドロックにるリスクをにえるため、なはにスレッドのをけてください。のスレッドがのスレッドにしてのみメッセ―ジをし、それをつことはないスレッドのは、こののにはがりません。しかし、リソ―スロックにづいたデッドロックにしてはまだです。

デッドロックリソ―スのと

デッドロックは、2つのスレッドがおいがするのをっているとき、またはにつようなでリソ―スをするときにします。

thread1がリソ―スAのロックをしていて、スレッドBがリソ―スBをしているときにリソ―スBがされるのをっていて、リソ―スAがされるのをっている、それらはデッドロックされます。

のコ―ドでbutton1をクリックすると、アプリケ―ションはのデッドロックになり、ハングします

```
private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
```

```
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

このようにデッドロックされないようにするには、Monitor.TryEnterlock_object、timeout_in_millisecondsをして、オブジェクトにロックがすでにされているかどうかをできます。 Monitor.TryEnterがtimeout_in_millisecondsのにlock_objectのロックをできなかったはfalseをし、スレッドにのされているリソ―スをしてさせるをえ、このようにししたバ―ジョンのように

```
private void button_Click(object sender, EventArgs e)
{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        bool mustDoWork = true;
```

```
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

このは、スレッド2がそのロックにしてであり、スレッド1がしようとしているので、thread2がにされることにしてください。また、スレッド1は、リソースAをロックしたのをやりすがあります。したがって、2つのするスレッドでこのアプローチをするときにはしてください.2つのスレッドがののをけておいにをすにこる、いわゆるライブロックをするリスクがあります、りしめます。

オンラインでスレッディングをむ https://riptutorial.com/ja/csharp/topic/51/スレッディング

79: スレッドセーフなをる

Examples

Parallel.Forループのへのアクセスの

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

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500.  sum is: {0}", sum );
    }
}
```

read-modify-writeがアトミックではないため、ロックなしで`sum = sum + i`をするだけではです。スレッドへのきされ`sum`がのみったに`sum`が、それはの`sum + i`バックに`sum`。

オンラインでスレッドセーフなをるをむ https://riptutorial.com/ja/csharp/topic/4140/スレッドセーフなをる

80: ダイナミックタイプ

dynamicキーワードは、コンパイルにがからないをします。 dynamicにはのをめることができ、のはにできます。

「Metaprogramming in .NET」のでしたように、Cにはdynamicキーワードのバッキングタイプはありません。

> dynamicキーワードによってになるは、ローカルスコープのサイトコンテナでCallSiteオブジェクトをしてするコンパイラアクションのなセットです。コンパイラは、プログラマがCallSiteインスタンスをじてオブジェクトとしてするをします。コンパイルになをうパラメータ、りの、フィールド、およびプロパティには、のためにされたことをすメタデータがいていますが、そのになるデータはにSystem.Objectます。

Examples

の

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

にる

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
}
```

プロパティをしたオブジェクトの

```
using System;
```

```
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

コンパイルになのの

のとのがられます。

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}

class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }
```

```
    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

になのは、しいタイプのハンドルをするだけで、しいタイプのDebugToStringInternalのオーバーロードをするがあることです。また、でタイプにキャストするもありません。

オンラインでダイナミックタイプをむ https://riptutorial.com/ja/csharp/topic/762/ダイナミックタイプ

81: タイプ

はあるのデ―タをのにすることです。タイプキャスティングともばれます。 Cでは、キャストにはの2つのがあります。

の - これらのは、Cでにされます。たとえば、よりさいのやクラスからクラスへのなどです。

な - これらのは、されたをするユ―ザ―によってにわれます。なにはキャストがです。

Examples

MSDNのの

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implict conversion called");
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implict conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

のとばれる
ディジットののにダブル
num = 7 dig2 = 12

.NET Fiddleのライブデモ

な

```
using System;
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

オンラインでタイプをむ https://riptutorial.com/ja/csharp/topic/3489/タイプ

82: タイマー

- `myTimer.Interval` - 「Tick」イベントがびされるをミリでします。
- `myTimer.Enabled` - タイマを/にするブール
- `myTimer.Start()` - タイマーをします。
- `myTimer.Stop()` - タイマーをします。

Visual Studioをしているは、ツールボックスからフォームにタイマーをコントロールとしてできます。

Examples

マルチスレッドタイマ

`System.Threading.Timer` - もなマルチスレッドタイマー。 2つのメソッドと1つのコンストラクタがまれます。

タイマーはDataWriteメソッドをびします。このメソッドは、5に「マルチスレッドがされました...」ときまれ、その、ユーザーがEnterキーをすまで1ごとに1ごとにきみます。

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
    Console.ReadLine();
    timer.Dispose(); // This both stops the timer and cleans up.
  }

  static void DataWrite (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data); // Writes "multithread executed..."
  }
}
```

マルチスレッドタイマーをするためののセクションをします。

`Change` - このメソッドは、タイマーのをするときにびすことができます。

`Timeout.Infinite` - 1だけしたい。これをコンストラクタののでします。

`System.Timers` - .NET Frameworkによってされるのタイマークラス。これは、`System.Threading.Timer`ラップし`System.Threading.Timer`。

- `IComponent`

- Visual Studioのデザイナーのコンポーネントトレイにできるようにする
- `Change`メソッドではなく`Interval`プロパティ
- コールバック`delegate`わりに`Elapsed event`
- タイマーをおよびする`Enabled`プロパティー `default value = false`
- `Enabled`プロパティのポイントでしたにえて、 `Start Stop`メソッド
- `AutoReset` - なイベントをすため `default value = true`
- WPFとWindowsフォームコントロールのメソッドをにびすためのInvokeメソッドとBeginInvokeメソッドとのSynchronizingObjectプロパティ

のすべてのをす

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
  static void Main()
  {
    Timer timer = new Timer(); // Doesn't require any args
    timer.Interval = 500;
    timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
    Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
    timer.Start(); // Restart the timer
    Console.ReadLine();
    timer.Dispose(); // Permanently stop the timer
 }

 static void timer_Elapsed(object sender, EventArgs e)
 {
   Console.WriteLine ("Tick");
 }
}
```

`Multithreaded timers` - スレッドプールをして、いくつかのスレッドがのタイマーをできるようにします。コールバックメソッドまたは`Elapsed`イベントは、びされるたびにのスレッドでトリガされるがあります。

`Elapsed` - このイベントは、の`Elapsed`イベントがをしたかどうかにかかわらず、の`Elapsed`とともににします。このため、コールバックまたはイベントハンドラはスレッドセーフでなければなりません。マルチスレッドタイマーのは、OSによってなりますが、は1020 msです。

`interop` - よりながなは、これをしてWindowsマルチメディアタイマーをびします。これは1msまでのをち、 `winmm.dll`でされてい`winmm.dll`。

`timeBeginPeriod` - これをにびし、いタイミングがであることをOSにする

`timeSetEvent` -にこれをびす`timeBeginPeriod`マルチメディアタイマーをします。

`timeKillEvent` - `timeKillEvent`したらこれをびすと、タイマーがします

`timeEndPeriod` - これをびすと、いタイミングがになったことをOSにします。

`dllimport winmm.dll timesetevent`というキーワードをすることで、マルチメディアタイマーをするなをインターネットでつけることができます。

タイマーのインスタンスの

タイマーは、のでタスクをするためにされますYごとにDo X。は、タイマーのしいインスタンスをするです。

これは、WinFormsをするタイマーにされます。 WPFをしているは、 `DispatcherTimer`

```
    using System.Windows.Forms; //Timers use the Windows.Forms namespace

    public partial class Form1 : Form
    {

        Timer myTimer = new Timer(); //create an instance of Timer named myTimer


        public Form1()
        {
            InitializeComponent();
        }

    }
```

Timerに "Tick" イベントハンドラをりてる

タイマーでされるすべてのアクションは、「Tick」イベントでされます。

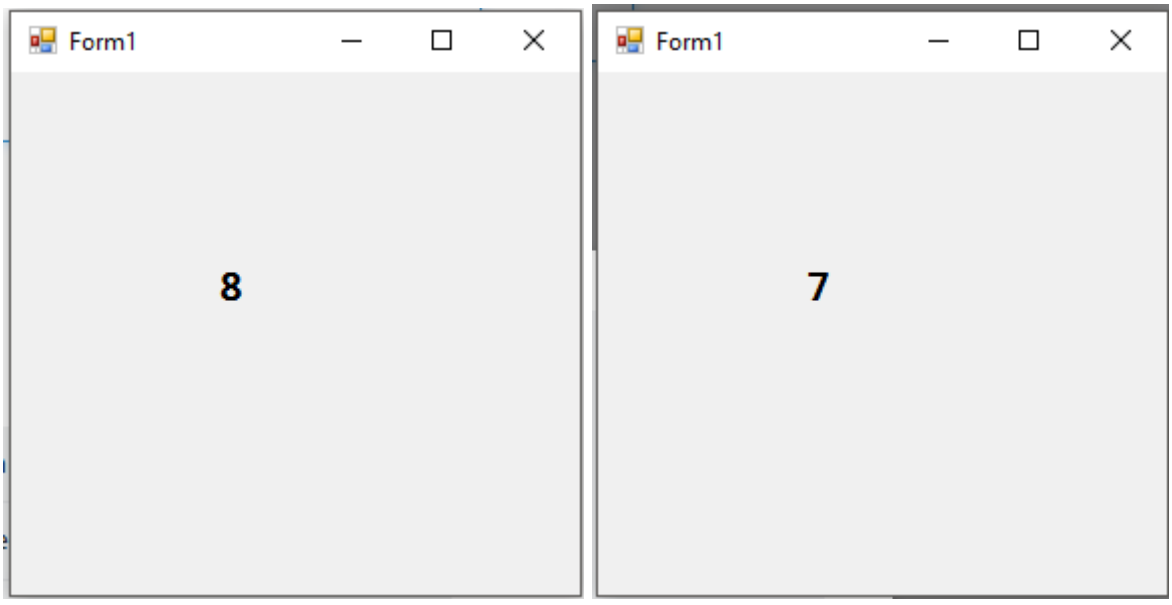
```
public partial class Form1 : Form
{

    Timer myTimer = new Timer();


    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

タイマーをしてなカウントダウンをする。

```
    public partial class Form1 : Form
```

```
{

Timer myTimer = new Timer();
int timeLeft = 10;

    public Form1()
    {
        InitializeComponent();

        //set properties for the Timer
        myTimer.Interval = 1000;
        myTimer.Enabled = true;

        //Set the event handler for the timer, named "myTimer_Tick"
        myTimer.Tick += myTimer_Tick;

        //Start the timer as soon as the form is loaded
        myTimer.Start();

        //Show the time set in the "timeLeft" variable
        lblCountDown.Text = timeLeft.ToString();

    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        //perform these actions at the interval set in the properties.
        lblCountDown.Text = timeLeft.ToString();
        timeLeft -= 1;

        if (timeLeft < 0)
        {
            myTimer.Stop();
        }
    }
}
```

は...



々...

オンラインでタイマーをむ https://riptutorial.com/ja/csharp/topic/3829/タイマー

83: タスクライブラリ

Examples

Parallel.ForEach

Parallel.ForEachループをして、されたWebサイトURLのにpingをする。

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

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

        var result = ping.Send(url);

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

Parallel.For

Parallel.Forループをして、されたWebサイトURLのにpingをする。

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

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

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

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

```
        }
    });
}
```

Parallel.Invoke

にメソッドまたはアクションをびすパラレル

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

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

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

    var result = ping.Send(url);

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

のにするりしポーリングタスク

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

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

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

    public void CancelExecution()
    {
```

```
        this._cts.Cancel();
    }

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

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

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

CancellationTokenSourceをしてキャンセルなポーリングタスク

```
public class Foo
{
    private CancellationTokenSource _cts;

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

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

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

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

        while ( !token.IsCancellationRequested )
        {
```

```
            Console.WriteLine("Do your task work in this loop");
        }
    }
}
```

PingUrlのバージョン

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

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

        var result = await ping.SendPingAsync(url);

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

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

        return response;
    }
```

オンラインでタスクライブラリをむ https://riptutorial.com/ja/csharp/topic/1010/タスクライブラリ

84: タスクライブラリTPLのデータフロー

Examples

JoinBlock

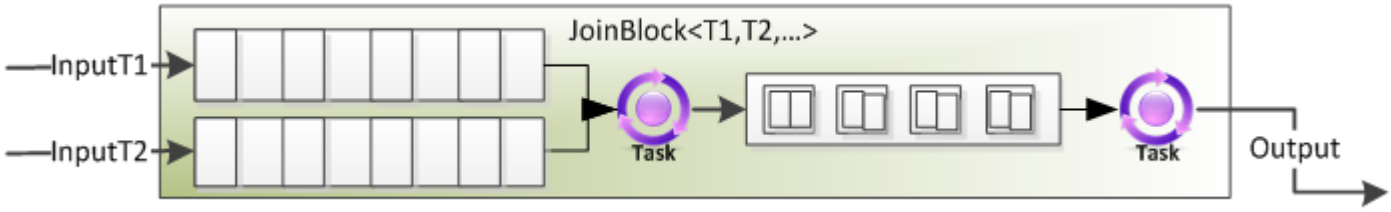
23のをめてタプルにする

BatchBlockとに、JoinBlock <T1、T2、...>はのデータソースからデータをグループできます。、それはJoinBlock <T1、T2、...>のなです。

たとえば、JoinBlock <string、double、int>はISourceBlock <Tuple <string、double、int >>です。

BatchBlockのとに、JoinBlock <T1、T2、...>はモードとモードのですることができます。

- デフォルトのりモードでは、タプルをするのになデータがのターゲットにないでも、ターゲットにされるすべてのデータがけれられます。
- でないモードでは、ブロックのターゲットは、すべてのターゲットになデータがされてタプルがされるまでデータをします。このので、ブロックは2フェーズコミットプロトコルをしてソースからなすべてのアイテムをアトミックにします。このにより、のエンティティがそのにデータをし、システムがすることがになります。



プールされたオブジェクトのがられているリクエストの

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

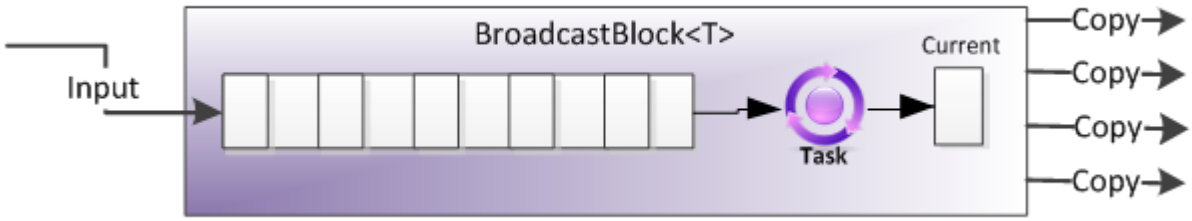
Stephen ToubTPLデータフロー

BroadcastBlock

アイテムをコピーし、それがリンクされているすべてのブロックにコピーをする

BufferBlockとはって、BroadcastBlockのは、ブロックからリンクされたすべてのターゲットがされたすべてののコピーをし、にのをきすることです。

さらに、BufferBlockとはなり、BroadcastBlockはデータをにしません。のデータムがすべてのターゲットにされた、そのはのにあるデータによってきされますすべてのデータフローブロックとに、メッセージはFIFOにされます。そのはすべてのターゲットにされます。



プロデューサをしたプロデューサ/コンシューマ

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

エージェントからのステータスの

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }

    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
```

```
        Status.Post("Doing cool stuff");
        …
        Status.Post("Done");
    }
}
```

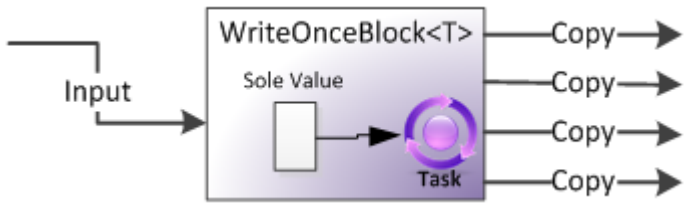
Stephen ToubTPLデータフロー

WriteOnceBlock

Readonlyのデータをし、そのコピーをとしてします。のすべてのデータはします

BufferBlockがTPL Dataflowのもなブロックである、WriteOnceBlockはもです。
くても1つのがされ、そのがされると、またはきされることはありません。

WriteOnceBlockは、コンストラクタでのみで、であるのではなく、でのをいて、Cのreadonlyメンバとているとえることができます。



タスクのの

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try
    {
        result.Post(await task);
    }
    catch(Exception ex)
    {
        exception.Post(ex);
    }
}
```

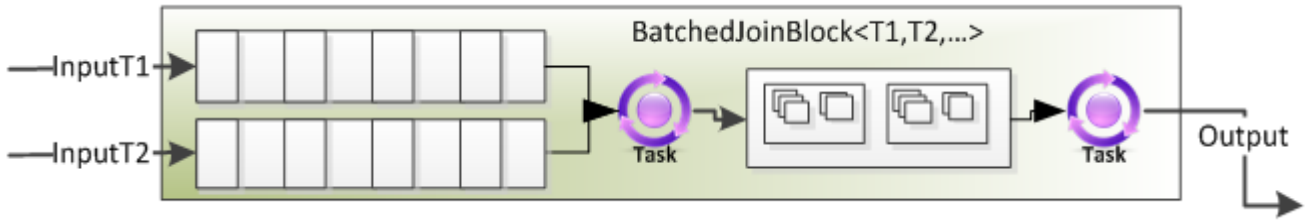
Stephen ToubTPLデータフロー

BatchedJoinBlock

23のからののアイテムをし、それらをデータアイテムののタプルにグループする

BatchedJoinBlock <T1、T2、...>は、あるではBatchBlockとJoinBlock <T1、T2、...>のみわせです。

JoinBlock <T1、T2、...>はターゲットからの1つのをタプルにするためにされ、BatchBlockはNのをコレクションにするためにされ、BatchedJoinBlock <T1、T2、...>すべてのターゲットをコレクションのタプルにみみます。



スキャッタ/ギャザー

Nのがされ、そのうちのいくつかがしてをし、そのがしてをするのあるスキャッタ/ギャザーのをえてみましょう。

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}
```

Stephen ToubTPLデータフロー

TransformBlock

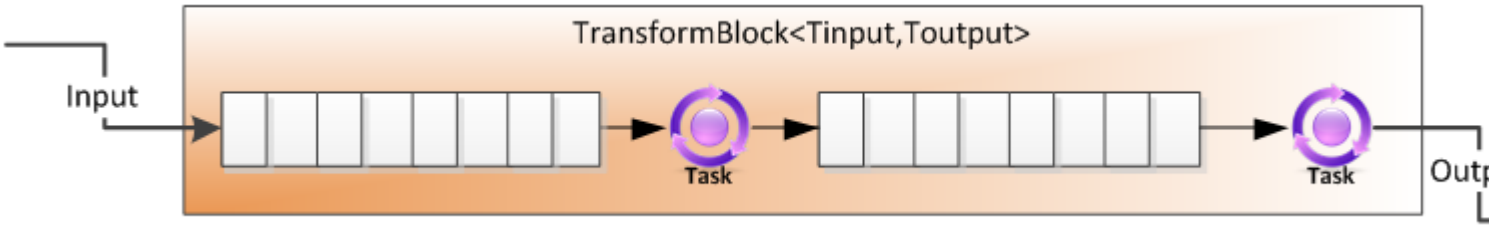
、11

ActionBlockとに、TransformBlock <TInput、TOutput>はデリゲートのがデータにしてらかのアクションをできるようにします。 **ActionBlock**とはなり、このにはがあります。このデリゲートは、Func <TInput、TOutput>とすることができます。この、デリゲートがされたときにそののがしたとなされるか、Func <TInput、Task>になります。デリゲートがされたときに、されたタスクがしたとき。 LINQにしているにとって、Selectはをけり、らかのでそのをし、をするでSelectとています。

デフォルトでは、TransformBlock <TInput、TOutput>は、MaxDegreeOfParallelismが1にしいでデ

—タをします。バッファされたをけってすることにえて、このブロックはされたとバッファをすべてりますされたデ—タをします。

これには2つのタスクがあります1つはデ—タをするタスク、もう1つはデ—タをのブロックにプッシュするタスクです。



パイプライン

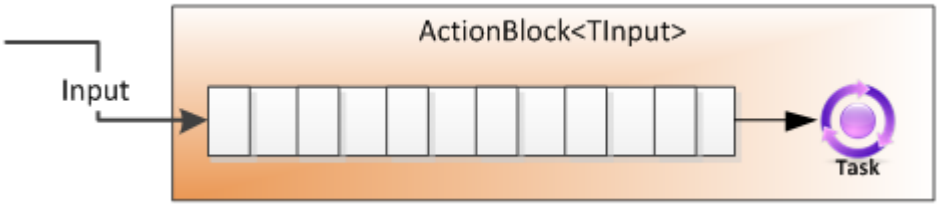
```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

Stephen ToubTPLデ—タフロ—

ActionBlock

foreach

このクラスは、には、デ—タをするタスクとをする「デ—タフロ—ブロック」をみわせて、するデ—タのバッファとしてえることができます。もないでは、ActionBlockをインスタンスしてActionBlockに「」することができます。 ActionBlockのでされたデリゲ—トは、ポストされたすべてのデ—タにしてにされます。




```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

ダウンロ—ドをで5つまでにする

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

Stephen ToubTPLデータフロー

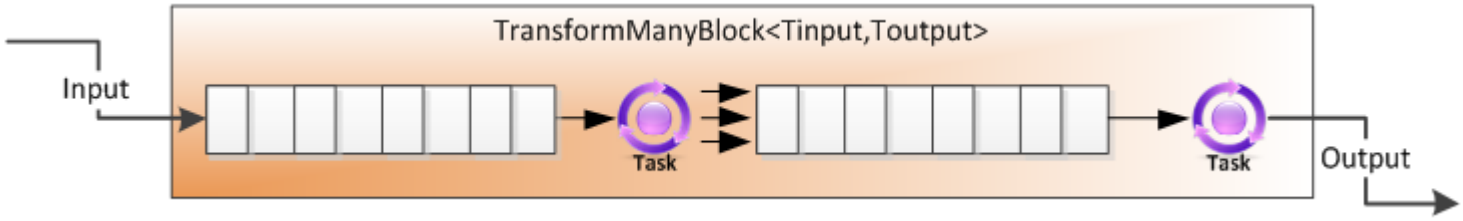
TransformManyBlock

SelectMany、1-mこのマッピングのは、LINQのSelectManyとに「」されています

TransformManyBlock <TInput、TOutput>は、TransformBlock <TInput、TOutput>とによくています。
ないは、TransformBlock <TInput、TOutput>はにして1つのしかしませんが、
TransformManyBlock <TInput、TOutput>はごとにの0のをするです。 ActionBlockおよび
TransformBlock <TInput、TOutput>とに、このは、とののをしてできます。

Func <TInput、IEnumerable>はにされ、Func <TInput、Task <IEnumerable >>はにされます。
ActionBlockとTransformBlock <TInput、TOutput>のとに、TransformManyBlock <TInput、
TOutput>はデフォルトでになりますが、それのはされます。

マッピングデリゲートは、バッファににされるアイテムのコレクションをします。



Webクローラー

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Enumerableをそのにする

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

1から**0**または**1**からなるフィルタリング

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Stephen ToubTPLデータフロー

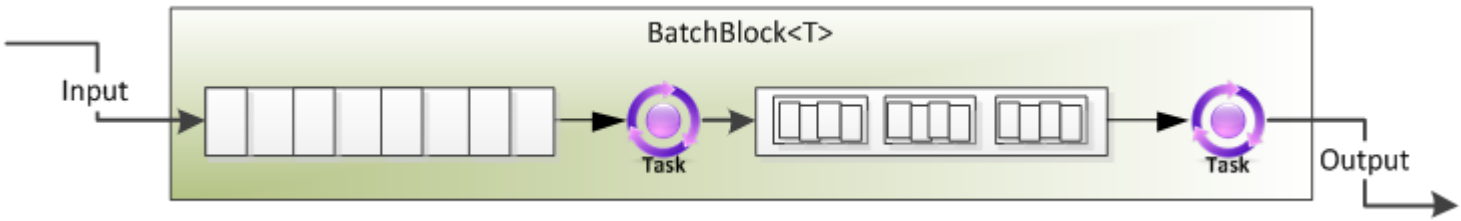
バッチブロック

のデータをデータのにグループする

BatchBlockは、Nのを1つのバッチにし、のとしてします。のバッチサイズでインスタンスがされ、ブロックはそののをけるとすぐにバッチをし、バッチをバッファににします。

BatchBlockは、モードとモードのですることができます。

- デフォルトのモードでは、ののソースからブロックにされたすべてのメッセージがけれられ、バッファにれられてバッチにされます。
- - モードでは、バッチをするのになソースがブロックにメッセージをするまで、すべてのメッセージはソースからされます。したがって、BatchBlockをして、Nのソースのそれぞれから1の、1のソースからNの、およびそれらのにのオプションをけることができます。



リクエストを**100**のグループにまとめてデータベースにする

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

1に**1**バッチをする

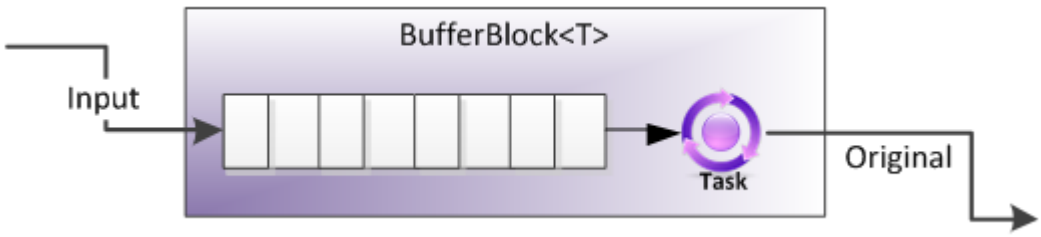
```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Stephen ToubTPLデータフロー

BufferBlock

FIFOキューってるデータは、てくデータです

するに、BufferBlockは、Tのインスタンスをするためのまたはバッファをします。
Tのインスタンスをブロックに「ポスト」することができます。これにより、ポストされたデータがブロックによってれしFIFOにされます。
あなたはブロックから ""することができます。これにより、にされた、またはなTのインスタンスつまり、FIFOをまたはにできます。



プロデューサをしたプロデューサ/コンシューマ

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

Stephen ToubTPLデータフロー

オンラインでタスクライブラリTPLのデータフローをむ
https://riptutorial.com/ja/csharp/topic/3110/タスクライブラリ-tpl-のデータフロー

85: タプル

Examples

タプルの

タプルは、タイプ`Tuple<T1>` - `Tuple<T1,T2,T3,T4,T5,T6,T7,T8>`をしてされます。それぞれのは、18のをむタプルをします。はなるタイプのものにすることができます。

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

タプルは、な`Tuple.Create`メソッドをしてすることもできます。この、のはCコンパイラによってされます。

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7.0

C7.0、タプルはValueTupleをしてにできます。

```
var tuple = ("foo", 123, true, new MyClass());
```

なのためににをけることができます。

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

タプルへのアクセス

タプルにアクセスするには、 `Item1` - `Item8`プロパティをします。タプルサイズのインデックスをつプロパティのみがになりますつまり、 `Tuple<T1,T2>` `Item3`プロパティにアクセスできません。

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new My Class()
```

タプルのとソート

タプルはそのについてできます。

として、 `Tuple`のをつは、されたでされたについてソートできます。

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

ソートのいをにするには

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

メソッドからのをす

タプルは、パラメータをせずにメソッドからのをすためにできます。のでは、 `AddMultiply`をして2つのsum、productがされます。

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

53
700

C7.0では、タプルをするメソッドからのをすわりのがされています`ValueTuple`にするさらにしい。

オンラインでタプルをむ https://riptutorial.com/ja/csharp/topic/838/タプル

86: チェックされているとチェックされていない

- チェックa + b//チェックされた
- チェックされていないa + b//チェックされていない
- チェック{c = a + b; c + = 5; } //チェックされたブロック
- チェックされていない{c = a + b; c + = 5; } //チェックのブロック

Examples

チェックされているとチェックされていない

Cステートメントは、チェックされたコンテキストまたはチェックされていないコンテキストでされます。チェックされたコンテキストでは、オーバーフローによってがします。チェックされていないコンテキストでは、オーバーフローはされ、はりてられます。

```
short m = 32767;
short n = 32767;
int result1 =  checked((short)(m + n));   //will throw an OverflowException
int result2 =  unchecked((short)(m + n)); // will return -2
```

どちらもされていない、デフォルトコンテキストはコンパイラオプションなどののにします。

スコープとしてチェックされ、チェックされていない

キーワードは、のをするためにスコープをすることもできます。

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

オンラインでチェックされているとチェックされていないをむ
https://riptutorial.com/ja/csharp/topic/2394/チェックされているとチェックされていない

87: データベースへのアクセス

Examples

ADO.NET

ADO.NETは、Cアプリケーションからデータベースにするもなの1つです。らは、クエリをするためにデータベースをすプロバイダとのにしています。

データプロバイダクラス

のうちくは、データベースとそのするのクエリによくされるクラスです。

- `System.Data.SqlClient` `SqlConnection`、 `SqlCommand`、 `SqlDataReader`
- `OleDbConnection`、 `OleDbCommand`、 `System.Data.OleDb` `OleDbDataReader`
- `MySqlConnection`、 `MySqlCommand`、 `MySqlDbDataReader` from `MySql.Data`

これらはすべて、Cをしてデータにアクセスするためににされ、データのアプリケーションをするににします。じ`FooConnection`、 `FooCommand`、 `FooDataReader`クラスをする、されていないのくのクラスは、じようにすることができます。

ADO.NETのアクセスパターン

ADO.NETをしてデータにアクセスするにできるなパターンは、のようになります。

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

         // Add your parameters here if necessary

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

または、なをしていてがなは、じがされます。

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

          // Add parameters here

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

インタフェースのセットにしてプログラムをすることもでき、プロバイダのクラスについてするはありません。 ADO.NETによってされるコアインターフェイスはのとおりです。

- IDbConnection - データベースをする
- IDbCommand - SQLコマンドの
- IDbTransaction - トランザクション
- IDataReader - コマンドによってされたデータをみむ
- IDataAdapter - データセットとのでデータをチャネリングする

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

var factory = DbProviderFactories.GetFactory(providerName);

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

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

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

エンティティフレームワークの

Entity Frameworkは、 `DbContext`ようなクラスのでとなるデータベースとするためにされるクラスをします。これらのコンテキストは、に、できるなコレクションをする`DbSet<T>`プロパティでされています。

```
public class ExampleContext: DbContext
{
```

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

DbContextはデータベースとのをし、、からののをするために、からなデータをみります。

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

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

エンティティフレームワークのクエリ

にEntity Frameworkのクエリをするのはにで、にコンテキストのインスタンスをし、そのコンテキストでなプロパティをしてデータをまたはアクセスするがあります

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

また、Entity Frameworkは、 SaveChanges()メソッドをびしてをデータベースにプッシュするだけで、データベースのエントリのをするためにできるチェンジトラッキングシステムをします。

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

は、のデータソースにする、および、などのをしてするをするです。

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

の

、はファイルASP.NETアプリケーションのapp.configやweb.configなどにされます。は、これらの

ファイルのいずれかでロ―カルがどのようにえるかのです。

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

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

これにより、アプリケ―ションはプログラムで`WidgetsContext`をじてにアクセスできます。`Integrated Security=SSPI`と`Integrated Security=True`がじをしますが、 `Integrated Security=SSPI`は、SQLClientとOleDBプロバイダ―ので、 `Integrated Security=true`としてするため、OleDbプロバイダ―でするとがスロ―されるため、されます。

なるプロバイダのなる

データプロバイダSQL Server、MySQL、Azureなどはすべて、ののをえ、さまざまななプロパティをします。 ConnectionStrings.comはあなたのものがどのようにえるべきかなはになリソ―スです。

オンラインでデ―タベ―スへのアクセスをむ https://riptutorial.com/ja/csharp/topic/4811/データベースへのアクセス

88: データ

Examples

DisplayNameAttribute

`DisplayName`は、が0のプロパティ、イベント、またはpublic voidメソッドのをします。

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

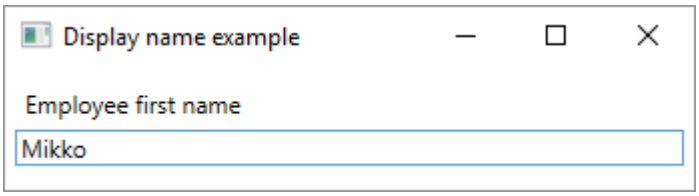
XAMLアプリケーションのな

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

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

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

</Window>
```

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

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

        public Employee Employee
```

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

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

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

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



EditableAttributeデータモデリング

EditableAttributeは、ユーザーがクラスプロパティのをできるがあるかどうかをします。

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

XAMLアプリケーションのな

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

```
        Height="70" Width="360" Title="Display name example">

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

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

</Window>
```

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

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

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

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

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

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

バリデーションは、クラスまたはクラスメンバーのなで、さまざまなバリデーションルールをするためにされます。すべてのは、 ValidationAttributeクラスからしています。

RequiredAttribute

ValidationAttribute.ValidateメソッドによってValidationAttribute.Validateれると、 Nameプロパティがnullまたはのみをむ、このはエラーをします。

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

StringLengthAttribute

StringLengthAttributeは、がのよりもさいかどうかをします。オプションでをすることもできます。のがです。

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

RangeAttribute

RangeAttributeは、フィールドのとをRangeAttributeます。

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

```
}
```

CustomValidationAttribute

`CustomValidationAttribute`クラスをすると、のためにカスタム`static`メソッドをびすことができます。カスタムメソッドは`static ValidationResult [MethodName] (object input)`なければなりません。

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

メソッド

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

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

        return result;
    }
}
```

カスタムの

カスタムは、 `ValidationAttribute`クラスからし、にじて`virtual`メソッドをオ―バ―ライドすることでできます。

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

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

        return isValid;
    }
}
```

このは、のようにできます。

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

データアノテーションの

データは、クラスまたはクラスのメンバーにコンテキストをするです。のなカテゴリは3つあります。

- データにをする
- ユーザーにデータをするをする
- モデルのクラスとのやにするをする

2つの`ValidationAttribute`と1つの`DisplayAttribute`がされているをにします。

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

データは、にASP.NETなどのフレームワークでされます。たとえば、 `ASP.NET MVC`では、コントローラメソッドによってモデルがされると、 `ModelState.IsValid()`をして、したモデルがその`ValidationAttribute`すべてするかどうかをることができます。 `DisplayAttribute`は、 `ASP.NET MVC` Webページにをするをするためにもされます。

をでする

ほとんどの、はフレームワークASP.NETなどでされます。これらのフレームワークは、のをします。しかし、をでするはどうすればよいでしょうか `Validator`クラスをしますリフレクションはありません。

コンテキスト

バリデーションには、がされているかにするをえるためにコンテキストがです。これには、するオブジェクト、いくつかのプロパティ、エラーメッセージにするなど、さまざまなをめることができます。

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

コンテキストがされると、をうはあります。

オブジェクトとそのすべてのプロパティをする

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

オブジェクトのプロパティをする

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

もっと

のについては、をしてください。

- ValidationContextクラスのドキュメント
- バリデータクラスのドキュメント

オンラインでデータをむ https://riptutorial.com/ja/csharp/topic/4942/データ

89: デコレータデザインパターンの

デコレータをする

- になるでしいをすることができます
- のい
- クライアントはしたいをできます

Examples

カフェテリアのシミュレーション

デコレータはパターンの1つです。これは、オブジェクトのを、、またはするためにされます。このドキュメントでは、Decorator DPをにするについてします。

あなたになでそれのアイデアをしましょう。あなたがなコーヒーのスターバックスにいるとしてください。クリームと、クリームとトッピング、さらにくのみわせで、あなたがむコーヒーをすることができますしかし、すべてのみのベースはコーヒーダーク、いみ、あなたはすることができます。コーヒーマシンをシミュレートするシンプルなプログラムをってみましょう。

まず、ベースドリンクをするクラスをし、クラスをするがあります。

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

さあ、、ミルク、トッピングのようなエキストラをってみましょう。されたクラスは、`AbstractCoffee`するがあります - らはそれをります

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }
    public override string ShowCoffee()
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}
public class Sugar : AbstractCoffee
{
```

```
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
        else return "Topping";
    }
}
```

たちはおにりのコーヒーをることができます

```
public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee);  //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());

    }
}
```

コードをすると、のがされます。

ミルクりシュガーりトッピングコーヒー

オンラインでデコレータデザインパターンのをむ https://riptutorial.com/ja/csharp/topic/4798/デコレータデザインパターンの

90: ヌルき

- XY; // XがnullのはNull else XY
- XYZ; // XがnullまたはYがnullのはNull else XYZ
- X[index]; // XがnullのはNull else X [index]
- X.ValueMethod; // XがnullのはNull else X.ValueMethodの;
- X.VoidMethod; // XがnullのはもしませんElse call X.VoidMethod;

タイプTヌルをするは、 `Nullable<T>`すことにしてください。

Examples

Nullき

`?.`はなヌルチェックをけるためにのです。 セーフナビゲーションオペレータとしてもられています。

のでされるクラス

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

オブジェクトがnullのがあるをすなど、 `NullReferenceException`がしないようにオブジェクトをまずチェックするがあります。 nullきがなければ、のようになります。

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

null-conditionalをしたじ

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

オペレータの

null-conditionalは、オブジェクトのメンバーとサブメンバーでできます。

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

NULLとの

null-conditionalは、 NULLとみわせて、デフォルトをすることができます。

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

ヌル

?.に?. NULLきは、NULLになるのあるコレクションにけするときに、NULLをします。

```
string item = collection?[index];
```

のの

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

NullReferenceExceptionsをける

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

このはすることができます

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

ヌルはメソッドでできます

メソッドはnullでもしますが、 `?.`をでき`?.`とにかくヌルチェックする。

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

、このメソッドは`null`にしてトリガされ、-1をし`null`。

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

`?.`をしてい`?.`このメソッドは`null`にしてトリガされず、 は`int?`

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

このは、には`?.`がする `NullReferenceExceptions`をけるために、nullインスタンスのインスタンスメソッドびしをける。ただし、メソッドがされるのいにもかかわらず、じロジックがメソッドにされます。

のでメソッドがびされるのについては、 メソッド - nullチェックのドキュメントをしてください。

オンラインでヌルきをむ https://riptutorial.com/ja/csharp/topic/41/ヌルき

91: ヌル

- var result = possibleNullObject ?? defaultValue;

パラメーター

パラメータ	
possibleNullObject	ヌルをテストする。 nullでないは、このがされます。 nullなでなければなりません。
defaultValue	possibleNullObjectがnullのにされる。 possibleNullObjectとじでなければなりません。

nullは、2つのしたのです ??

これは、のです。

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

のオペランドテストのオブジェクトは、nullまたはでなければなりません。そうしないと、コンパイルエラーがします。

はとのでします。

Examples

な

null-coalescing operator (??)すると、のオペランドがnullにヌルのデフォルトをできます。

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

.NET Fiddleのライブデモ

これはにはのものとです。

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
```

```
    Console.WriteLine("The specified string is - " + testString);
}
```

または:)をします 。

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

ヌルフォールスルーとチェーニング

のオペランドはNULLでなければならず、のオペランドはNULLでなければならない。それにじてがタイプされます。

null

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

System.Int32
3

デモをる

Nullable

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output`は`int?`になります`int? b`としいか、または`null`です。

の

はでうこともできます

```
int? a = null;
int? b = null;
int c = 3;
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

System.Int32

3

デモをる

Nullきチェイン

ヌルオペレーターは、 ヌルとしてして、オブジェクトのプロパティへのよりなアクセスをできます。

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

System.String
デフォルト

デモをる

メソッドびしによるヌルの

nullは、 `null`をすメソッドがデフォルトにることをにするのをにします。

ヌルをしない

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

ヌルの

```
string name = GetName() ?? "Unknown!";
```

のものをするかする

このがにつなシナリオは、コレクションのオブジェクトをしていて、しないはしいオブジェクトをするがあるです。

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

ヌルをしたプロパティの

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

にプロパティ.FooBarsにアクセスすると、 _fooBarsはnullとしてされ、ステートメントにってのをしてします。

スレッドの

これは、なプロパティをするスレッドセーフなではありません 。スレッドセーフなには、.NET Frameworkにみまれたクラスをします。

のをったC6の

C6、このはプロパティのをしてできます。

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

そののプロパティへのアクセスは、 _fooBarsにされたをします。

MVVMパターンの

これは、MVVMパターンでコマンドをするときによくされます。ビューモデルのににコマンドをするのではなく、のようにこのパターンをしてコマンドをします。

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

オンラインでヌルをむ https://riptutorial.com/ja/csharp/topic/37/ヌル

92: ネットワーキング

- TcpClientホスト、intポート;

`client.GetStream()`から`client.GetStream()`を`TcpClient`して`NetworkStream`をし、`StreamReader/StreamWriter`に`StreamReader/StreamWriter`て、のみきメソッドにアクセスすることができます。

Examples

なTCPクライアント

このコードは、TCPクライアントをし、ソケットをして "Hello World"をし、をじるにサーバーをコンソールにきみます。

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivilent but safer
than calling Close();
```

Webサーバーからファイルをダウンロードする

インターネットからファイルをダウンロードすることは、あなたがしようとするほとんどのアプリケーションでとされるになです。

これをするには、 " System.Net.WebClient "クラスをします。

これを「する」パターンをしてもにするをにします。

```
using (var webClient = new WebClient())
```

```
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

このでは、「」をして、したらWebクライアントがしくクリーンアップされたことをし、のパラメータのURLからされたリソースをローカルハードドライブの2のファイルパラメータ。

のパラメータは " System.Uri "で、2のパラメータは " System.String "です。

また、このをでることもできます。これにより、バックグラウンドでダウンロードがされ、アプリケーションはかのものとやりとりされます。このようなびしをすることは、のアプリケーションではですユーザーインターフェイスのをします。

Asyncメソッドをすると、をできるようにするイベントハンドラをすることができます。たとえば、のようなバーをできます。

```
var webClient = new WebClient())
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

しかし、Asyncのバージョンをしているのえておくべきなの1つは、「」でそれらをすることににしてください。

このはにです。ダウンロードファイルメソッドをびすとすぐにります。これをブロックにれているは、そのブロックをしてすぐにクラスオブジェクトをして、のダウンロードをキャンセルします。

をするために「する」をするは、がするまでみブロックにとどまるようにしてください。

TCPクライアント

Cアプリケーションで`async/await`をすると、マルチスレッドがになります。これは、 `async/await` をTcpClientとともにするです。

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // Asynchronsly attempt to connect to server
    await client.ConnectAsync(host, port);
```

```
    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivilent but safer than calling Close();
```

なUDPクライアント

このコードでは、UDPクライアントをし、ネットワークでしたに "Hello World"をします。 UDPはコネクションレスなので、リスナーはアクティブであるはなく、にメッセージをブロードキャストします。メッセージがされると、クライアントのがします。

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
     using (var client = new UdpClient())
     {
         IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
         client.Connect(ep);
         client.Send(data, data.Length);
     }
}
catch (Exception ex)
{
     Console.WriteLine(ex.ToString());
}
```

は、のクライアントをするUDPリスナのです。これは、にのポートにあるトラフィックをちけ、そのデータをコンソールにきむだけです。このには、でされていないフラグ ' `done` 'がまれており、これをしてリスナーのとをするものがあります。

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}",
listenEndPoint.ToString());
```

```
        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World"
sent from above client
    }
}
```

オンラインでネットワーキングをむ https://riptutorial.com/ja/csharp/topic/1352/ネットワーキング

93: バイナリシリアル

バイナリシリアルエンジンは.NETフレームワークのですが、ここでしたはCのです。 .NETフレームワークにみまれているのシリアライゼーションエンジンとすると、バイナリシリアライザはかつであり、はさせるためになコードをとしません。ただし、コードにはあまりではありません。つまり、オブジェクトをシリアルしてオブジェクトのをしすると、しくデシリアライズされません。

Examples

オブジェクトをシリアライズにする

`[Serializable]`をして、オブジェクトをバイナリシリアルにマークします。

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

`[NonSerialized]`をしてにオプトアウトしないり、すべてのメンバーがシリアルされます。このでは、 `X` 、 `Y` 、 `Z` 、および`Name`はすべてシリアルされています。

`[NonSerialized]`または`[OptionalField]`マークがいていないり、すべてのメンバーはでするがあります。このでは、 `X` 、 `Y` 、および`Z`はすべてであり、ストリームにしないはデシリアライズにします。 `DontSerializeThis`はに`default(decimal)` 0にされ`default(decimal)`。 `Name`がストリームにする、そのにされます。そうでない、 `default(string)` nullにされ`default(string)`。 `[OptionalField]`のは、しのバージョンをすることです。

によるの

`[NonSerialized]`をすると、オブジェクトでわれたになく、デシリアライズのデフォルト `int`は0、`string`はnull、 `bool`はfalseなどはにそのメンバーになりますコンストラクタ、など。これをうために、 `[OnDeserializing]` にBEFOREデシリアライズとぶと`[OnDeserialized]` のデシリアライズとばれるをする`[OnSerializing]`と`[OnSerialized]`します。

Vectorに「Rating」をし、がに1からまることをしたいとします。にきまれるは、デシリアライズに0になります。

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

このをするには、クラスのにのメソッドをして1にするだけです。

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

または、されたにするは、デシリアライズがしてからするまでつことができます。

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

に、 `[OnSerializing]`と`[OnSerialized]`をって、のきをすることができます。

ISerializableをしてよりくのコントロールをする

それはのよりくの、のとみみのをるでしょう

ISerializableインターフェイスをし、コンパイルするためののコンストラクタをする

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
```

```
    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

データのシリアルでは、するとのタイプをできます

```
info.AddValue("_name", _name, typeof(string));
```

データがデシリアライズされると、のタイプをみることができます

```
_name = (string)info.GetValue("_name", typeof(string));
```

サロゲートISerializationSurrogateの

あるオブジェクトがのオブジェクトのとシリアルをできるようにするセレクタをします。

に、それはできないクラスをにまたはすることができます

ISerializationSurrogateインターフェイスをする

```
public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}
```

に、SurrogateSelectorをしてし、それをあなたのIFormatterにりてることによって、あなたのIFormatterにについてらせるがあります

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

クラスがとマーークされていなくても。

```
//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

なソリューーションン

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext
context)
        {
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext
context, ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
```

```
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

シリアライゼーションバインダ

バインダーをすると、アプリケーションドメインにロードされているタイプをべることができます

SerializationBinderからしたクラスをする

```
class MyBinder : SerializationBinder
```

```
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

これで、どのタイプがロードされているのかをし、にけってほしいものをすることができます

バインダーをするには、バインダーをBinaryFormatterにするがあります。

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

なソリューションン

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
```

```
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var binaryFormatter = new BinaryFormatter();
            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var binaryFormatter = new BinaryFormatter
            {
                Binder = new MyBinder()
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

の

このさなは、これについてにをつけなければ、プログラムのをうをしています。シリアライズプロセスをよりにする

に、プログラムののバ―ジョンのをきます

バ―ジョン1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

そして、プログラムの2のバ―ジョンでしいクラスをしたとしましょう。そして、にするがあります。

これでコードはのようになります

バージョン2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

シリアライズとデシリアライズのコード

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

そして、v2のプログラムでデ―タをシリアライズし、v1のプログラムでデシリアライズしようとするとどうなりますか

あなたにはがあります

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem
in the Formatter.Source=mscorlib
StackTrace:
   at System.Runtime.Serialization.ObjectManager.DoFixups()
   at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler
handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
   at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
   at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

どうして

ObjectManagerには、ととのをするためののロジックがあります。アセンブリにはしないしいのをしました。

ObjectManagerがをしようとすると、グラフがされます。をるとちにそれをすることができないので、ダミ―をしてでをします。

このタイプはアセンブリにまれていないため、をすることはできません。らかので、フィックスののリストからをせず、に "IncorrectNumberOfFixups"がスロ―されます。

シリアライゼ―ションのでは、いくつかの「とし」があります。らかので、しいのにしてのみしくしません。

```
A Note:
Similar code will work correctly if you do not use arrays with new classes
```

それをしてをするのは

- クラスではなくしいのコレクションをするか、ディクショナリなクラスをします。ディクショナリはkeyvalueペアのコレクションなので、
- いコ―ドをできないは、ISerializableをしてください

オンラインでバイナリシリアルをむ https://riptutorial.com/ja/csharp/topic/4120/バイナリシリアル

94: はじめにJson with C

き

のトピックでは、CとシリアライゼーションとシリアルのをしてJsonをするをします。

Examples

なJsonの

```
{
    "id": 89,
    "name": "Aldous Huxley",
    "type": "Author",
    "books":[{
                "name": "Brave New World",
                "date": 1932
              },
              {
                "name": "Eyeless in Gaza",
                "date": 1936
              },
              {
                "name": "The Genius and the Goddess",
                "date": 1955
              }]
}
```

あなたがJsonをめてごじでしたら、ここになチュートリアルがあります。

まずにJsonとするライブラリ

CをしてJsonをするには、Newtonsoft.netライブラリをするがあります。このライブラリは、プログラマがオブジェクトなどをおよびできるメソッドをします。メソッドとそののをりたいはチュートリアルがあります。

Visual Studioをしているは、 *Tools / Nuget Package Manager / Manage Package to Solution* にし、バーに「Newtonsoft」としてパッケージをインストールします。 NuGetをおちでないは、このなチュートリアルをにしてください。

Cの

コードをむに、jsonをってアプリケーションをプログラミングするのにつなをすることがです。

> シリアライゼーション アプリケーションをしてできるバイトストリームにオブジェクトをするプロセス。のコードは、してのjsonにできます。

デシリアライゼーション json /バイトのストリームをオブジェクトにするプロセス。そののシリアルのプロセス。のjsonは、のにすように、Cオブジェクトにシリアルできます。

これをするには、すでにしたプロセスをするためにjsonをクラスにすることがです。 Visual Studioをするは、「Edit / Paste Special / JSONをクラスとしてりけ」をし、jsonをりけるだけで、にjsonをクラスにできます。

```
using Newtonsoft.Json;

  class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;       //"id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type= type;
        this.books = books;
    }
}

 class Book
{
   [JsonProperty("name")]
   public string name;
   [JsonProperty("date")]
   public DateTime date;
}
```

シリアライゼーション

```
 static void Main(string[] args)
    {
        Book[] books = new Book[3];
        Author author = new Author(89,"Aldous Huxley","Author",books);
        string objectDeserialized = JsonConvert.SerializeObject(author);
        //Converting author into json
    }
```

メソッド ".SerializeObject"は、 オブジェクトをパラメータとしてけるので、かをにれることができます。

シリアル

どこからでも、ファイルから、またはサーバーからでもjsonをけることができるので、のコード

にはまれていません。

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

メソッド ".DeserializeObject"は、 " *jsonExample* "を " *Author* "オブジェクトにします。このため、クラスにjsonをすることがなので、メソッドはそれをたすためにアクセスします。

シリアライゼーションとデシリアライゼーションユーティリティ

このサンプルは、すべてのオブジェクトのおよびシリアルにのでした。

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
public static class IGUtilities
{
        public static string Serialization(this T obj)
        {
        string data = JsonConvert.SerializeObject(obj);
        return data;
        }

        public static T Deserialization(this string JsonData)
        {
        T copy = JsonConvert.DeserializeObject(JsonData);
        return copy;
        }

         public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
}
}
```

オンラインではじめにJson with Cをむ https://riptutorial.com/ja/csharp/topic/9910/はじめに-json-with-c-

95: ハッシュ

MD5とSHA1はではないのでけてください。このは、ののためであり、レガシーソフトウェアがこれらのアルゴリズムをしているがあるためです。

Examples

MD5

ハッシュは、のさのバイナリをのさなバイナリにマップします。

MD5アルゴリズムは、128ビットのハッシュ16バイト、32の32をするくされているハッシュです。

System.Security.Cryptography.MD5クラスのComputeHashメソッドは、16バイトのとしてハッシュをします。

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

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

Hello WorldのMD5ハッシュ isED076287532E86365E841E92BFC50D8C

セキュリティの

ほとんどのハッシュとに、MD5ももエンコードもされていません。ブルートフォースによってに

すことができ、やプリィメ―ジにするながあります。

SHA1

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-",String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
 }
```

Hello WordのSHA1ハッシュ is2EF7BDE608CE5404E97D5F042F95F89F1C232871

SHA256

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Hello WorldのSHA256ハッシュ

7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069です。

SHA384

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Hello WorldのSHA384ハッシュ
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD

SHA512

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Hello WorldのSHA512ハッシュは
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

パスワードハッシングのPBKDF2

PBKDF2「パスワードベースのキー2」は、パスワードハッシュにされるハッシュの1つです。これはrfc-2898のです。

.NETの`Rfc2898DeriveBytes` ClassはHMACSHA1についています。

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2にはとがです。

がいと、アルゴリズムがくなるため、パスワードクラッキングがににになります。そのためにはのがされる。 PBKDF2は、えばMD5よりもいである。

は、 のテーブルのハッシュのをぎます。パスワードハッシュとにするがあります。 1つのパスワードたり1つの1つのグローバルではないをします。

Pbkdf2をしたなパスワードハッシングソリューション

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
  {
```

```
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could
be persisted as bytes or as a string using the convenience methods in the next class to
convert from byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary
based attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
      }
    }
    /// <summary>
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
```

```
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
  public sealed class PasswordHashContainer
  {
    /// <summary>
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
```

```
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
```

については、このれたリソースCrackstation - Salted Password Hashing - Rightingをごください。

このソリューションのハッシュは、そのサイトのコードについていました。

オンラインでハッシュをむ https://riptutorial.com/ja/csharp/topic/2774/ハッシュ

96: パラレルLINQPLINQ

- ParallelEnumerable.Aggregatefunc
- ParallelEnumerable.Aggregateseed、func
- ParallelEnumerable.Aggregateseed、updateAccumulatorFunc、combineAccumulatorsFunc、resultSelector
- ParallelEnumerable.AggregateseedFactory、updateAccumulatorFunc、combineAccumulatorsFunc、resultSelector
- ParallelEnumerable.All
- ParallelEnumerable.Any
- ParallelEnumerable.Any
- ParallelEnumerable.AsEnumerable
- ParallelEnumerable.AsOrdered
- ParallelEnumerable.AsParallel
- ParallelEnumerable.AsSequential
- AsynchronousArray
- ParallelEnumerable.Averageセレクタ
- ParallelEnumerable.Cast
- ParallelEnumerable.Concatsecond
- ParallelEnumerable.Containsvalue
- ParallelEnumerable.Containsvalue、comparer
- ParallelEnumerable.Count
- ParallelEnumerable.Count
- ParallelEnumerable.DefaultIfEmpty
- ParallelEnumerable.DefaultIfEmptydefaultValue
- ParallelEnumerable.Distinct
- ParallelEnumerable.Distinctcomparer
- ParallelEnumerable.ElementAtindex
- ParallelEnumerable.ElementAtOrDefaultindex
- ParallelEnumerable.Empty
- ParallelEnumerable.Exceptsecond
- ParallelEnumerable.Exceptsecond、comparer
- ParallelEnumerable.First
- ParallelEnumerable.First
- ParallelEnumerable.FirstOrDefault
- ParallelEnumerable.FirstOrDefault
- ParallelEnumerable.ForAllアクション
- ParallelEnumerable.GroupBykeySelector
- ParallelEnumerable.GroupBykeySelector、comparer
- ParallelEnumerable.GroupBykeySelector、elementSelector
- ParallelEnumerable.GroupBykeySelector、elementSelector、comparer
- ParallelEnumerable.GroupBykeySelector、resultSelector
- ParallelEnumerable.GroupBykeySelector、resultSelector、comparer
- ParallelEnumerable.GroupBykeySelector、elementSelector、ruleSelector

- ParallelEnumerable.GroupBykeySelector、elementSelector、ruleSelector、comparer
- GroupJoininner、outerKeySelector、innerKeySelector、resultSelector
- GroupJoininner、outerKeySelector、innerKeySelector、resultSelector、comparer
- ParallelEnumerable.Intersect
- ParallelEnumerable.Intersectsecond、comparer
- ParallelEnumerable.Joininner、outerKeySelector、innerKeySelector、resultSelector
- ParallelEnumerable.Joininner、outerKeySelector、innerKeySelector、resultSelector、comparer
- ParallelEnumerable.Last
- ParallelEnumerable.Last
- ParallelEnumerable.LastOrDefault
- ParallelEnumerable.LastOrDefault
- ParallelEnumerable.LongCount
- ParallelEnumerable.LongCount
- ParallelEnumerable.Max
- ParallelEnumerable.Maxセレクタ
- ParallelEnumerable.Min
- ParallelEnumerable.Minセレクタ
- ParallelEnumerable.OfType
- ParallelEnumerable.OrderBykeySelector
- ParallelEnumerable.OrderBykeySelector、comparer
- ParallelEnumerable.OrderByDescendingkeySelector
- ParallelEnumerable.OrderByDescendingkeySelector、comparer
- ParallelEnumerable.Range、カウント
- ParallelEnumerable.Repeat、カウント
- ParallelEnumerable.Reverse
- ParallelEnumerable.Selectセレクタ
- ParallelEnumerable.SelectManyselector
- ParallelEnumerable.SelectManycollectionSelector、resultSelector
- ParallelEnumerable.SequenceEqualsecond
- ParallelEnumerable.SequenceEqualsecond、comparer
- ParallelEnumerable.Single
- ParallelEnumerable.Single
- ParallelEnumerable.SingleOrDefault
- ParallelEnumerable.SingleOrDefault
- ParallelEnumerable.Skipcount
- ParallelEnumerable.SkipWhile
- ParallelEnumerable.Sum
- ParallelEnumerable.Sumセレクタ
- ParallelEnumerable.Takecount
- ParallelEnumerable.TakeWhile
- ParallelEnumerable.ThenBykeySelector
- ParallelEnumerable.ThenBykeySelector、comparer
- ParallelEnumerable.ThenByDescendingkeySelector
- ParallelEnumerable.ThenByDescendingkeySelector、comparer

- ParallelEnumerable.ToArray
- ParallelEnumerable.ToDictionarykeySelector
- ParallelEnumerable.ToDictionarykeySelector、comparer
- ParallelEnumerable.ToDictionaryelementSelector
- ParallelEnumerable.ToDictionaryelementSelector、comparer
- ParallelEnumerable.ToList
- ParallelEnumerable.ToLookupkeySelector
- ParallelEnumerable.ToLookupkeySelector、comparer
- ParallelEnumerable.ToLookupkeySelector、elementSelector
- ParallelEnumerable.ToLookupkeySelector、elementSelector、comparer
- ParallelEnumerable.Unionsecond
- ParallelEnumerable.Unionsecond、comparer
- ParallelEnumerable.Where
- ParallelEnumerable.WithCancellationcancellationToken
- ParallelEnumerable.WithDegreeOfParallelismdegreeOfParallelism
- ParallelEnumerable.WithExecutionModeexecutionMode
- ParallelEnumerable.WithMergeOptionsmergeOptions
- ParallelEnumerable.Zipsecond、resultSelector

Examples

な

このは、PLINQをして、のスレッドをして1から10,000ののをするをしています。リストはされないことにしてください

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

WithDegreeOfParallelism

は、をするためににされるタスクのです。

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

AsOrdered

このは、PLINQをして、のスレッドをして1から10,000ののをするをしています。 `AsOrdered`はののパフォーマンスを`AsOrdered`せるがあることにして`AsOrdered`。 であればけられていないがされま

す。

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

AsUnordered

くのをう、がっているとパフォーマンスがするがあります。これをするために、シーケンスがになったときに`AsUnordered`をびすことができます。

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
                          .OrderBy(x => x)
                          .Take(5000)
                          .AsUnordered()
                          .Where(x => x % 2 == 0) // This line won't be affected by ordering
                          .ToList();
```

オンラインでパラレルLINQPLINQをむ https://riptutorial.com/ja/csharp/topic/3569/パラレルlinq-plinq-

97: ビルトインタイプ

Examples

-

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

の - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

の - short、int、longき16ビット、32ビット、64ビット

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;

// assigning a signed long to its minimum value (note the long postfix)
```

```
long l = -9223372036854775808L;

// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

これらのをnullableにすることもできます。つまり、のにえて、nullもりてることができます。nullのがされていないは、0ではなくNULLになります。ヌルは、そののにをけることによってマークされます。

```
int a; //This is now 0.
int? b; //This is now null.
```

の - ushort、uint、ulongなし16ビット、32ビット、64ビット

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;

// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

これらのをnullableにすることもできます。つまり、のにえて、nullもりてることができます。nullのがされていないは、0ではなくNULLになります。ヌルは、そののにをけることによってマークされます。

```
uint a; //This is now 0.
uint? b; //This is now null.
```

の - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

boolキーワードは、System.Booleanのエイリアスです。ブールをするをするために`true`と`false`。

ボックスされたのとの

が`object`のにされている、はボックスされます。は`System.Object`インスタンスにされます。これは、を`==`とするとしないにつながるがあります。

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

これは、オーバーロードされた`Equals`メソッドをすることでできます、されるがられます。

```
var comparison2 = left.Equals(right); // true
```

わりに、 `int`がされるように`left`と`right`をアンボックスすることでじことができます

```
var comparison3 = (int)left == (int)right; // true
```

ボックスされたのの

ボックスされたのは、たとえ2つの`Type`のがであっても、の`Type`にのみアンボックスすることができます。

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

これは、にの`Type`ボックスをすることでできます。

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

オンラインでビルトインタイプをむ https://riptutorial.com/ja/csharp/topic/42/ビルトインタイプ

98: ファイルとストリームのI / O

き

ファイルをします。

- `new System.IO.StreamWriter(string path)`
- `new System.IO.StreamWriter(string path, bool append)`
- `System.IO.StreamWriter.WriteLine(string text)`
- `System.IO.StreamWriter.WriteAsync(string text)`
- `System.IO.Stream.Close()`
- `System.IO.File.ReadAllText(string path)`
- `System.IO.File.ReadAllLines(string path)`
- `System.IO.File.ReadLines(string path)`
- `System.IO.File.WriteAllText(string path, string text)`
- `System.IO.File.WriteAllLines(string path, IEnumerable<string> contents)`
- `System.IO.File.Copy(string source, string dest)`
- `System.IO.File.Create(string path)`
- `System.IO.File.Delete(string path)`
- `System.IO.File.Move(string source, string dest)`
- `System.IO.Directory.GetFiles(string path)`

パラメーター

パラメータ	
パス	ファイルの。
する	ファイルがする、trueはファイルのにデータをし、falseはファイルをきします。
テキスト	きまれるかされるテキスト。
	きまれるのコレクション。
ソース	するファイルの。
dest	ファイルをする。

- に`Stream`オブジェクトをじてください。のように`using`ブロックを`using`か、 `myStream.Close()`でびしてこれをうことができます。
- のユーザーに、ファイルをしようとしているパスにながあることをします。
- `@"C:\MyFolder\MyFile.txt"`ようにバックスラッシュをむパスをするときは、 をするがあります`@"C:\MyFolder\MyFile.txt"`

Examples

System.IO.File クラスをしたファイルからのみり

System.IO.File.ReadAllTextをすると、ファイルのをにみることができます。

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

System.IO.File.ReadAllLinesをして、ファイルをのとしてみることもできます。

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

System.IO.StreamWriter クラスをしてファイルにをきむ

System.IO.StreamWriterクラス

のエンコーディングのストリームにをきむためのTextWriterをします。

WriteLineメソッドをすると、コンテンツをファイルに1WriteLineきむことができます。

StreamWriterオブジェクトがスコープかられてファイルがじられるとすぐにされるようにするusingキーワードのにしてください。

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

StreamWriterは、そのコンストラクタに2のboolパラメータをけることができ、ファイルをきするのではなくファイルにAppendすることができます。

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

System.IO.File クラスをしてファイルにきむ

System.IO.File.WriteAllTextをして、をファイルにきむことができます。

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

System.IO.File.WriteAllLinesをして、 IEnumerable<String>を2のパラメータとしてけることもできますのではのではありません。これにより、のからをきむことができます。

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

IEnumerableをしてファイルを1ずつみみする

きなファイルをするは、 System.IO.File.ReadLinesメソッドをして、ファイルのすべてのを IEnumerable<string>にみむことができます。これはSystem.IO.File.ReadAllLinesとていますが、ファイルをにメモリにロードするのではなく、ファイルをうときににになります。

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

*File.ReadLines*の2のパラメータはオプションです。エンコーディングをするがあるにできます。

ToArrayやToListなどのをびすと、すべてのがにみまれるため、 ReadLinesをするはになります。このメソッドをするは、 foreachループまたはLINQをしてIEnumerableをすることをおforeachます。

ファイルの

ファイルクラス

Fileクラスの Createメソッドをすることにより、 Fileを Createすることができます。メソッドは、されたパスにファイルをし、にファイルをき、ファイルのFileStreamます。あなたがそれをした、ファイルをじることをしてください。

ex1

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// you can write to the fileStream1
}
```

ex3

```
File.Create("samplePath").Close();
```

FileStream クラス

このクラスのコンストラクタにはくのオーバーロードがありますが、にはここでしくしています

。のは、このクラスのもっともされているをカバーするです。

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

これらのリンクから、 FileMode 、 FileAccess 、およびFileShareのをチェックすることができます。にはのようなです

FileMode Answers "ファイルをするがありますかオープンしていないはしてからオープンしますか"ちょっとした。

FileAccess Answers "ファイルをみみ、ファイルにきむか、そのにきむことができますか"ちょっとした。

FileShare Answers「にしているにのユーザーがファイルをみきできるか」ちょっとした。

ファイルをコピーする

ファイルクラス

Fileクラスは、こののためににできます。

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

このでは、ファイルがコピーされます。つまり、ソースからみられ、パスにきまれます。これはリソースをするプロセスで、ファイルサイズにながかかり、スレッドをしないとプログラムがフリーズするがあります。

ファイルをする

ファイルクラス

ファイルクラスは、こののためににできます。

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remark1ファイルのインデックスのみをしますファイルがじボリュームでされている。このでは、ファイルサイズのなはかかりません。

Remark2パスののファイルをきすることはできません。

ファイルをする

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

ファイルがしない、 `Delete`はをスローしませんが、されたパスがであるか、びしになアクセスがないなど、がスローされます。 try-catchブロックの`Delete`へのびしをにラップし、されるすべてのをするがあります。がするがあるは、 ロックステートメントのロジックをラップします。

ファイルとディレクトリ

ディレクトリのすべてのファイルをする

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

されたディレクトリのすべてのファイルをす`FileInfo`のをします。

のをつファイルをする

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

されたをつされたディレクトリのすべてのファイルをす`FileInfo`のをします。

でStreamWriterをしてファイルにテキストをきむ

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
{
   // Can write either a string or char array
   await file.WriteAsync(text);
}
```

オンラインでファイルとストリームのI / Oをむ https://riptutorial.com/ja/csharp/topic/4266/ファイルとストリームのi---o

99: フォントリソースをむ

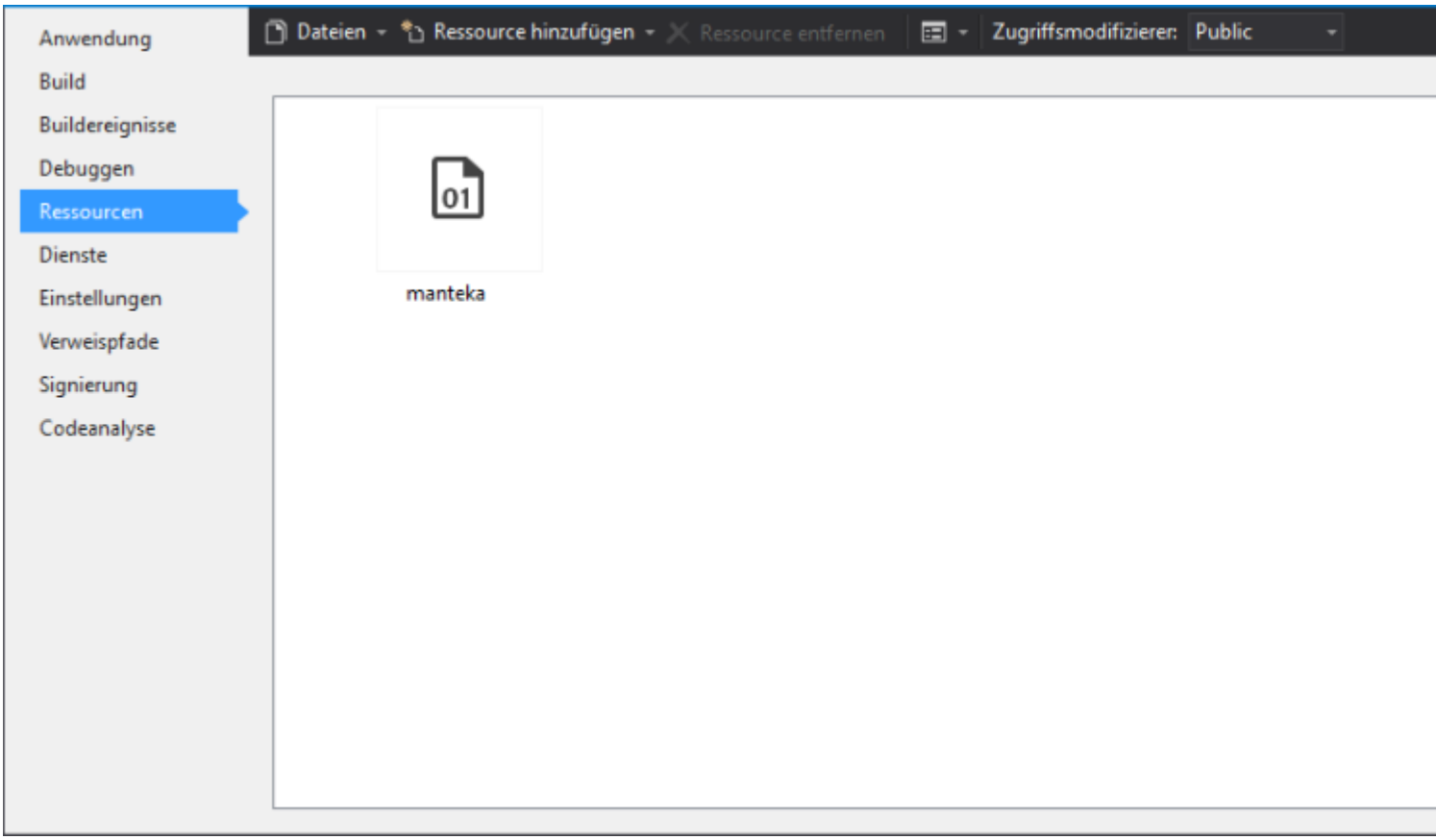
パラメーター

パラメータ	
フォントバイト	バイナリからバイト.ttf

Examples

リソースから 'Fontfamily'をインスタンスする

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```




```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
    return pfc.Families[0];
}
```

'ボタン'での

```
    public static class Res
    {
        /// <summary>
        /// URL: https://www.behance.net/gallery/2846011/Manteka
        /// </summary>
        public static FontFamily Maneteke =
GetResourceFontFamily(Properties.Resources.manteka);

        public static FontFamily GetResourceFontFamily(byte[] fontbytes)
        {
            PrivateFontCollection pfc = new PrivateFontCollection();
            IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
            Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
            pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
            Marshal.FreeCoTaskMem(fontMemPointer);
            return pfc.Families[0];
        }
    }

    public class FlatButton : Button
    {
        public FlatButton() : base()
        {
            Font = new Font(Res.Maneteke, Font.Size);
        }

        protected override void OnFontChanged(EventArgs e)
        {
            base.OnFontChanged(e);
            this.Font = new Font(Res.Maneteke, this.Font.Size);
        }
    }
```

オンラインでフォントリソースをむをむ https://riptutorial.com/ja/csharp/topic/9789/フォントリソースをむ

100: フライウェイトデザインパターンの

Examples

RPGゲームでのマップの

フライウェイトはパターンの1つです。したオブジェクトでできるだけくのデータをすることによって、されるメモリのをらすためにされます。このはFlyweight DPをしくうをえてくれるでしょう。

あなたになでそれのアイデアをしましょう。あなたがRPGゲームにりんでおり、いくつかのキャラクターをむなファイルをロードするがあるとします。えば

- #はです。あなたはそれをくことができます。
- $はです
- @はです。あなたはそれをくことはできません。
- %はです

マップのサンプル

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@#############@@@@@@@@@@@@@#$@@@

@###############@@@@@@@@@@@@@@@@@

@########%#######@#############@@@

@#############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

これらのオブジェクトはのをつため、マップフィールドごとにのオブジェクトをするはありません。はフライウェイトをするをします。

フィールドがするインターフェースをしましょう

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

これで、フィールドをすクラスをできます。とかそれらをするがありますはをしました

```
public enum FieldType
{
    GRASS,
```

```
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

がったように、たちはフィールドにのインスタンスをするはありません。フィールドのリポジトリをするがあります。 Flyweight DPのは、なにのみオブジェクトをにし、リポジトリにはまだしないか、すでにするはオブジェクトをすことです。これをするなクラスをしましょう

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
            default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); //add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
    }
    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
```

```
        if (f != null) return f;
        else return AddField(type);
    }
}
```

すばらしいですこれでコードをテストできます

```
public class Program
{
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}
```

コンソールのはのようになります。

> Grassのしいインスタンスをしました
>
> ロックのしいインスタンスをしました

しかし、なぜ々は2それをしたい、はだけされますかこれは、めて`GetField` grassインスタンスをリポジトリにしないとふためです 。されましたが、にがなときはにしているので、すだけです。

オンラインでフライウェイトデザインパターンのをむ https://riptutorial.com/ja/csharp/topic/4619/フライウェイトデザインパターンの

101: プリプロセッサディレクティブ

- #define *[symbol]* //コンパイラシンボルをします。
- #undef *[symbol]* //コンパイラシンボルをします。
- #warning *[*メッセージ] //コンパイラをします。 #ifでです。
- #error *[*エラーメッセージ] //コンパイラエラーをします。 #ifでです。
- #line *[]*ファイル //コンパイラおよびオプションでソースファイルをきします。 T4テキストテンプレートでされます 。
- #pragma warning [disable | restore] *[warning numbers]* //コンパイラのをします。
- #pragma checksum " *[*ファイル*]* " " *[guid]* " *[*チェックサム] "//ソースファイルのをします。
- #region *[region name]* //りたたみなコードをします。
- #endregion //コードブロックをします。
- #if *[condition]* //がであれば、のコードをします。
- #else // #ifのにされます。
- #elif *[condition]* // #ifのにされます。
- #endif // #ifでまるブロックをします。

プリプロセッサディレクティブは、なるでにソースプログラムをしやすくコンパイルするためにされます。ソースファイルのディレクティブは、プリプロセッサにのアクションをするようします。たとえば、プリプロセッサは、テキストのトークンをきえたり、のファイルのをソースファイルにしたり、テキストのセクションをしてファイルののコンパイルをしたりすることができます。プリプロセッサラインは、マクロのにされされます。したがって、マクロがプリプロセッサコマンドのようにえるものにされた、そのコマンドはプリプロセッサによってされません。

プリプロセッサ・ステートメントは、ソース・ファイル・ステートメントとじキャラクタ・セットをしますが、エスケープ・シーケンスはサポートされません。プリプロセッサでされるキャラクタ・セットは、キャラクタ・セットとじです。プリプロセッサは、のもします。

`#if` 、 `#if` `#elif`などは、ブールのされたサブセットをサポートします。らです

- `==`と`!=` 。これらは、シンボルがかされていないかどうかのテストにのみできます。
- `&&` 、 `||` 、 `!`
- `()`

えば

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

「OK」とされるコードをコンパイルします。 `DEBUG`がされていないは、 `SOME_SYMBOL`または`SOME_OTHER_SYMBOL`がされ、 `RELEASE`がされているは、コンソールにされます。

これらのはコンパイルにわれるため 、にすることはできません。 `#if`をしてされたコードは、コ

ンパイラののではありません。

MSDNのCプリプロセッサディレクティブ

Examples

がコンパイルされると、どのディレクティブがされているかによってなるがされます。

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

は、、デバッグビルドのをするためにされます。

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
    }
}
```

コンパイラのとエラーの

コンパイラのをしてすることができ`#warning`ディレクティブを、そしてエラーがにしてすることができます`#error`ディレクティブを。

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

シンボルのと

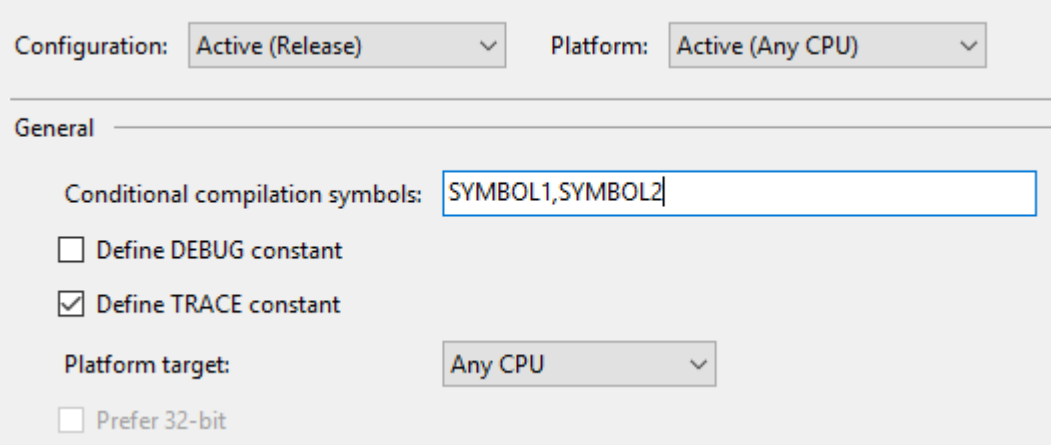
コンパイラシンボルは、コンパイルにされるキーワードであり、きでコードののセクションをす

るためにチェックできます。

コンパイラシンボルをするには3つのがあります。それらはコードによってすることができます

```
#define MYSYMBOL
```

Visual Studioの[プロジェクトプロパティ] - [ビルド] - [きコンパイルシンボル]でできます。



*DEBUG*と*TRACE*はのチェックボックスがあり、にするはありません。

または、Cコンパイラ`csc.exe`の`/define:[name]`スイッチをしてコンパイルにできます。

また、 `#undefine`ディレクティブをしてシンボルをにすることもできます。

もなは`DEBUG`シンボルで、アプリケーションがデバッグモードリリースモードとしてでコンパイルされたときにVisual Studioによってされます。

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
#if DEBUG
        System.Diagnostics.Trace.WriteLine("Unhandled exception!");
        System.Diagnostics.Trace.WriteLine(ex);
        throw;
#else
        LoggingFramework.LogError(ex);
        DisplayFriendlyErrorMessage();
#endif
    }
}
```

のでは、アプリケーションのビジネスロジックにエラーがすると、アプリケーションがデバッグ

モードでコンパイルされ`DEBUG`シンボルがされていると、エラーがトレースログにきまれ、がびしますデバッグのためにげられます。ただし、アプリケーションがリリースモードでコンパイルされていて`DEBUG`シンボルがされていない、ロギングフレームワークをしてエラーをかにし、エンドユーザにフレンドリなエラーメッセージがされます。

ブロック

りたたみなコードをするには`#region`と`#endregion`をします。

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

これらのディレクティブは、コードをするためにりたたみな Visual Studioなどをサポートする IDEをするにのみです。

そののコンパイラの

ライン

`#line`は、とエラーをするときにコンパイラによってされるとファイルをします。

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

プラグマチェックサム

`#pragma checksum`は、デバッグにされたプログラムデータベースPDBののチェックサムをします。

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

きの

`System.Diagnostics`の`Conditional`をメソッドにすることは、ビルドでびされるメソッドとそうでないメソッドをするクリーンなです。

```
#define EXAMPLE_A

using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
    static void ExampleA() {...}

    [Conditional("EXAMPLE_B")]
    static void ExampleB() {...}
}
```

コンパイラのと

`#pragma warning disable`をしてコンパイラのを`#pragma warning disable`、 `#pragma warning restore`を`#pragma warning restore`することができます。

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;

#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

カンマりのをできます。

```
#pragma warning disable CS0168, CS0219
```

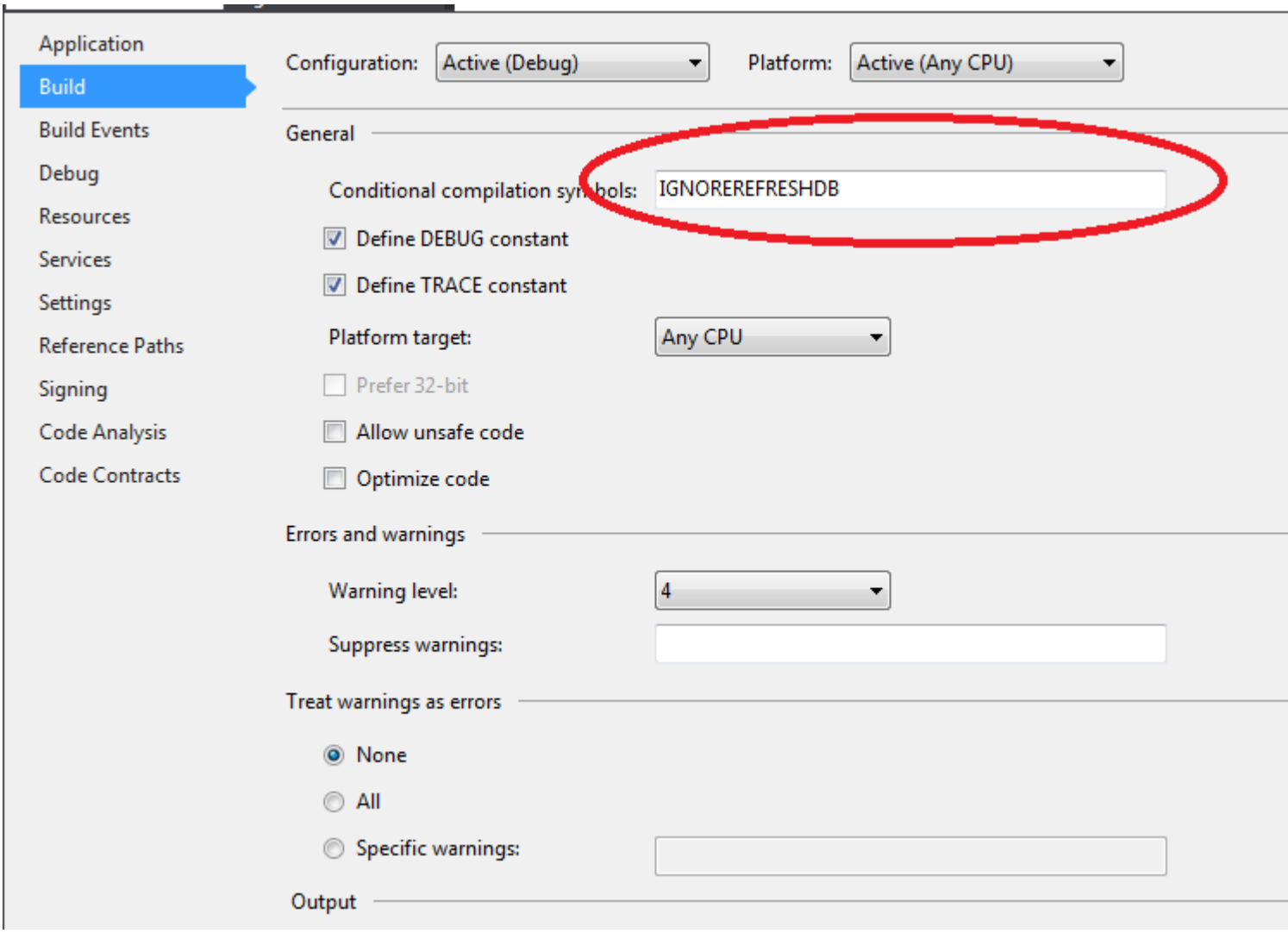
`CS`プレフィックスはオプションで、させることもできますこれはベストプラクティスではありません。

```
#pragma warning disable 0168, 0219, CS0414
```

プロジェクトレベルでのカスタムプリプロセッサ

テストのためにいくつかのアクションをスキップするがあるときに、プロジェクトレベルでカスタムきをするとです。

`Solution Explorer` - >プロジェクトを□クリックして、をする - > `Properties` - > `Build` - >フィールド `Conditional compilation symbols`をし、をここにします



いくつかのコードをスキップするコード

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

オンラインでプリプロセッサディレクティブをむ https://riptutorial.com/ja/csharp/topic/755/プリプロセッサディレクティブ

102: プレーンテキストエディタとCコンパイラcsc.exeをしたコンソールアプリケーションの

Examples

プレーンテキストエディタとCコンパイラをしたコンソールアプリケーションの

プレーンテキストエディタをしてCでされたコンソールアプリケーションをするには、Cコンパイラがです。 Cコンパイラcsc.exeは、のにあります。
`%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

システムにインストールされている.NET Frameworkのバージョンにじて、のパスをするがあります。

コードをする

このトピックのは、Cコンパイラのもないし、のプレーンテキストエディタをして、 どのようにコンソールアプリケーションをするが、どのように[のファイルをする] 1をコンパイルするをえるために、あなたをえることではありませんたとえば、メモ。

1. キーボードショートカット「 `Windows`キー + `R`」をして、「」ダイアログをきます。
2. `notepad` し 、 `Enter`キーを します
3. のサンプルコードをメモにりけます
4. ファイルをし`ConsoleApp.cs` 、そのにる、...→をけてを ファイルにくことによって、 `ConsoleApp.cs` 、にし、テキストフィールド「ファイル」をして`All Files`ファイル·タイプとして。
5. `Save`クリックし`Save`

ソースコードのコンパイル

1. `Windows`キー + `R`をして、[]ダイアログをきます
2.をします。

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

ここで、々 `ConsoleApp.cs`ファイルをしたに`ConsoleApp.cs`ます。 ファイル `ConsoleApp.exe` がされます。 `ConsoleApp.exe`をダブルクリックしてきます。

それでおしまいコンソールアプリケーションがコンパイルされました。ファイルがされました。これでなコンソールアプリケーションがされました。

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;

            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

オンラインでプレーンテキストエディタとCコンパイラcsc.exeをしたコンソールアプリケーションのをむ https://riptutorial.com/ja/csharp/topic/6676/プレーンテキストエディタとc-コンパイラ-csc-exe-をしたコンソールアプリケーションの

103: プロパティ

プロパティは、フィールドのクラスデータストレージとメソッドのアクセシビリティをします。プロパティ、フィールドをするプロパティ、またはフィールドをするメソッドのどちらをするかをめるのがしいがあります。として、

- がおよび/またはされるのみ、プロパティはフィールドなしでするがあります。のロジックはしません。そのような、フィールドをすると、のためにコードをすることになります。
- データをまたはするがあるときは、プロパティをフィールドとともにするがあります。たとえば、のとのスペースをしたり、がでないことをしたりすることがあります。

メソッドvsプロパティにして、の `get` と `set` のがなは、プロパティがしています。また、.Netはクラスのをするくのをします。たとえば、フォームにグリッドをすると、.Netはデフォルトでそのフォームのクラスのすべてのプロパティをします。したがって、このようなは、このるいがにましいにプロパティをすることをしています。また、にをしたくないメソッドもあります。

Examples

コンテキストのさまざまなプロパティ

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned
through the public property.
    public DateTime DOB
    {
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
    }

    //this is not a property but a method; though it could be rewritten as a property if
desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
```

```
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}
```

ゲット

ゲッター—は、クラスからをするためにされます。

```
string name;
public string Name
{
    get { return this.name; }
}
```

セット

セッター—は、プロパティにをりてるためにされます。

```
string name;
public string Name
{
    set { this.name = value; }
}
```

プロパティへのアクセス

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = "   Hans Trimmer  ";
        aPerson.DOB = new DateTime(1961, 11, 11);
```

```
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only
accessible within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as
there's validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
    }
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
```

```
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB.  The date
must not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}
```

プロパティのデフォルト

デフォルトをするには、InitializersC6をします。

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

みみのは、のようなをすことができます

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
  }
```

されたプロパティ

されたプロパティはC3でされました。
されたプロパティは、のgetterおよびsetterアクセサでされます。

```
public bool IsValid { get; set; }
```

されたプロパティがコードにきまれると、コンパイラはプロパティのアクセサをしてのみアクセスできるプライベートなフィールドをします。

のされたプロパティステートメントは、このいコードをくこととじです

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

されたプロパティは、アクセサにロジックをめることはできません。たとえば、のようになります。

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

ただし、されたプロパティには 、そのアクセサのアクセスがなるがあります。

```
public bool IsValid { get; private set; }
```

C6では、されたプロパティにはセッターをたないそのはコンストラクタでのみできるか、またはハードコードされているため、にすることができます。

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

されたプロパティののについては、 Auto-property initializersのドキュメントをしてください。

みりプロパティ

な、には、 `readonly`プロパティは、 `readonly`キーワードでマークされているプロパティです。それはしくないとにのコンパイルのエラーです

```
public readonly string SomeProp { get; set; }
```

プロパティはゲッターをっているのみみりです。

```
public string SomeProp { get; }
```

みりプロパティをしてクラスをする

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

オンラインでプロパティをむ https://riptutorial.com/ja/csharp/topic/49/プロパティ

104: プロパティの

プロパティのをめるときは、ににするために、されたプロパティからめてください。

にじてバッキングフィールドをつプロパティにりえます。なセットのがなは、バッキングフィールドをするがあります。

Examples

C6.0プロパティの

getterおよび/またはsetterでプロパティをし、すべてを1でします。

```
public string Foobar { get; set; } = "xyz";
```

バッキングフィールドによるプロパティの

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

コンストラクタのプロパティの

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

オブジェクトのインスタンスのプロパティの

プロパティは、オブジェクトのインスタンスにできます。

```
var redCar = new Car
{
    Wheels = 2,
    Year = 2016,
    Color = Color.Red
};
```

オンラインでプロパティのをむ https://riptutorial.com/ja/csharp/topic/82/プロパティの

105: ポインタ

ポインタと unsafe

その、ポインタはなコードをします。したがって、ポインタをするには、 `unsafe`コンテキストがです。

`System.IntPtr`は、 `void*`むなラッパーです。これは、ではないコンテキストがのタスクをでするがない、 `void*`よりなとしてされています。

の

CやC ++とに、ポインタのはのビヘイビアをびすがあり、メモリやしないコードのがのがあります。ほとんどのポインタのなのため、ポインタのしいはプログラマのです。

ポインタをサポートする

CやC ++とはなり、すべてのCにするポインタがあるわけではありません。ののがされる、タイプ`T`はするポインタタイプをすることができる。

- `T`は、structまたはポインタです。
- `T`は、これらののをにたすメンバーのみがまれます。

Examples

アクセスのポインタ

このは、Cのへの Cのようなアクセスのためにポインタをするをしています。

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
        }
    }
}
```

`unsafe`キーワードは、のでCにアクセスするときにされるチェックをポインタへのアクセスが

unsafeためにです。

fixedキーワードは、セーフなでオブジェクトをfixedするをするようにCコンパイラにします。ガベージコレクタがメモリのをしないようにするには、をすポインタをにするために、ピンでするがあります。

ポインタ

ポインターのとはとはなります。ポインタがインクリメントまたはデクリメントされると、それがしすアドレスはのサイズだけします。

えば、タイプintのSystem.Int32のきさをしているintアドレス0にすることができ、そのののintこれにアドレス4にされ、そしてすることができます。コード

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

に、タイプlongのSystem.Int64のきさをするlongアドレス0にすることができ、そのののlongように、アドレス8にされ、そしてすることができます。コード

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

voidはなものであり、 voidのポインタもなもので、がわからないときやでないときにcatch-allポインタとしてわれます。サイズにしないのため、 voidポインタはインクリメントまたはデクリメントできません。

```
var ptr = (void*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

アスタリスクはタイプのです

CおよびC ++では、ポインタののアスタリスクは、されているのです 。 Cでは、のアスタリスクがのです 。

C、C ++、およびCでは、のスニペットはintポインタをしています。

```
int* a;
```

CおよびC ++では、のスニペットは`int`ポインタと`int`をしています。 Cでは、2つの`int`ポインタをしています。

```
int* a, b;
```

CおよびC ++では、のスニペットは2つの`int`ポインタをしています。 Cではです

```
int *a, *b;
```

void *

Cは、C ++やC ++からし、 `void*`にしない、サイズにしないポインタとしてします。

```
void* ptr;
```

のをして、のポインタを`void*`りてることができます。

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

のは、ながです。

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

->をしたメンバーアクセス

CがCとC ++シンボルのを`->`けされたポインタをしてインスタンスのメンバにアクセスするとして。

のをえてみましょう

```
struct Vector2
{
    public int X;
    public int Y;
}
```

これは、メンバーにアクセスするための`->`のです。

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
```

```
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

なポインタ

ポインタをサポートするためにタイプがたさなければならない は、ジェネリックのからすることはできません。したがって、ジェネリックのパラメータでされたへのポインタをしようとするとします。

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

オンラインでポインタをむ https://riptutorial.com/ja/csharp/topic/5524/ポインタ

106: ポインタとでないコード

Examples

でないコードの

Cでは、 `unsafe`でマークされているときにコードブロックのでポインタをできます。でないコードまたはアンマネージコードは、ポインタをするコードブロックです。

ポインタは、がののアドレス、つまりメモリのアドレスであるです。のまたはとに、アドレスをするにポインタをするがあります。

ポインタのなはのとおりです。

```
type *var-name;
```

なポインタはのとおりです。

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

のは、でないをしてCでポインタをするをしています。

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

のコードをコンパイルしてすると、のがされます。

```
Data is: 20
Address is: 99215364
```

メソッドをでないとするわりに、コードのをでないものとしてすることもできます。

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

ポインタをしたデータの

ToStringメソッドをして、ポインタがするにされているデータをできます。のはこれをしています

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

のコードをコンパイルしてすると、のがされます。

```
Data is: 20
Data is: 20
Address is: 77128984
```

メソッドへのパラメータとしてのポインタのけし

ポインタをメソッドにパラメータとしてすことができます。のは、これをしています。

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }
```

```
      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

のコードをコンパイルしてすると、のがされます。

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

ポインタをしたへのアクセス

Cでは、と、データとじデータへのポインタはじではありません。たとえば、`int *p`と`int[] p`はじではありません。ポインタ`p`はメモリではされていませんが、アドレスはメモリでされているためインクリメントすることはできません。

したがって、はCやC ++でっていたように、ポインタをしてデータにアクセスするがあるは、fixedキーワードをしてポインタをするがあります。

のはこれをしています

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }

         Console.ReadKey();
      }
   }
}
```

のコードをコンパイルしてすると、のがされます。

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

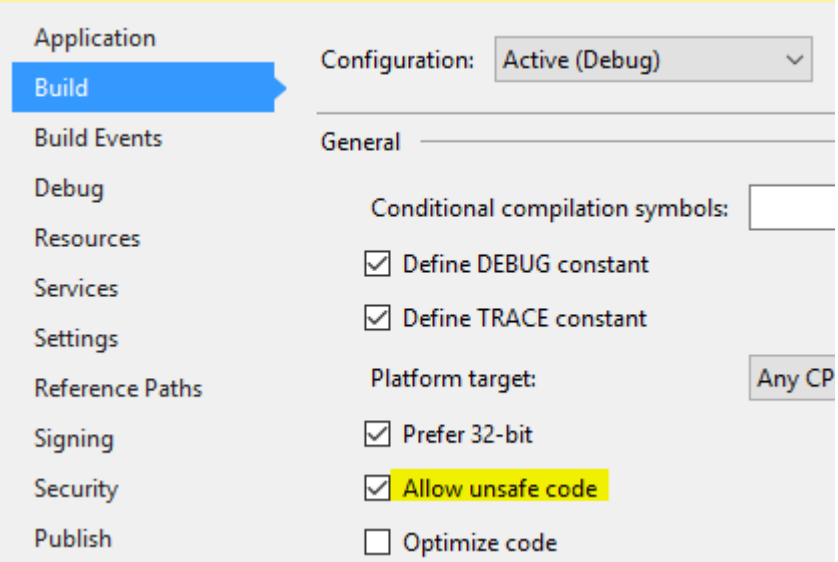
でないコードのコンパイル

でないコードをコンパイルするには、コマンドラインコンパイラで`/unsafe`コマンドラインスイッチをするがあります。

たとえば、でないコードをむprog1.csというのプログラムをコマンドラインからコンパイルするには、のコマンドをします。

```
csc /unsafe prog1.cs
```

Visual Studio IDEをしているは、プロジェクトのプロパティででないコードをできるようにするがあります。



これをする

- ソリューションエクスプローラでプロパティノードをダブルクリックしてプロジェクトプロパティをきます。
- [ビルド]タブをクリックします。
- [でないコードをする]オプションをし、

オンラインでポインタとでないコードをむ https://riptutorial.com/ja/csharp/topic/5514/ポインタとでないコード

107: メソッド

Examples

メソッドの

すべてのメソッドには、アクセサ `public` 、 `private` 、...、オプションの `abstract` 、、およびにじてメソッドパラメ―タからなるのシグネチャがあります。りのはのではないことにしてください。メソッドプロトタイプはのようになります。

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier`は`public` 、 `protected` 、 `pirvate`またはデフォルトで`internal`ます。

`OptionalModifier`は、 `static`または`abstract virtual override new`または`sealed`することができます。

`ReturnType`はりがないは`void`になり`void` 、なクラスのは`int`として、のいずれかのにすることができます。

1つのメソッドには、パラメ―タがあるとないがあります。メソッドのパラメ―タをするには、の `int a` のようにするがあります。のパラメ―タのは、それらのにコンマ `int a, int b` をするがあります。

パラメ―タにはデフォルトがされています。このためには、パラメ―タのをするがあります `int a = 0` 。パラメ―タにデフォルトがある、のはオプションです。

のメソッドは、2つののをします。

```
private int Sum(int a, int b)
{
    return a + b;
}
```

のびし

メソッドをびす

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

メソッドをびしてりをする

```
string input = System.Console.ReadLine();
```

インスタンスメソッドのびし

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

ジェネリックメソッドのびし

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

パラメーターと

メソッドはののパラメーターをできますこのでは、 `i` 、 `s` 、 `o`がパラメーターです。

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

メソッドにをすためにパラメーターをすることができます。そのため、メソッドはそれらのメソッドでします。これは、をする、パラメーターでされるオブジェクトをする、をするなど、あらゆるのがです。

メソッドをびすときは、すべてのパラメーターにのをすがあります。こので、にメソッドびしにすはとばれます。

```
DoSomething(x, "hello", new object());
```

りの

メソッドはnothing `void` またはされたのをすことができます

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

メソッドがりをする、メソッドはをすがあります。これは、 `return`ステートメントをしています。 `return`ステートメントにすると、されたとそれされないコードがされますは`finally`ブロックで

あり、メソッドがるにされます。

メソッドがもさない `void` 、メソッドからすぐにりたいは、をしないで`return`ステートメントをできます。そのようなメソッドのわりには、 `return`ステートメントはです。

な`return`の

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

をスローすると、をさずにメソッドのをできます。また、yieldキーワードをしてりがされるブロックもありますが、これはこのではされないなケースです。

デフォルトパラメータ

パラメータをするためのオプションをするは、デフォルトのパラメータをできます。

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

このようなメソッドをびし、デフォルトがされているパラメーターをすると、コンパイラはそのデフォルトをします。

デフォルトをつパラメータは、デフォルトをたないパラメータのにするがあることにしてください。

```
static void SaySomething(string say, string what = "ehh") {
        //Correct
        Console.WriteLine(say + what);
    }

static void SaySomethingElse(string what = "ehh", string say) {
        //Incorrect
        Console.WriteLine(say + what);
    }
```

このようにするため、デフォルトがになることがあります。メソッド･パラメーターのデフォルトをし、そのメソッドのすべてのびしをコンパイルしないと、それらのびしはコンパイルにしたデフォルトをききし、がするがあります。

メソッドのオーバーロード

じをつのメソッドがなるパラメ―タでされているは、メソッドのオ―バ―ロ―ドとばれます。メソッドのオ―バ―ロ―ドは、、はじですが、なるデ―タをパラメ―タとしてけれるようにされたをします。

をえる

- の
- のタイプ
- りの**

さまざまなをけり、をすをする`Area`というのメソッドをえてみましょう。

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

このメソッドは1つのをけり、をします。メソッドをたとえば`5` とすると、は`"Area of Square is 25"`ます。

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

に、このメソッドに2つのdoubleをすと、は2つののであり、doubleになります。これは、けのほか、の

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

これは、のをつけるためににすることができます。このは、double `radius` をけり、のdoubleをそのAreaとしてします。

これらのメソッドのそれぞれは、することなくにびすことができます。コンパイラはメソッドびしのパラメ―タをべて、するのあるバ―ジョンの`Area`をします。

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

** returnだけでは2つのメソッドをできないことにしてください。たとえば、じパラメ―タをつAreaのが2つあるは、のようになります。

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
```

```
// This will NOT compile.
```

クラスになるをすじメソッドをさせるがあるは、インター―フェイスをしてそのをにすることで、あいまいさのをりくことができます。

```
public interface IAreaCalculatorString {

    public string Area(double width, double height);

}

public class AreaCalculator : IAreaCalculatorString {

    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
```

メソッド

メソッドは、デリゲートパラメータとしてコードブロックをすをします。らはをつメソッドですが、はありません。

```
delegate int IntOp(int lhs, int rhs);
```

```
class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };

        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;

        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}
```

アクセス

```
// static: is callable on a class even when no instance of the class has been created
```

```
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual  void MyMethod()

// internal: access is limited within the current assembly
internal  void MyMethod()

//private: access is limited only within the same class
private  void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the
containing class.
protected internal void MyMethod()
```

オンラインでメソッドをむ https://riptutorial.com/ja/csharp/topic/60/メソッド

108: ユーザとパスワードでネットワークフォルダにアクセスする

き

PInvokeをしてネットワークファイルにアクセスする。

Examples

ネットワークファイルにアクセスするためのコード

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

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

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

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

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

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

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
```

```
        GC.SuppressFinalize(this);
    }

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

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

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

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

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

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

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
    Tree = 0x0a,
    Ndscontainer = 0x0b
```

```
}
```

オンラインでユーザとパスワードでネットワークフォルダにアクセスするをむ
https://riptutorial.com/ja/csharp/topic/9627/ユーザとパスワードでネットワークフォルダにアクセスする

109: ラムダ

ラムダは、をインラインでするためのです。よりには、 Cプログラミングガイド

> ラムダは、デリゲートまたはツリーのにできるです。ラムダをすると、としてすことも、びしのとしてすこともできるローカルをすることができます。

ラムダは、 `=>`をしてされます。オペレータのにパラメータをします。には、これらのパラメータをできるをします。このはのりとしてされます。もっとまれに、にじて、 `{code block}`をですることができます。りのがvoidでない、ブロックにはreturnがまれます。

Examples

ラムダをパラメータとしてメソッドにす

```
List<int> l2 = l1.FindAll(x => x > 6);
```

ここで、 `x => x > 6`は、6をえるのみがされるようにするとしてするラムダです。

デリゲートのとしてのラムダ

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

のラムダのは、のようなコードとです。

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

ラムダは`Func`と`Action`の

、lambdaはなをするためにわれます にlinqので

```
var incremented = myEnumerable.Select(x => x + 1);
```

ここでの`return`はです。

しかし、lambdaとしてアクションをすこともです

```
myObservable.Do(x => Console.WriteLine(x));
```

のパラメータまたはパラメータなしのラムダ

`=>`のにあるのまわりのカッコをして、のパラメ―タをします。

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

に、カッコののセットは、がパラメ―タをけれないことをします。

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

ステ―トメントラムダにのステ―トメントをれる

ラムダとはなり、ステ―トメントラムダにはセミコロンでられたのステ―トメントをめることができます。

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

ステ―トメントは`{}`まれています。

ステ―トメントlambdasはツリ―のにできないことにしてください。

Lambdaは`Func`と`Expression`のとしてすることができます

の`Person`クラスをします。

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

のラムダ

```
p => p.Age > 18
```

のメソッドのとしてすことができます

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

コンパイラはラムダをデリゲ―トと`Expression`のにすることができるためです。

らかに、LINQプロバイダは、クエリをしてストアクエリにできるように、 `Expression`

`IQueryable<T>` に`IQueryable<T>`インターフェイスをしてされていますにきくしています。

イベントハンドラとしてのラムダ

ラムダをしてイベントをすることができます。これはのにです。

- ハンドラはいです。
- ハンドラのをするはありません。

ラムダイベントハンドラがされるいをにします。

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

コードののあるでされたイベントハンドラのをするがある、イベントハンドラのをにし、そのをして/をうがあります。

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

これをわなく、にそれをするためににラムダされている `-=` Cコンパイラは、ずしも2つのがしくしていないということです。

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

のステートメントがラムダにされた、コンパイルにエラーがすることなく、なのがってされるがあることにしてください。えば

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

これはコンパイルされますが、ラムダ`(sender, args) => Console.WriteLine("Email sent");`を し`(sender, args) => Console.WriteLine("Email sent");`イベントハンドラとしてし、`emailSendButton.Enabled = true;`をし`emailSendButton.Enabled = true;`すぐに。これをするには、ラムダのをでむがあります。これは、からをすることでけることができます。ラムダイベントハンドラにのをするときや、ラウンドダッシュをラウンドブラケットでむときはしてください。

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

オンラインでラムダをむ https://riptutorial.com/ja/csharp/topic/46/ラムダ

110: ラムダ

ラムダはにされるをりみ、クロージャをします 。クロージャは、いくつかのコンテキストとにです。コンパイラは、ラムダがそののからのを 'むenclose'たびにクロージャをする。

の

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

safeApplyFilterPredicateは、 filtererののへのプライベートをち、 Invokeメソッドがのようにする、しくされたオブジェクトをします

```
o => (o != null) && filterer.Predicate(i);
```

safeApplyFilterPredicateののがされているsafeApplyFilterPredicate 、 safeApplyFilterPredicateがしているオブジェクトへのがするため、これはfiltererです。これはガベージコレクションにし、filtererいるオブジェクトがfiltererにしないがするがあります。

、クロージャをして、のオブジェクトへのをむビヘイビアをカプセルするエフェクトをすることができます。

えば

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

クローズをしてマシンをモデルすることもできます。

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

Examples

なラムダ

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;

// Behaviourally equivalent to:

int Add1(int i)
```

```
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

LINQをったラムダ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

ラムダをしてクロ―ジャをする

クロ―ジャのについては、をしてください。インター―フェイスがあるとします。

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

のようにされる。

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

`machineClosure`からをする`int`にする`int`し、 `IMachine`インスタンス`machine`をうためにをします。`machine`がになっても、 `machineClosure`オブジェクトがされているり、の`IMachine`インスタンスはコンパイラによってにされる'クロ―ジャ'のとしてされます。

これは、じびしがなるになるをすことをするがありますたとえば、マシンがのをしているなど。くの、これはしないものであり、なスタイルのコ―ドではけるべきです。でしないクロ―ジャ―がバグのとなります。

ブロックボディをむラムダ

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

System.Linq.Expressionsによるラムダ

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

オンラインでラムダをむ https://riptutorial.com/ja/csharp/topic/7057/ラムダ

111: ランタイムコンパイル

Examples

RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript`はしいCスクリプトエンジンです。

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

ステートメント、、メソッド、クラス、またはコードセグメントをコンパイルしてすることができます。

CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider`をしてCクラスをコンパイルできます。

```
var code = @"
    public class Abc {
       public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

オンラインでランタイムコンパイルをむ https://riptutorial.com/ja/csharp/topic/3139/ランタイムコンパイル

112: リアクティブエクステンションRx

Examples

TextBoxでのTextChangedイベントの

Observableは、TextBoxのTextChangedイベントからされます。また、がのとなる、および0.5にがなかったにのみ、すべてのがされます。こののは、コンソールにされます。

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

なデータベースからのストリーミングデータ

`IEnumerable<T>`すメソッドがあるとすると、fe

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Observableをし、メソッドをでします。 `SelectMany`はコレクションをし、サブスクリプションは`Buffer`して200ごとにされます。

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

オンラインでリアクティブエクステンションRxをむ https://riptutorial.com/ja/csharp/topic/5770/リアクティブエクステンション-rx-

113: リテラル

- **bool** trueまたはfalse
- **byte** None、intからにされたリテラル
- **sbyte** None、intからにされたリテラル
- **char**をでみます。
- Mまたはm
- **double** D、d、または
- **float** Fまたはf
- **int**なし、 **int**ののののデフォルト
- **uint** U、u、またはuintのの
- **long** L、l、またはlongのの
- **ulong** UL、ul、Ul、uL、LU、lu、Lu、lU、またはulongのの
- **short**なし、intからにされたリテラル
- **ushort**なし、intからにされたリテラル
- でをみます。オプションでに`@`
- **null** リテラル`null`

Examples

`int`リテラル

`int`リテラルは、にののをしてされている`int`

```
int i = 5;
```

`uint`リテラル

`uint`リテラルは、`U`または`u`をするか、または`uint`ののをしてし`uint`。

```
uint ui = 5U;
```

リテラル

`string`リテラルは、でをラップすることによってされています`"`。

```
string s = "hello, this is a string literal";
```

リテラルにはエスケープシーケンスをめることができます。 エスケープシーケンスをしてください

また、Cがリテラルをサポートしていを 。これらは、でをラップすることによってされています`"`

、としてそれを@エスケープシーケンスリテラルではされ、すべてのがまれています。

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

charリテラル

charリテラルは、でをラップすることによってされています'。

```
char c = 'h';
```

リテラルにはエスケープシーケンスをめることができます。 エスケープシーケンスをしてください

リテラルは、すべてのエスケープシーケンスがされたに、ちょうど1でなければなりません。のリテラルはです。デフォルトの default(char)またはnew char()によってされるは'\0'、またはNULL nullリテラルおよびnullとしないでくださいです。

バイトリテラル

byteにはリテラルサフィックスがありません。リテラルはintからにされます

```
byte b = 127;
```

バイトリテラル

sbyteにはリテラルサフィックスがありません。リテラルはintからにされます

```
sbyte sb = 127;
```

のリテラル

decimalリテラルは、のMまたはmをしてされます。

```
decimal m = 30.5M;
```

リテラル

Dまたはdをするか、またはをしてdoubleリテラルをします。

```
double d = 30.5D;
```

floatリテラルは、Fまたはfをするか、またはをしてされます。

```
float f = 30.5F;
```

いリテラル

longリテラルは、Lまたはlをするか、またはlongののをしてされます。

```
long l = 5L;
```

ulongリテラル

ulongリテラルは、してされたUL、 ul、 Ul、 uL、 LU、 lu、 Lu、またはlU、はのをいてulong。

```
ulong ul = 5UL;
```

いリテラル

shortにはリテラルはありません。リテラルはintからにされます

```
short s = 127;
```

ushortリテラル

ushortにはリテラルがありません。リテラルはintからにされます

```
ushort us = 127;
```

ブールリテラル

boolリテラルはtrueまたはfalseいずれかtrue。

```
bool b = true;
```

オンラインでリテラルをむ https://riptutorial.com/ja/csharp/topic/2655/リテラル

114: ルーピング

Examples

ループスタイル

もなループタイプ。ループのどこにいるのかをるためのながかりがないというしかありません。

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

う

`while`とて`while`が、はではなくループのでされます。この、なくとも1はループをします。

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

にとって

のなループスタイル。インデックス `i` をループするとし、それをすることができます。、のにされます。

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

フォアハ

`IEnumarable`オブジェクトをループするされた。アイテムのインデックスやリストのアイテムについてえるはありません。

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

Foreach メソッド

のスタイルはコレクションのをまたはするためにされますが、このスタイルは、コレクションのすべてのをすぐにびすメソッドにされます。

```
list.ForEach(item => item.DoSomething());

// or
list.ForEach(item => DoSomething(item));

// or using a method group
list.ForEach(Console.WriteLine);

// using an array
Array.ForEach(myArray, Console.WriteLine);
```

このメソッドは`List<T>`インスタンスでしかできず、 `Array`メソッドとしてもできることにしてください。これはLinqのではありません 。

Linq Parallel Foreach

Linq Foreachのように、これはをしています。つまり、コレクションのすべてのアイテムが、されたアクションをににすることをします。

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

ブレーク

には、ループのでループをチェックするがあります。はおそらくよりエレガントです

```
for (;;)
{
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
    }
}
```

ネストされたループや`switch`は、な`break`のものをするがあります。

Foreachループ

foreachは、 IEnumerable IEnumerable<T>がそれをすることにしてくださいをするクラスのオブジェクトをりしします。このようなオブジェクトには、みみがいくつかまれていますが、 List<T> 、 T[] のの、 Dictionary<TKey, TSource> 、およびIQueryableやICollectionなどのインターフェイスにされません。

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

1. ItemTypeタイプはアイテムのなタイプとするはなく、アイテムのタイプからりてることができます
2. ItemTypeわりに、わりにvarをして、 IEnumerableのをすることによって、enumerableObjectからアイテムをします
3. ステートメントは、ブロック、ステートメント、またはのステートメント ; でもかまい;
4. enumerableObjectがIEnumerableしていない、コードはコンパイルされません
5. りしの、のはItemTypeキャストされItemType これがされていなくてもvarをしてコンパイラーされても。をキャストできない、 InvalidCastExceptionがスローされます。

このをえてみましょう。

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

のものとです。

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

whileループ

```
int n = 0;
```

```
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

0
1
2
3
4

IEnumeratorsはwhileループですることができます

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

サンプル

/
ニューヨークメトロ
サンパウロ
ソウル/

For Loop

Forループは、あるのをかけてするのにです。 Whileループとていますが、はにまれています。

Forループはのようにされます。

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

- ループでのみできるしいローカルをします。
- ループは、がであるにのみされます。
インクリメント - ループがされるたびにがどのようにするか。

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

0
1
2
3
4

For Loopにスペースをすこともできますが、するためにはすべてのセミコロンがです。

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

3の

3
5
7
9
11

Do - Whileループ

whileループとていますが、ループののをテストするがなります。 Do - Whileループは、がであるかどうかにかかわらずループを1します。

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

ネストされたループ

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
```

```
}
```

する

`break`にえて、キーワード`continue`ます。ループをにするのではなく、にのをスキップします。のがされているにコードをしたくないにです。

ここになが あります

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

```
9
10
```

`Continue`はwhileループやdo-whileループでもです。 For-loopsは、よくされたで、それほどをることはできません。

オンラインでルーピングをむ https://riptutorial.com/ja/csharp/topic/2064/ルーピング

115: ロックステートメント

- ロックobj{}

lockステートメントをすると、コードブロックのコードへのなるスレッドのアクセスをできます。をぐためによくされます。たとえば、のスレッドがコレクションからをみんだり、したりするなどです。ロックがスレッドをにのスレッドがコードブロックをするのをつようにすると、のメソッドでできるをきこすがあります。

MSDN

> lockキーワードは、のオブジェクトにするロックをし、ステートメントをしてロックをすることによって、ステートメントブロックをクリティカルセクションとしてマークします。
>
> lockキーワードは、のスレッドがクリティカルセクションにあるに、あるスレッドがコードのクリティカルセクションにらないようにします。のスレッドがロックされたコードをしようとすると、オブジェクトがされるまでブロックされます。
>
> ロックするプライベートオブジェクトをするか、 プライベートオブジェクトをしてすべてのインスタンスにのデータをすることをおめします。

C5.0では、 lockステートメントはのものとです。

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

C4.0では、 lockステートメントはのものとです。

```
System.Threading.Monitor.Enter(refObject);
try
{
    // code
}
finally
{
     System.Threading.Monitor.Exit(refObject);
}
```

Examples

ない

lockなはなセクションです。

のでは、 ReserveRoomはなるスレッドからびされることになっています。ここでをぐもなは、 lock とのです。メソッドは2つのスレッドがにできないようにlockでまれていlock。

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

あるスレッドがのスレッドをしているにそのスレッドがlockブロックにすると、のスレッドはのスレッドをブロックしてブロックをします。

> ロックするプライベートオブジェクトをするか、プライベートオブジェクトをしてすべてのインスタンスにのデータをすることをおめします。

ロックステートメントでののスロー

のコードはロックをします。はありません。のロックステートメントは、 try finallyとしてします

```
lock(locker)
{
    throw new Exception();
}
```

は、 C5.0をしてください。

フォームのlockステートメント

```
lock (x) ...
```

ここで、 xはのであり、

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

ただし、 `x`はだけされます。

ロックでのり

のコードはロックをします。

```
lock(locker)
{
    return 5;
}
```

なについては、 このSOのがされます。

ロックのためにObjectのインスタンスをする

Cのみみ`lock`ステートメントをする、あるのインスタンスがですが、そのはありません。 `object`のインスタンスはこれにです

```
public class ThreadSafe {
  private static readonly object locker = new object();


  public void SomeThreadSafeMethod() {
    lock (locker) {
      // Only one thread can be here at a time.
    }
  }
}
```

NB。インスタンス`Type` のコードでは、このためにすべきではない`typeof(ThreadSafe)`のインスタンスからである `Type`アプリケーションドメインでされているので、ロックのは、り、それはいけないコードをむことができるえば、`ThreadSafe`にロードされますじプロセスに2つのAppDomainがあり、その`Type`インスタンスでロックするとにロックされます。

パターンとき

スタックにりてられた/ローカルにするロック

`lock`をしているのりの1つは、のロッカーとしてのローカルオブジェクトのです。これらのローカル・オブジェクト・インスタンスはのびしごとになるため、 `lock`はどおりにされません。

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

がオブジェクトへのアクセスをするとすると

あるスレッドが `lock(obj)` をびし、のスレッドが`obj.ToString()`びすと、2のスレッドはブロックされません。

```
object obj = new Object();

public void SomeMethod()
{
     lock(obj)
    {
       //do dangerous stuff
    }
 }

 //Meanwhile on other tread
 public void SomeOtherMethod()
 {
   var objInString = obj.ToString(); //this does not block
 }
```

サブクラスがいつロックするかをることをする

には、ベースクラスは、そのサブクラスがのされたフィールドにアクセスするときにロックをするようにされています。

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

テンプレートメソッドをしてロッキングをカプセルするがはるかにです。

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
```

```
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

ボックスされたValueTypeのロックがしない

のでは、プライベートは、モニタリソースがロックされることをして、へのobjectとしてされるため、にまれていobject。ボクシングは、IncInSyncをびすにします。したがって、ボックスされたインスタンスは、がびされるたびにのヒープオブジェクトにします。

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

ボクシングはIncでします

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld       UserQuery+BulemicCounter.counterLock
IL_0008:  box         System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

これは、ボックスされたValueTypeをモニタのロックにできないことをするものではありません。

```
private readonly object counterLock = 1;
```

ボクシングはコンストラクタでしますが、これはロックにしています

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

よりながするにロックをにする

になパターンは、プライベート`List`または`Dictionary`をスレッドセーフなクラスでし、アクセスするたびにロックすることです。

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

`values`にアクセスするメソッドがあるは、コードがにくなるがあります。さらになことは、すべてのをロックすることでそのがいされることです。ロックもにれやすいため、なロックがないとバグをつけにくくなるがあります。

`ConcurrentDictionary`をすることで、にロックされないようにすることができます。

```
public class Cache
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

をすると、 すべてがロックフリーをあるしているため、パフォーマンスがします。

オンラインでロックステートメントをむ https://riptutorial.com/ja/csharp/topic/1495/ロックステー

トメント

116:

Examples

System.String クラス

CおよびNETでは、はSystem.Stringクラスでされます。 `string`キーワードは、このクラスのエイリアスです。

System.Stringクラスはです。つまり、されるとはできません。

したがって、Substring、Remove、Replace、 `+` operatorなどをったのようなにしてするすべてのは、しいをしてします。

デモンストレーションについては、のプログラムをしてください。

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

これはそれぞれ`string`と`mystring`ます。

と

なは、メモリののオブジェクトをするのではなく、されたときにメモリのオブジェクトのしいバージョンをするです。これのもなはみみの`string`です。

のコードをて、 "Hello"というに "world"をします。

```
string myString = "hello";
myString += " world";
```

この、メモリでがこっているかは、2の`string`にするとしいオブジェクトがされるということです。これをきなループのとしてすると、アプリケーションでパフォーマンスのがするがあります。

`string`のなは`StringBuilder`

のコードをする

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

これをすると、メモリの`StringBuilder`オブジェクトがされます。

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1863/

117:

デリゲートは、のメソッドシグネチャをすです。こののインスタンスは、するをつのメソッドをします。メソッドのパラメータにはデリゲートがあるため、この1つのメソッドにのメソッドへのをすことができます。

みみデリゲート `Action<...>` 、 `Predicate<T>` および `Func<...,TResult>`

`System`がまれている`Action<...>` `Predicate<T>`と`Func<...,TResult>`「...」0および16ジェネリックパラメータのをすデリゲートを、0パラメータのため、 `Action`ですジェネリック。

`Func`はりのが`TResult`にするメソッドをし、 `Action`はりのないメソッドvoidをします。どちらのも、のジェネリックパラメータは、メソッドパラメータとにします。

`Predicate`はブールのりをつメソッドをし、Tはパラメータです。

カスタムデリゲートの

きのデリゲートは、 `delegate`キーワードをしてできます。

デリゲートのびし

デリゲートは、メソッドとじをしてびすことができます。デリゲートインスタンスのと、パラメータをむがきます。

にりてる

は、のでりてることができます。

- きメソッドのりて
- ラムダをしたメソッドのりて
- `delegate`キーワードをしたきメソッドのりて。

デリゲートの

`+`をすると、のデリゲートオブジェクトを1つのデリゲートインスタンスにりてることができます。 `-`をすると、のデリゲートからコンポーネントデリゲートをできます。

Examples

きメソッドののとなる

デリゲートにきメソッドをりてるとき、のにじオブジェクトをします。

- それらはクラスのじインスタンスのじインスタンスメソッドです
- それらはクラスでじメソッドです

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

デリゲートの

のは、`delegate`をつタイプ`NumberInOutDelegate`るす、 `int`とす`int`。

```
public delegate int NumberInOutDelegate(int input);
```

これはのようにできます。

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
```

```
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

exampleデリゲートインスタンスは、じでされるSquare。デリゲートインスタンスは、り、びしのとしてします。びしがデリゲートをびし、デリゲートがターゲットメソッドをびします。このインダイレクションは、びしをターゲットメソッドからりします。

あなたはジェネリックデリゲートをすることができ、そのには、あなたはタイプがであることをしてもよいout またはinのでは。えば

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

のジェネリックと、ジェネリックデリゲートにはwhere TFrom : struct, IConvertible where TTo : new()などのがあります。

イベントハンドラなど、マルチキャストにされるデリゲートのおよびをけます。これは、のためにのがコンパイルのとなる、 + がするがあるためです。たとえば、をします。

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

わりに、のジェネリックをしてください

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

また、いくつかのパラメータがrefまたはoutによってされるデリゲートもサポートされています。

```
public delegate bool TryParser<T>(string input, out T result);
```

サンプルTryParser<decimal> example = decimal.TryParse; 、またはのパラメータにparamsがあるは

します。デリゲートタイプにはオプションのパラメータをできますデフォルトをします。デリゲートでは、シグネチャやりのに`int*`や`char*`などのポインタをできます `unsafe`キーワードをする。デリゲートとそのパラメータは、カスタムをつことができます。

ザ.ファンク 、アクションデリゲート

Systemがまれている`Func<..., TResult>`をす、015のなパラメータでデリゲートを`TResult`。

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but
the first as parameters of that type
    Console.WriteLine(output);
}
```

Systemネームスペースには、ジェネリックパラメータのがなる `Action<...>`デリゲートもまれています。これは`Func<T1, .., Tn>`とていますが、に`void`し`void`。

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type
arguments
}
```

`Predicate<T>`は`Func`ですが、に`bool`します。とは、カスタムをするです。とでされたロジックのにじて、 `true`または`false`がされます。したがって、 `Predicate<T>`は`Func<T, bool>`とじようにし、ともしてじでできます。

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");

// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

`Predicate<T>`か`Func<T, bool>`のどちらをするかは、にはのです。 `Predicate<T>`はおそらくのをよりしていますが、 `Func<T, bool>`はCのにみそうです。

それにえて、にのAPIとやりりするときに、オプションのうちの1つだけがなもあります。えば、`List<T>`と`Array<T>`にそのメソッドにして`Predicate<T>`をります。ほとんどのLINQでは`Func<T, bool>`しかけれません。

デリゲートにきメソッドをりてる

きのメソッドは、シグネチャがするデリゲートにりてることができます。

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne`り`int`とす`int`、そのは、デリゲートとする`Func<int,int>`。 `Example.AddOne`は、するシグネチャをつため`addOne`りてることができます。

の

`.Equals()`をびすと、のがされます。

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

これらのは、マルチキャストデリゲートで`+=`または`-=`をするにもされますたとえば、イベントのサブスクライブおよびサブスクライブの。

ラムダによるへのりて

Lambdasをして、デリゲートにりてるメソッドをできます。

```
Func<int,int> addOne = x => x+1;
```

このようにをするときは、のがであることにしてください。

```
var addOne = x => x+1; // Does not work
```

をパラメータとしてす

デリゲートは、きのポインタとしてできます。

```
class FuncAsParameters
{
  public void Run()
  {
    DoSomething(ErrorHandler1);
    DoSomething(ErrorHandler2);
  }

  public bool ErrorHandler1(string message)
  {
    Console.WriteLine(message);
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public bool ErrorHandler2(string message)
  {
    // ...Write message to file...
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public void DoSomething(Func<string, bool> errorHandler)
  {
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
      if (!errorHandler("Some error occurred!"))
      {
        // The handler decided we can't continue
        return;
      }
    }
  }
}
```

デリゲートのマルチキャストデリゲート

+および-をしてデリゲートインスタンスをできます。デリゲートには、りてられたデリゲートのリストがまれています。

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
```

```
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

このでは、 `d3`は`d1`と`d2`デリゲートのみわせであるため、びされるとプログラムは`1`と`System.Int32`のをします。

デリゲートを**void**りのとみわせる

マルチキャストデリゲートが`nonvoid`りのりをつ、びしはにびされるメソッドからのりをけります。のメソッドはききびされますが、りはされます。

```
    class Program
    {
        public delegate int Transformer(int x);

        static void Main(string[] args)
        {
            Transformer t = Square;
            t += Cube;
            Console.WriteLine(t(2));  // O/P 8
        }

        static int Square(int x) { return x * x; }

        static int Cube(int x) { return x*x*x; }
    }
```

`t(2)`はに`Square`をびし、に`Cube`をびします。 Squareのりはされ、のメソッドのり、つまり`Cube`はされます。

なびしマルチキャストデリゲート

、マルチキャストデリゲートをびたいとっていましたが、チェインのいずれかでがしたとしても、びしリストをびすがあります。あなたはがかったので、リストのがしたに`AggregateException`スローするメソッドをしました。

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del,params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }
```

```
    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}
```

これは、

```
Target 2 executed
Target 1 executed
```

SaveInvokeなしでびすと、ターゲット2だけがされます。

デリゲートのクロージャ

クロージャは、 Parentメソッドのやのスコープでされているのメソッドをできるインラインのメソッドです。

> に、クロージャはでできるコードブロックですが、にされたをします。つまり、それをしたメソッドのローカルなどをすることができます。メソッドのがしました。 - **Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();

}
```

.NETのClosureからられた

のをカプセルする

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
```

```
            CallAnotherMethod();
        }
    }
}
```

クリーンなコーディングので、のようなチェックとをFuncとしてカプセルすると、コードをみやすくしやすくなります。のはにですが、のなるをつのDateTimeプロパティがあり、なるみわせをしたいはどうなりますかそれぞれがリターンロジックをしたシンプルな1Funcは、コードのかけのさをし、みやすくすることができます。のFuncのびしをえて、どのようにくのコードがメソッドをにしているのかしてみてください。

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1194/

118:

Examples

な

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

じコードですべてのをすることは、しばしばのではないことにしてください。
これは、のとして、ルーチンがしたによくされます。

のタイプの

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

はにされ、がされるようにしてください。だから、あなたはもなものからめ、らのでわるがあります。ので、1つのキャッチブロックのみがされます。

オブジェクトの

のコードでをしてスローすることはされています。のインスタンスは、のCオブジェクトとじでわれます。

```
Exception ex = new Exception();
```

```
// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

`throw`キーワードをしてをさせることができます。

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

キャッチブロックにしいをげるは、のが ""としてされるようにしてください。

```
void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.Write(ex.ToString());
    }
}
```

この、をすることはできませんが、いくつかのながメッセージにされますのにはブロックによって`ex.InnerException`をしてアクセスできます。

それはのようにされます

System.DivideByZeroExceptionゼロであるためbでりれません--->
System.DivideByZeroException0でしようとしました。
C[...]のUserQuery.g__DoSomething0_0で\ LINQPadQuery.cs36
---スタックトレーースの---
C[...]のUserQuery.g__DoSomething0_0で\ LINQPadQuery.cs42
UserQuery.MainでC[...] \ LINQPadQuery.cs55

このをLinqPadでしている、はそれほどがありませんいつもあなたをけるとはりません。しかし、にしたようににになるエラーテキストをすと、エラーのをするがにされます。このでは、

```
c = a / b;
```

`DoSomething()`します。

.NET Fiddleでしてみてください

にブロック

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

`try / catch / finally`ブロックは、ファイルからみむときににです。
えば

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}
```

tryブロックのには、 `catch`ブロックまたは`finally`ブロックのいずれかがかなければなりません。ただし、キャッチブロックがしないため、によってされます。するに、finallyブロックのステートメントがされます。

ファイルみみでは、 `FileStream OpenRead`がすものとして`using`ブロックをしていたがあります。こ

れは`IDisposable`しています。

`try`ブロックに`return`があっても、 `finally`ブロックはされます。それができないがいくつかあります

- StackOverflowがしたとき。
- `Environment.FailFast`
- アプリケーションプロセスは、はソースによってされます。

WCFサービスのIErrorHandlerの

WCFサービスにIErrorHandlerをすることは、エラーとロギングをするためのれたです。ここにすは、WCFサービスの1つをびしたとしてスローされるのをすべてするがあります。このでは、カスタムオブジェクトをすと、デフォルトのXMLではなくJSONをすもしています。

IErrorHandlerをする

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new
List<Type> { typeof(BaseDataResponseContract) }));
```

```
            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process
them differently
            }

            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name,
webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

このでは、サービスのるいにハンドラをアタッチします。にIEndpointBehavior、IContractBehavior、またはIOperationBehaviorにこれをすることもできます。

サービスビヘイビアへのアタッチ

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
BindingParameterCollection bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();
```

```
            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace
```

Web.configの

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

このトピックでつリンクがいくつかあります

https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-または-client-description-what-methods-do-they-the-する

そのの

HTTPステータスコードが401である、なメッセージをするIErrorHandler

IErrorHandlerはWCFのエラーをしていないようです。

どのようにカスタムWCFエラーハンドラをOKのhttpコードでJSONをすようにするには

HttpClientリクエストのContent-Typeヘッダーはどのようにしますか

カスタムの

のとにスローされるカスタムをすることができます。これは、にをのエラーとできるようにしたいにしています。

このでは、なをするにアプリケーションでするがあるをにするためのカスタムをします。

カスタムクラスの

カスタムをするには、のサブクラスをし`Exception`。

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

カスタムは、キャッチャーにをするににになります。

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
      base($"Parser error in {fileName}:{lineNumber}")
    {
      FileName = fileName;
      LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

さて、 `catch(ParserException x)`を`catch(ParserException x)`すると、をするためののセマンティクスがられます。

カスタムクラスは、のシナリオをサポートするためにのをできます。

げ

プロセス、のはとしてです。このでは、コードはであるとされるをしようとしているため、`FormatException`

です。この、カスタムは ' **InnerException** 'のインクルードをサポートするがあります。

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

によっては、がAppDomainのをえなければならないことがあります。これは、しいパーサーのホットリロードをサポートするために、パーサーがのAppDomainでされているです。 Visual Studioでは、 `Exception`テンプレートをしてこのようなコードをできます。

```
[Serializable]
public class ParserException : Exception
{
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
    // generic automatically generated messages.
    public ParserException(string message)
        : base(message)
    {}

    // Constructor for serialization support. If your exception contains custom
    // properties, read their values here.
    protected ParserException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {}
}
```

ParserExceptionの

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

のキャッチとラップにカスタムをすることもできます。このようにして、くのなるエラーをアプリケーションにとってよりなのエラータイプにすることができます。

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
      $"Failed to read {token} as number.",
      FileName,
      LineNumber,
      ex);
}
```

のカスタムをさせてをするは、に、のように`InnerException`プロパティにのをするがあります。

セキュリティの

のをすると、ユーザーがアプリケーションのをることをすることによってセキュリティがなわれるがあるは、をラップするのはいえです。これは、がするクラスライブラリをするにされます。

をラップせずにカスタムをさせるはのとおりです。

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
}
```

ラップするかアンラップされたしいをしてカスタムをびすは、びしにとってのあるをさせるがあります。えば、クラスライブラリのユーザは、そのライブラリがそのをどのようにしているかについてくをらないかもしれない。クラスライブラリのによってスローされるはがありません。むしろ、ユーザは、クラスライブラリがこれらのをったでどのようにしているかにするをむ。

```
try
{
  // ...
}
catch (IOException ex)
{
  // ...
  throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

をみむ

のようにをにスローするがあります。

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

のようなをスローするとのがわかりにくくなり、のスタックトレースがわれます。もこれをしてはいけませんキャッチとスローのスタックトレースはわれます。

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw ex;
}
```

の

if-thenやwhileループのようなのフローのわりにをうべきではありません。このアンチパターンは、 のとばれることがありま す 。

アンチパターンのをにします。

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
```

```
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

ここでそれをうよりいです

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.Write("Here are your account details: " + found.Details.ToString());
```

キャッチ

あなたのコードでなのをキャッチするはほとんどありませんもないがあります。それのは、コードのバグをすため、するとされるのだけをキャッチするがあります。

```
try
{
     var f = File.Open(myfile);
     // do something
}
catch (Exception x)
{
     // Assume file not found
     Console.Write("Could not open file");
     // but maybe the error was a NullReferenceException because of a bug in the file handling
code?
}
```

よりい

```
try
{
     var f = File.Open(myfile);
     // do something which should normally not throw exceptions
}
catch (IOException)
{
     Console.Write("File not found");
}
// Unfortunatelly, this one does not derive from the above, so declare separatelly
catch (UnauthorizedAccessException)
{
     Console.Write("Insufficient rights");
}
```

それのがしたは、にアプリケーションがクラッシュするため、デバッガでされ、をできます。と

にかくこれらのがしたは、プログラムをしないでください。クラッシュすることはではありません。

はいでもあります。をしてプログラミングエラーをするためです。それはらのためにされたものではありません。

```
public void DoSomething(String s)
{
     if (s == null)
         throw new ArgumentNullException(nameof(s));
     // Implementation goes here
}

try
{
     DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

1つのメソッドからのをする

1つののでをスローすることはできません。 AggregateExceptionsをいこなすことにれていないは、ったことをくすためにのデータをしたくなるかもしれません。もちろん、ではないのデータがのなどよりです。 AggregateExceptionsをってんでいても、にいるかもしれませんし、にそれらをしていることにいていないと、あなたにつかもしれません。

メソッドをすることはにであり、たとえそれがとしてにわっても、スローされたにっていたのをしたいとうでしょう。として、このは、メソッドがどのようにのスレッドにされたタスクがどのようにし、いくつのスレッドがをスローするがあり、これをするがあるかでわかります。あなたがこれからどのようにをけるかもしれないかのかながここにあります

```
    public void Run()
    {
        try
        {
            this.SillyMethod(1, 2);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine(ex.Message);
            foreach (Exception innerException in ex.InnerExceptions)
            {
                Console.WriteLine(innerException.Message);
            }
        }
    }

    private void SillyMethod(int input1, int input2)
    {
        var exceptions = new List<Exception>();
```

```
        if (input1 == 1)
        {
            exceptions.Add(new ArgumentException("I do not like ones"));
        }
        if (input2 == 2)
        {
            exceptions.Add(new ArgumentException("I do not like twos"));
        }
        if (exceptions.Any())
        {
            throw new AggregateException("Funny stuff happended during execution",
exceptions);
        }
    }
```

のネストとキャッチブロックの

1つの/ `try` `catch`ブロックをもう1つのにれにすることができます。

このようにして、のみをすことなくできるさなコードブロックをできます。

```
try
{
//some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
 //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}
```

キャッチブロックにげるときにをみまないようにしてください

ベストプラクティス

カンニングペーパー

う	しないでください
によるフロー	をむフロー

う	しないでください
ロギングによってされたされたをする	する
`throw`をしてをりします。	をスロ―する - `throw new ArgumentNullException()` `throw ex`か、または`throw ex`
みのシステムをスロ―する	のシステムとのカスタムをげる
アプリケ―ションロジックにとってなは、カスタム/されたをスロ―する	カスタム/されたをスロ―して、フロ―にをする
したいをキャッチする	すべてのをキャッチ

をしてビジネスロジックをしないでください。

フロ―はによってすべきではありません。わりにをしてください。コントロ―ルが`if-else`ステ―トメントではっきりとできるは、とパフォ―マンスがするため、をしないでください。

Mr. Bad Practicesののスニペットをえてみましょう。

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

が`Console.WriteLine(myObject.ToString());`したとき`Console.WriteLine(myObject.ToString());`アプリケ―ションはNullReferenceExceptionをスロ―します。バッドプラクティスことにづい`myObject` nullで、キャッチするためにのスニペットをした`NullReferenceException`

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

のスニペットはのロジックしかカバ―していないので、 `myObject`がこのでnullでないはどうすればよいですかロジックのこのはどこでカバ―するがありますか

`Console.WriteLine(myObject.ToString());Console.WriteLine(myObject.ToString());` `try...catch`ブロックのはどうですか

ベストプラクティスはどうですかはこれをどのようにいますか

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

ベストプラクティスは、よりないコードとかつかりやすいロジックでじロジックをしました。

をスローしないでください

のげはです。パフォーマンスにをえます。にするコードでは、デザインパターンをしてパフォーマンスのをにえることができます。 このトピックでは、がパフォーマンスになをえるにつ2つのパターンについてします。

ロギング なしでをしない

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

をみまないでください。をすると、そのがかれますが、でのためにがじます。をするときは、なスタックトレースがされ、メッセージのみがされるようにインスタンスをするがあります。

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

あなたができないをキャッチしないでください

このようなくのリソースは、なぜあなたがそれをキャッチしているでをキャッチしているのかをくします。そのでできるのであれば、をキャッチするがあります。のアルゴリズムをしたり、バックアップデータベースにしたり、のファイルをしたり、30ってからもうやりしたり、にしたりするなど、をするためにかがしたは、エラーをキャッチしてそのことをうことができます。もしあなたがかつにうことができるものがなければ、に "して"をよりいレベルでさせてください。がにで、ののためにプログラムがクラッシュするのながないは、クラッシュさせてください。

```
try
{
    //Try to save the data to the main database.
}
catch(SqlException ex)
{
    //Try to save the data to the alternative database.
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the
exception bubble up to a level where it can be handled.
```

スレッド

AppDomain.UnhandledExceptionこのイベントは、キャッチされていないのをします。システムデフォルトハンドラがをユーザにしてするに、アプリケーションがにするをできるようにします。アプリケーションのにするながなは、でするためにプログラムデータをするなどのがられるがあります。がされないとプログラムデータがするがあるため、がです。

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

Application.ThreadExceptionこのイベントにより、Windowsフォームアプリケーションで、Windowsフォームスレッドでするのをできます。イベントハンドラをThreadExceptionイベントにアタッチして、これらのをします。これにより、アプリケーションはなになります。であれば、はされたブロックでするがあります。

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
        Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
    }
```

に

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (Exception)e.ExceptionObject;
        // your code
    }

static void ThreadException(object sender, ThreadExceptionEventArgs e)
    {
        Exception ex = e.Exception;
        // your code
    }
```

のスロー

あなたのコードは、かがきたときにをスローすることができます。

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/40/

119:

のウィキペディアのは

ソフトウェアエンジニアリングでは、は、をするためののをするソフトウェアパターンです。は、なオブジェクトサービスです。インジェクションとは、それをするオブジェクトクライアントへののきしです。

** このサイトには、5のにをするにするにするがされています。 John Munschによってされたもいをたえは、の**5**のをとしたくほどなアナロジーをしますあなたがってでからをりすと、をきこすがあります。あなたはドアをいたままにしておくかもしれません、ママかパパがあなたにってほしくないかをるかもしれません。あなたはたちがっていない、あるいはれのかをしているかもしれません。あなたがしなければならないことは、「でむものがです」というをべることです。そして、ってべるときにかがあることをします。これはオブジェクトのソフトウェアのでこれがをしているのですか**5**はインフラストラクチャにして

このコードはMEFをしてDLLをにロードし、をします。 ILoggerのは、MEFによってされ、ユーザークラスにされます。ユーザークラスはILoggerのなをけりません。どのタイプのロガーをしているのかかりません。

Examples

MEFをした

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
```

```
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);

        //Fill the imports of this object
        try
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
```

```
}
```

CとASP.NET with Unity

まず、たちのコードでdepedency injectionをうべきはたちのプログラムでは、のクラスとのクラスをしたいとえています。たとえば、のようなコードをつAnimalControllerクラスがあります。

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

このコードをると、すべてがないとっていますが、AnimalControllerはオブジェクト_SantaAndHisReindeerにしています。にのコントローラはテストにく、のコードのはにしいでしょう。

たちがDepedency Injectionをし、 ここでインターフェイスをるべきについてのにい。

UnityがDIをするようにしたい、これをするはにです:) NuGetパッケージマネージャをうと、コードへのをにインポートできます。

> Visual Studio Tools - > NuGet Package Manager - >ソリューションパッケージの - >の - >プロジェクトの - >インストールをクリック

すぐなコメントをつ2つのファイルがされます。

> App-DataフォルダのUnityConfig.csとUnityMvcActivator.cs
>
> UnityConfig - RegisterTypesメソッドでは、コンストラクタにされるをることができます。

```
namespace Vegan.WebUi.App_Start
{

public class UnityConfig
{
    #region Unity Container
    private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
        return container;
    });

    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
```

```
    {
        return container.Value;
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API
controllers (unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not
previously registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IProductRepository, ProductRepository>();

        container.RegisterType<ISanta, SantaAndHisReindeer>();

     }
 }
}
```

UnityMvcActivator - >また、このクラスがUnityをASP.NET MVCとするとうなコメントがあります

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
        var container = UnityConfig.GetConfiguredContainer();


FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().Firs

        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
```

```
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRe
    }

    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

は、コントローラをクラスSantAndHisReindeerからりすことができます:)

```
 public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

アプリケーションをするににうべきことが1つあります。

Global.asax.csでは、UnityWebActivator.Startというしいをするがあります。これは、Unityをし、タイプをします。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/5766/

120: のとの

- しpublic void Doubleref int numberToDouble{}

き

の

のは、2つのうちのなものです。は、データをすためによくわれます。 3Dの、ブール、またはポイントは、すべてなタイプのです。

は、structキーワードをしてされます。しいをするのについては、セクションをしてください。

に、のをするためにされる2つのキーワードがあります。

-
-

はややです。は、オブジェクトプログラミングのでのなオブジェクトです。したがって、およびそのをサポートし、ファイナライザをサポートします。

Cではにのようながあります。

- クラス
-
- インターフェイス

しいクラスは、classキーワードをしてされます。については、しいをするのセクションをしてください。

ない

とのないをにします。

のはスタックにし、はヒープにします

これはしばしばされる2つのいですが、には、Cでをすると、プログラムはそのをしてそのをします。 int mine = 0とうと、mineは0をしてです。しかし、は、なオブジェクトへのをにしていますがするように。これはC ++などののののポインタにています。

すぐにこれにかないかもしれませんが、はそこにあり、でです。については、のをしてください。

このは、ののののなであり、るがあります。

のは、メソッドでをしてもされません。

のがパラメ―タとしてメソッドにされたときに、メソッドがをした、そのはされません。に、じのメソッドにをし、それをすると、そのじオブジェクトをするのものは、のではなくたにされたオブジェクトをちます。

については、とメソッドののをしてください。

もしがそれらをしたいのですが

"ref"キ―ワ―ドをしてメソッドにすだけで、このオブジェクトをしします。、それはメモリのじオブジェクトです。あなたがうはされます。については、しのをしてください。

のをnullにすることはできません。

かなりわれるように、にnullをりてることができます。つまり、りてたにはのオブジェクトがりてられていないことをします。ただし、の、これはです。しかし、これがであれば、のをnullにできるようにNullableをうことができますが、これがあなたのえであるは、クラスがのアプロ―チではないかどうかをくえるタイプ。

Examples

のでのの

```
public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

printListリストがタイプミスのにのよりにされていても、PrintPrintingListメソッドはされたをします。

```
Scott Duke
Vincent Kong
Craig Brett
```

これは、のリストがじへのリストをしているためです。となるのオブジェクトをすることにより、いずれかのリストによってにします。

のクラスはのようになります。

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

し

とメソッドののがしくするようにするには、しするパラメータとメソッドのびしに、メソッドのシグネチャでrefキーワードをします。

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

これらのをえると、のがどおりにわれます。つまり、numberのコンソールは8になります。

refキーワードをしてし。

ドキュメントから

> Cでは、はまたはのいずれかによってパラメータにすことができます。しにより、メンバ、メソッド、プロパティ、インデクサ、、およびコンストラクタはパラメータのをし、そのをびしでさせることができます。でパラメーターをすには、 `ref`または`out`

キーワードをします。

refびoutつまりoutされたパラメータをいてされたends.inコントラストパラメータにりてなければならないことをrefまたはのままにすることができます。

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

```
Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7
```

りて

```
var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well
```

に`List<int>`のコピーをしません`List<int>`。わりに、を`List<int>`にコピーします。このようなをさせるをびします。

メソッドパラメータのrefとoutとのい

でをすには、`ref`と`out` 2つのがあります。いは、`ref`をすことによってをするがありますが、それをすときではない`out`です。`out`をすると、メソッドびしのににがされます。

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
    value += 4 // CS0269: Use of unassigned out parameter `value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter
`value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;   // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165  Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

キャッチは、していることである`out`パラメータを`must`ので、のがつことがであり、をれるにすること`ref`とではなく、`out`

```
public void EmtyRef(bool condition, ref int value)
```

```
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current
method
```

これは、 `condition`がしないと、 `value`がりてられなくなるためです。

refとoutのパラ メータ

コード

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the
```

```
method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30

Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

オンラインでのとのをむ https://riptutorial.com/ja/csharp/topic/3014/のとの

121:

Examples

のでをする

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

されたのの

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

Substringストリングは、されたインデックスから、または2つのインデックスをむのSubstringします。

がされたシーケンスでまるかどうかをする

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

のをつける

System.String.Containsをすると、のがにするかどうかをべることができます。このメソッドはbooleanをします。がするはtrueをし、そうでないはfalseをします。

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

のおよび/またはのなのトリミング。

String.Trim()

```
string x = "   Hello World!    ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi!*";
```

```
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

String.TrimStart()および**String.TrimEnd()**

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

の

String.Format()メソッドをして、の1つのをされたオブジェクトのにきえます。

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

のをしいにする

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

をにパディングする

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);        // paddedLeft = "  Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo+++"
string noPadded = s.PadLeft(2);          // noPadded = "Foo" (original string is never
shortened)
```

からをする

String.Joinメソッドは、またはのからをするのにちます。このメソッドは2つのパラメータをけれます。のものは、のをるのにつデリミタまたはセパレータです。 2のパラメータはそのものです。

char arrayからの**char array**

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

a,b,c delimiterを""すると、はabcます。

List of charからの

```
string delimiter = "|";
```

```
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

```
a|b|c
```

の**List of Strings**からの

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

```
Ram is a boy
```

の**array of strings**

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

```
Ram_is_a_boy
```

ToStringをした

、フォ―マットので`String.Format`メソッドをしていますが、`.ToString`は、のタイプをにするためにされます。がわれているにToStringメソッドとにフォ―マットをすることができるので、のフォ―マットをけることができます。さまざまなタイプでどのようにするかをしましょう。

からきへ

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be   "Input value
is 10"
```

フォ―マットされたに

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00");    // output 10
string customFormat = doubleValue.ToString("Input value is 0.0");  // Input value is 10.5
```

ToStringをしたDateTimeの

```
DateTime currentDate = DateTime.Now; //  {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

のからxをする

Visual Basicには、の、、およびからのをするLeft、Right、およびMidがあります。これらのメソッドはCではしませんが、 `Substring()` できます。これらは、のようなメソッドとしてできます。

```
public static class StringExtensions
{
   /// <summary>
   /// VB Left function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Left-most numchars characters</returns>
   public static string Left( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( 0, numchars );
   }

   /// <summary>
   /// VB Right function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Right-most numchars characters</returns>
   public static string Right( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( stringparam.Length - numchars );
   }

   /// <summary>
   /// VB Mid function - to end of string
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
   /// <returns>Balance of string beginning at startindex character</returns>
   public static string Mid( this string stringparam, int startindex )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative startindex being passed
      startindex = Math.Abs( startindex );
```

```
         // Validate numchars parameter
         if (startindex > stringparam.Length)
            startindex = stringparam.Length;

         // C# strings are zero-based, convert passed startindex
         return stringparam.Substring( startindex - 1 );
      }

      /// <summary>
      /// VB Mid function - for number of characters
      /// </summary>
      /// <param name="stringparam"></param>
      /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
      /// <param name="numchars">number of characters to return</param>
      /// <returns>Balance of string beginning at startindex character</returns>
      public static string Mid( this string stringparam, int startindex, int numchars)
      {
         // Handle possible Null or numeric stringparam being passed
         stringparam += string.Empty;

         // Handle possible negative startindex being passed
         startindex = Math.Abs( startindex );

         // Handle possible negative numchars being passed
         numchars = Math.Abs( numchars );

         // Validate numchars parameter
         if (startindex > stringparam.Length)
            startindex = stringparam.Length;

         // C# strings are zero-based, convert passed startindex
         return stringparam.Substring( startindex - 1, numchars );

       }
    }
```

このメソッドは、のようにできます。

```
string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);  // "World!"
string myLeftString = myLongString.Left(5);    // "Hello"
string myMidString1 = myLongString.Left(4);    // "lo World"
string myMidString2 = myLongString.Left(2,3);    // "ell"
```

String.IsNullOrEmptyおよび**String.IsNullOrWhiteSpace**をしての**String**をチェックしています。

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;
```

```
result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

のインデックスでののとの

Substringストリングメソッドをして、されたのからののをできます。ただし、を1つしかとしないは、インデクサをして、のようにのインデックスで1をできます。

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

stringをすSubstringメソッドとはなり、りのはcharであることにしてください。

ィンデクサをして、のをりしすることもできます。

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
/********* This will print each character on a new line:
h
e
l
l
o
**********/
```

10を2、8、16のにする

1. 10をバイナリにするには、 **2**をします。

   ```
   Int32 Number = 15;
   Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
   ```

2. 10を8にするには、 **8**をします。

   ```
   int Number = 15;
   Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
   ```

3. 10を16にするには、 **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

のでをする

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

["this"、 "is"、 "a"、 "complete"、 "sentence"]

をしくさせる

々がをにしなければならないほとんどの、かれなかれのようになります。

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

しかし、これらの々がしていないことは、これがにはっているということです。
そして、はヌルチェックがないためにしません。

Glyph / GraphemeClusterはいくつかのコードポイントでできるため、にはっています。

これがなぜそうであるかをるためには、に「」というがにするものをしなければなりません。

> キャラクターは、くのことをすることができるのです。
>
> コードポイントは、のです。テキストはコードポイントのシーケンスです。コードポイントは、Unicodeによってけられたです。
>
> は、がののとしてするのグラフィカルとしてされる1つのコードポイントのシーケンスです。えば、aとäはともにグラフェンですが、のコードポイントでされていてもかまいませんえば、äは2つのコードポイント、1つはキャラクタa、1つはダイアリシスですが、もう1つのレガシーなシングルコードこのをす。いくつかのコードポイントはしていかなるのでもありませんえば、ゼロの、またはのき。
>
> グリフはイメージであり、はフォントグリフのにされ、グラフェンまたはそのをすためにされます。フォントは、のグリフを1つのにすることができます。たとえば、のäがのコードポイントである、フォントは2つの々のにオーバーレイされたグリフとしてレンダリングすることをできます。 OTFの、フォントのGSUBテーブルとGPOSテーブルには、このをうためのとがまれています。フォントには、じののグリフもまれるがあります。

Cでは、はにはCodePointです。

つまり、もしあなたが`Les Misé rables`ようななをにすると、このようにえる

```
string s = "Les Mise\u0301rables";
```

のとして、のようになります。

セルセクシ―

ごのとおり、アクセントはeではなくRになります。
のにcharをにすると、string.reverse.reverseはのをしますが、こののはののではありません。

GraphemeClusterだけをにするがあります。
したがって、しくされると、のようなをにします。

```
    private static System.Collections.Generic.List<string> GraphemeClusters(string s)
    {
        System.Collections.Generic.List<string> ls = new
System.Collections.Generic.List<string>();

        System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
        while (enumerator.MoveNext())
        {
            ls.Add((string)enumerator.Current);
        }

        return ls;
    }


    // this
    private static string ReverseGraphemeClusters(string s)
    {
        if(string.IsNullOrEmpty(s) || s.Length == 1)
             return s;

        System.Collections.Generic.List<string> ls = GraphemeClusters(s);
        ls.Reverse();

        return string.Join("", ls.ToArray());
    }

    public static void TestMe()
    {
        string s = "Les Mise\u0301rables";
        // s = "noël";
        string r = ReverseGraphemeClusters(s);

        // This would be wrong:
        // char[] a = s.ToCharArray();
        // System.Array.Reverse(a);
        // string r = new string(a);

        System.Console.WriteLine(r);
    }
```

そして、あなたがしくこれをやれば、アジア/アジア/アジアのそしてフランス/スウェーデン/ノルウェーなどでもうまくいくとわかるでしょう。

のをする

System.String.Replaceメソッドをすると、のをのできえることができます。

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

すべてのがされます。

このメソッドは、 String.Emptyフィールドをして、のをするためにもできます。

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

ののIの

System.Stringクラスは、のとのでするいくつかのメソッドをサポートしています。

- System.String.ToLowerInvariantは、Stringオブジェクトをにしてし System.String.ToLowerInvariant。
- System.String.ToUpperInvariantは、にされたStringオブジェクトをすためにされます。

これらのメソッドのバージョンをするは、しないカルチャののをぐためです。これについてはここでしくします 。

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

あなたがのするかをできることにしてくださいをしてとにするString.ToLowerのCultureInfoをとString.ToUpperのCultureInfoにじた。

のをのにする

System.String.Joinメソッドをすると、にされたセパレータをして、のすべてのをでき System.String.Join。

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

は、 System.String.Concatメソッドをするか、 +をしてもっとにうことができます。

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

C6では、これはのようにうことができます

```
string concat = $"{first},{second}";
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/73/

122:

をすると、びしがスタックにされるため、コードになをえることにしてください。コールがすぎると、 **StackOverflow**がするがあります。ほとんどの「な」のようにすることができる`for`、`while`か`foreach`ループ、およびその**POSH**やなていないで、よりになります。

にえ、をにする - なぜそれをするかをる

- びしのがでないことがわかっているは、をするがあります
  - なは、なメモリ
- はコードバージョンがでクリーナーなのでされます。またはループベースのよりもみやすいです。しばしばこれは、よりクリーンでよりコンパクトなコードコードがないをするためです。
  - しかし、がするがあることにしてください。たとえば、フィボナッチでは、シーケンスの*n*のをするために、がにします。

よりくのがなら、んでください

- https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

Examples

オブジェクトをにする

は、メソッドがそれをびすときです。のがたされてからメソッドをにし、メソッドがびされたポイントにるまですることをおめします。そうでない、びしがすぎるためにスタックオーバーフローがするがあります。

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
    public string Name { get; set; }
    public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
```

```
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}
```

なによる

はのようにできます。

　のがたされるまでをびすメソッド。

れたなのは、えられたのをるです

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

このメソッドでは、メソッドが`number`をとることがわかります。

ステップバイステップ

このでは、 `Factorial(4)`すると、

1. `number (4) == 1`ですか
2. いいえ return `4 * Factorial(number-1)` 3
3. このメソッドはもうびされるため、しいとして`Factorial(3)`をしてのをりします。
4. これは`Factorial(1)`がされ、 `number (1) == 1`が1をすまできます。
5.

に、このは`4 * 3 * 2 * 1`をビルドアップし、に24をします。

ののは、メソッドがそれのしいインスタンスをびすことです。り、びしインスタンスのがされます。

をしてディレクトリツリーをする

のの1つは、ファイルシステムのディレクトリツリーのようなデータを、ツリーのレベルやレベルのオブジェクトをらなくてもナビゲートすることです。このでは、ディレクトリツリーのをして、されたディレクトリのすべてのサブディレクトリをし、ツリーをコンソールにするをします。

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
            if (!Directory.Exists(rootDirectoryPath))
            {
                throw new DirectoryNotFoundException(
                    $"Directory '{rootDirectoryPath}' not found.");
            }

            var rootDirectory = new DirectoryInfo(rootDirectoryPath);
            PrintDirectoryTree(rootDirectory, RootLevel);
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
    }

    private static void PrintDirectoryTree(
        DirectoryInfo directory, int currentLevel)
    {
        var indentation = string.Empty;
        for (var i = RootLevel; i < currentLevel; i++)
        {
            indentation += Tab;
        }
```

```
        Console.WriteLine($"{indentation}-{directory.Name}");
        var nextLevel = currentLevel + 1;
        try
        {
            foreach (var subDirectory in directory.GetDirectories())
            {
                PrintDirectoryTree(subDirectory, nextLevel);
            }
        }
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine($"{indentation}-{e.Message}");
        }
    }
}
```

このコードは、ディレクトリをするのをするためのチェックがまれているため、このをするためにはです。では、コードをそれぞれのとともにさなセグメントにしています。

`Main`

mainメソッドは、ルートディレクトリへのパスとしてされるとしてユーザからのをけります。に、このをパラメータとしてして`PrintDirectoryTree`メソッドをびします。

`PrintDirectoryTree(string)`

これはのディレクトリツリーのをするの2つのです。このメソッドは、ルートディレクトリへのパスをすをパラメータとしてります。パスがのディレクトリかどうかをチェックし、そうでないは`DirectoryNotFoundException`をスローし、catchブロックでされます。パスがディレクトリのは、パスから`DirectoryInfo`オブジェクトの`rootDirectory`がされ、2の`PrintDirectoryTree`メソッドが`rootDirectory`オブジェクトで`RootLevel` 、 `RootLevel` がゼロのがびされます。

`PrintDirectoryTree(DirectoryInfo, int)`

この2のは、のをいます。パラメータとして`DirectoryInfo`とをとります。 `DirectoryInfo`はのディレクトリで、はルートとのなディレクトリのさです。みやすいように、はのディレクトリのさごとにインデントされているので、はのようになります。

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

のディレクトリがされると、そのサブディレクトリがされ、このメソッドはのディレクトリのさレベルのよりも1つく、びされます。そのはですメソッドをびします。ツリーのすべてのディレクトリにアクセスするまで、このでプログラムがされます。サブディレクトリのないディレクトリにすると、メソッドはにります。

このメソッドは`UnauthorizedAccessException`もキャッチします`UnauthorizedAccessException`は、のデ

ィレクトリのサブディレクトリのいずれかがシステムによってされているにスロ―されます。エラ―メッセ―ジは、のためにのインデントレベルでされます。

のは、このにするよりなアプロ―チをします。

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

これには、のアプロ―チののエラ―チェックやはまれませんが、じことがわれます。`DirectoryInfo`ではなくのみをするため、アクセスなどののディレクトリプロパティへのアクセスをすることはできません。

フィボナッチシ―ケンス

をして、フィボナッチのをすることができます。

のi> 0について、Fn= Fn-2+ Fn-1のにいて、

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

の、えば9とされますは、そのと1ののです。したがって、えば、9 = 9×8 = 9×8×7 = 9×8×7×6×5×4×3×2×1である。

ですから、をするコ―ドでは、のようになります。

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

PowerOf

えられたのをすることは、にうこともできます。をえるとnとe、々はさせることにより、チャンクでをすることをするがe。

な

- 2 2 = 2×2
- 2 3 = 2×2×2、または2 3 = 2 2×2
  ここには、アルゴリズムのがありますのコードを。これは、をえ、それをよりさく、よりにけてチャンクをくことです。
- ノート
  - が0の、0を0としてすことにするがあります。0 = 0 x 0 x 0
  - が0の、これはであるため、に1をすようにするがあります。

コード

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,
aka: 2³ = 2 * 2² etc..
}
```

xUnitでロジックをするためのテスト
これはずしもではありませんが、ロジックをするためのテストをくことはにいことです。 xUnitフレームワークにかれているものをここにめます。

```
    [Theory]
    [MemberData(nameof(PowerOfTestData))]
    public void PowerOfTest(int @base, int exponent, int expected) {
        Assert.Equal(expected, CalcPowerOf(@base, exponent));
    }

    public static IEnumerable<object[]> PowerOfTestData() {
        yield return new object[] { 0, 0, 0 };
        yield return new object[] { 0, 1, 0 };
```

```
        yield return new object[] { 2, 0, 1 };
        yield return new object[] { 2, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 2, 25 };
        yield return new object[] { 5, 3, 125 };
        yield return new object[] { 5, 4, 625 };
}
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/2470/

123:

き

は、byte、sbyte、short、ushort、int、uint、long、ulongのいずれかのからすることができます。デフォルトはintで、enumでをすることでできます

Weekdaybyte {Monday = 1、Tuesday = 2、Wednesday = 3、Thursday = 4、Friday = 5}

これは、P /ネイティブコードへのびし、データソースへのマッピング、およびのでです。に、デフォルトのintをうべきです、なぜなら、ほとんどのはenumがintであるとしているからです。

- の{、、} // Enum
- enumbyte {、、} //のの
- の{= 23、= 45、= 12} //されたをつ
- Colors.Red // Enumのにアクセスする
- int value =intColors.Red // enumのintをする
- Colors color =ColorsintValue // intからenumをする

Enumのは、のでされるきのセットからなるです。

は、はさななをつをするのにもです。たとえば、ややをすのにできます。これらは、ビットのをしてまたはチェックできるフラグとしてもできます。

Examples

のすべてのメンバーをする

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

これはされます

```
One
Two
Three
```

フラグとしての

`FlagsAttribute`をにして、のにわせて`ToString()`をすることができます。

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

`FlagsAttribute`はが2のまたはそのみわせにし、はにであるため、となるのサイズによってされます。できるのは`UInt64`、64ののされていないフラグをできます。`enum`キーワードのデフォルトは、になる`int Int32`です。コンパイラは、32ビットよりいのをします。それらはなしにラップアラウンドし、じの2つのメンバーになります。したがって、enumが32のフラグのビットセットにするは、よりきなをにするがあります。

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

フラグはしばしばのビットですが、よりいやすくするためにきの"セット"にすることができます。

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

2のの10のスペルをけるために、 シフト<<をしてじをすることもできます

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,
```

```
    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

C7.0、バイナリリテラルもできます。

enumのにのフラグがされているかどうかをするには、 `HasFlag`メソッドをできます。たちがっているとしましょう

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

そして`value`

```
var value = MyEnum.One | MyEnum.Two;
```

`HasFlag`をすると、フラグがされているかどうかをできます

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
    Console.WriteLine("Enum has Two");

if(value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

また、enumのすべてのをりして、されているすべてのフラグをすることもできます

```
var type = typeof(MyEnum);
var names = Enum.GetNames(type);

foreach (var name in names)
{
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

または

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
```

```
    }
}
```

3つのすべてがされます

```
Enum has One
Enum has Two
```

ビットのをつフラグの**enum**をテストする

flags-styleのは、のとしないがあるため、ビットのでテストするがあります。

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

`Default`は、には、ビットのORでマージされた2つのみわせです。したがって、フラグのをテストするには、ビットのANDをするがあります。

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

との

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}


// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"
```

```
// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET
versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive
match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)


// String to enum (all .NET versions)
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Thows
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

enumのデフォルト== ZERO

のデフォルトはゼロです。でが0のがされていない、そのデフォルトはゼロになります。

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
        else
            Console.WriteLine("Unknown");
    }
}
```

https : //dotnetfiddle.net/l5Rwie

の

MSDNから

　　またはは、にりてることができるきのセットをにするをします 。

に、はオプションのセットのみをするタイプであり、オプションはにします。デフォルトでは、

がゼロからまるにしています。たとえば、のをすることができます。

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

そのはのようにできます

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

デフォルトでは`enum`のは`int`ですが、 `byte` 、 `sbyte` 、 `short` 、 `ushort` 、 `uint` 、 `long` 、および`ulong`もにできます。 `int`のをするは、ののにコロンをしてをするがあります。

```
public enum Day : byte
{
    // same as before
}
```

ののは、ではなくバイトです。のとなるをのようにできます。

```
Enum.GetUnderlyingType(typeof(Days));
```

```
System.Byte
```

デモ .NET fiddle

をいたビット

FlagsAttributeは、がのではなくフラグのコレクションをすときはにするがあります。enumにりてられたは、ビットをしてenumをするときにちます。

1[フラグ**]**

```
[Flags]
enum Colors
```

```
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

、をします。

2[フラグ**]**がない

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

プリント3

フラグに<<をする

シフト `<<` はフラグenumですることができ、フラグのフラグはずバイナリで`1`になります。

これはまた、ののフラグがたくさんあるきなのをさせるのにちます。

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

`MyEnum`はなフラグだけがまれており、な`Flag30 = 1073741822` または 11111111111111111111111111111110バイナリのようななものはまれていないことはらかです。

enumにのをする

によっては、enumにユーザーにしたいものよりみにくいなど、のをenumにすることができます。そのようなは、 `System.ComponentModel.DescriptionAttribute`クラスをできます。

えば

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

のenumのをすは、のようにします。

```
public static string GetDescriptionAttribute(PossibleResults result)
{
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}
```

これは、すべてののメソッドににすることもできます。

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())
typeof(DescriptionAttribute))).Description;
    }
}
```

そして、のようににします `Console.WriteLine(result.GetDescription());`

フラグきのをおよびする

このコーードは、フラグをけたenum-instanceからをしたりしたりするためのものです。

```
[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;
```

```
// set additional value
value |= MyEnum.Flag2;  //value is now Flag1, Flag2
value |= MyEnum.Flag3;  //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

はしないをつがあります

はなとのでキャストすることができるため、はのでされたのになることがあります。

ののDaysOfWeekは7つのみのしかありDaysOfWeekんが、それでもintをできます。

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOFWeek s = DaysOfWeek.Sunday;
s++; // No error
```

のところ、このをたないをするはありません。

ただし、 Enum.IsDefinedメソッドをすると、のenumをできます。えば、

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/931/

124: なデザインパターン

なパターンは、システムを、そのオブジェクトがどのようにされ、され、されているかからすることをしています。それらは、オブジェクトのの、、、について、システムのをさせます。パターンは、システムがするクラスにするをカプセルしますが、これらのクラスのインスタンスがどのようにされ、まとめられているかのをします。プログラマーは、をつシステムをすることは、それらのシステムをあまりにもさせることにくようになった。なパターンは、このなをるようにされています。

Examples

シングルトンパターン

シングルトンパターンは、クラスのをに1つのインスタンスにするようにされています。

このパターンは、のようなものが1つしかないのがにかなっているシナリオでされます。

- のオブジェクトのをするのクラス、。マネージャークラス
- のリソースをす1つのクラス、えば、ロギングコンポーネント

Singletonパターンをするもなの1つは、 `CreateInstance()`や`GetInstance()` またはC、 `Instance`プロパティ`GetInstance()`などのファクトリメソッドをし、にじインスタンスをすようにされています。

メソッドまたはプロパティへののびしは、Singletonインスタンスをしてします。その、メソッドはにじインスタンスをします。このようにして、シングルトンオブジェクトのインスタンスは1つしかありません。

クラスコンストラクタを`private`.することで、 `new`をしたインスタンスのをぐことができます `private`.

Cでシングルトンパターンをするためのなコードをにします。

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
    {
    }

    public static Singleton Instance
```

```
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

このパターンをさらにするために、のコードでは、Instanceプロパティがびされたときに、Singletonのインスタンスがされるかどうかをします。

```
class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}
```

このはスレッドセーフではありません。

このスレッドセーフなをむ、よりくのをるには、をごください Singleton Implementation

シングルトンははにはグローバルにしており、ののやをきこします。このため、シングルトンパターンはくパターンとなされます。

「シングルトンについてがいの」にするのについては、こちらをしてください。

Cでは、クラスを`static`にすることができます。これにより、すべてのメンバーがになり、クラスをインスタンスできなくなります。これをえると、シングルトンパターンのわりにされるクラスをるのがです。

のないについては、「CSingleton Pattern vs. Static Class」をしてください。

ファクトリメソッドは、なデザインパターンの1つです。これは、なタイプをせずにオブジェクトをするというにするためにされます。このドキュメントでは、ファクトリメソッドDPをしくするについてします。

あなたになでそれのアイデアをしましょう。、、の3のデバイスをするでいているとします。あなたは、されたデバイスをするコンピュータのプログラムをしていますが、あなたはをするかについてのをらないのです。

すべてのデバイスにのをつインターフェース`IDevice`をしましょう。

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

さて、たちのデバイスをすクラスをすることができます。それらのクラスは`IDevice`インタフェースをするがあります

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
{
    private Random r = null;
    public VoltMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}
```

ここでファクトリメソッドをするがあります。メソッドをにつ`DeviceFactory`クラスをしましょう

```
public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}
```

すばらしいですたちのコードをテストしましょう

```
public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.VOLT);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.OHM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();
    }
}
```

これは、このコードをしたにされるのです。

AmMeterがオンになります...

36

6

33

43

24

AmMeterは、さようならとってをさせます

VoltMeterがオンになります...

102

-61

85

138

36

VoltMeterがライトをさせ、さよならをっている

OhmMeterがオンになります...

723828

368536

685412

800266

578595

OhmMeterは、さようならをっているをさせる

ビル ダーパターン

なオブジェクトのをそのからして、じプロセスでなるをし、オブジェクトのアセンブリをいレベルでできるようにします。

このでは、さまざまながにみてられたBuilderパターンをしています。ショップでは、VehicleBuildersをしてのステップでさまざまなビークルをしています。

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

namespace GangOfFour.Builder
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      VehicleBuilder builder;

      // Create shop with vehicle builders
      Shop shop = new Shop();

      // Construct and display vehicles
      builder = new ScooterBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new CarBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new MotorCycleBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
  {
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
      vehicleBuilder.BuildFrame();
      vehicleBuilder.BuildEngine();
      vehicleBuilder.BuildWheels();
      vehicleBuilder.BuildDoors();
    }
  }

  /// <summary>
  /// The 'Builder' abstract class
  /// </summary>
  abstract class VehicleBuilder
  {
    protected Vehicle vehicle;
```

```
  // Gets vehicle instance
  public Vehicle Vehicle
  {
    get { return vehicle; }
  }

  // Abstract build methods
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
  public MotorCycleBuilder()
  {
    vehicle = new Vehicle("MotorCycle");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "MotorCycle Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}


/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
  public CarBuilder()
  {
    vehicle = new Vehicle("Car");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Car Frame";
  }

  public override void BuildEngine()
```

```
    {
      vehicle["engine"] = "2500 cc";
    }

    public override void BuildWheels()
    {
      vehicle["wheels"] = "4";
    }

    public override void BuildDoors()
    {
      vehicle["doors"] = "4";
    }
  }

  /// <summary>
  /// The 'ConcreteBuilder3' class
  /// </summary>
  class ScooterBuilder : VehicleBuilder
  {
    public ScooterBuilder()
    {
      vehicle = new Vehicle("Scooter");
    }

    public override void BuildFrame()
    {
      vehicle["frame"] = "Scooter Frame";
    }

    public override void BuildEngine()
    {
      vehicle["engine"] = "50 cc";
    }

    public override void BuildWheels()
    {
      vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
      vehicle["doors"] = "0";
    }
  }

  /// <summary>
  /// The 'Product' class
  /// </summary>
  class Vehicle
  {
    private string _vehicleType;
    private Dictionary<string,string> _parts =
      new Dictionary<string,string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
      this._vehicleType = vehicleType;
    }
```

```
    // Indexer
    public string this[string key]
    {
      get { return _parts[key]; }
      set { _parts[key] = value; }
    }

    public void Show()
    {
      Console.WriteLine("\n---------------------------");
      Console.WriteLine("Vehicle Type: {0}", _vehicleType);
      Console.WriteLine(" Frame : {0}", _parts["frame"]);
      Console.WriteLine(" Engine : {0}", _parts["engine"]);
      Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
      Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
  }
}
```

のタイプスクーターフレームスクーターフレーム
エンジンなし
ホイール2
#Doors0

フレームのフレーム
エンジン2500 cc
ホイール4
#Doors4

のMotorCycle
フレームモーターサイクルフレーム
エンジン500 cc
ホイール2
#Doors0

プロトタイプパターン

プロトタイプのインスタンスをしてするオブジェクトのをし、このプロトタイプをコピーしてしいオブジェクトをします。

このでは、じタイプののユーザーのをコピーしてしいColorオブジェクトをするプロトタイプパターンをします。

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

namespace GangOfFour.Prototype
```

```
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Prototype Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ColorManager colormanager = new ColorManager();

      // Initialize with standard colors
      colormanager["red"] = new Color(255, 0, 0);
      colormanager["green"] = new Color(0, 255, 0);
      colormanager["blue"] = new Color(0, 0, 255);

      // User adds personalized colors
      colormanager["angry"] = new Color(255, 54, 0);
      colormanager["peace"] = new Color(128, 211, 128);
      colormanager["flame"] = new Color(211, 34, 20);

      // User clones selected colors
      Color color1 = colormanager["red"].Clone() as Color;
      Color color2 = colormanager["peace"].Clone() as Color;
      Color color3 = colormanager["flame"].Clone() as Color;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Prototype' abstract class
  /// </summary>
  abstract class ColorPrototype
  {
    public abstract ColorPrototype Clone();
  }

  /// <summary>
  /// The 'ConcretePrototype' class
  /// </summary>
  class Color : ColorPrototype
  {
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
      this._red = red;
      this._green = green;
      this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
```

```
    {
      Console.WriteLine(
        "Cloning color RGB: {0,3},{1,3},{2,3}",
        _red, _green, _blue);

      return this.MemberwiseClone() as ColorPrototype;
    }
  }

  /// <summary>
  /// Prototype manager
  /// </summary>
  class ColorManager
  {
    private Dictionary<string, ColorPrototype> _colors =
      new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
      get { return _colors[key]; }
      set { _colors.Add(key, value); }
    }
  }
}
```

クローニングRGB255,0,0

クローニングカラーRGB128,211,128

クローニングRGB211,34,20

なパターン

なクラスをせずに、オブジェクトまたはオブジェクトのファミリをするためのインタフェースをします。

このでは、なるをするコンピュータゲームのためのなるののをす。によってられたはなるが、のはじままである。

```
using System;

namespace GangOfFour.AbstractFactory
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
```

```
    ContinentFactory africa = new AfricaFactory();
    AnimalWorld world = new AnimalWorld(africa);
    world.RunFoodChain();

    // Create and run the American animal world
    ContinentFactory america = new AmericaFactory();
    world = new AnimalWorld(america);
    world.RunFoodChain();

    // Wait for user input
    Console.ReadKey();
  }
}


/// <summary>
/// The 'AbstractFactory' abstract class
/// </summary>
abstract class ContinentFactory
{
  public abstract Herbivore CreateHerbivore();
  public abstract Carnivore CreateCarnivore();
}

/// <summary>
/// The 'ConcreteFactory1' class
/// </summary>
class AfricaFactory : ContinentFactory
{
  public override Herbivore CreateHerbivore()
  {
    return new Wildebeest();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Lion();
  }
}

/// <summary>
/// The 'ConcreteFactory2' class
/// </summary>
class AmericaFactory : ContinentFactory
{
  public override Herbivore CreateHerbivore()
  {
    return new Bison();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}
```

```
/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Bison
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'Client' class
/// </summary>
class AnimalWorld
{
  private Herbivore _herbivore;
  private Carnivore _carnivore;

  // Constructor
  public AnimalWorld(ContinentFactory factory)
  {
    _carnivore = factory.CreateCarnivore();
    _herbivore = factory.CreateHerbivore();
```

```
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

ライオンはワイルドビーストをべる

オオカミはバイソンをべる

オンラインでなデザインパターンをむ https://riptutorial.com/ja/csharp/topic/6654/なデザインパターン

125: の

Examples

の

はがけられないので、これらののはにけされなければなりません `var`。

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

メンバがされていないは、オブジェクトをするためにされるプロパティ/のにされます。

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

のがなプロパティアクセスのにのみ、をすることができます。メソッドびしやなのは、プロパティをするがあります。

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

と

では、チェックをしながら、をににすることなくオブジェクトをすることができます。

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

に、 `dynamic`にはコンパイルエラーのわりにエラーをするチェックがあります。

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

をつなメソッド

なメソッドでは、のをじてをできます。

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

これは、LINQをでできることをします。

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

をつジェネリックのインスタンス

ジェネリックコンストラクタをするには、のをけるがありますが、これはです。あるいは、なをしてをうことができる。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

`List<T>`の、にされたは、 `ToList` LINQメソッドをして`List<T>`できます。

```
var list2 = new[] {anon, anon2}.ToList();
```

の

のは、 `Equals`インスタンスメソッドによってえられます。 2つのオブジェクトは、すべてのプロパティにしてじとしい `a.Prop.Equals(b.Prop)` をつ、しいです。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

2つのは、プロパティのとがじで、じでされるにり、じとみなされます。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
```

```
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have diferent types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

にされた

のは、のですることができます。

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

オンラインでのをむ https://riptutorial.com/ja/csharp/topic/765/の

126:

き

リフレクションは、にオブジェクトのプロパティにアクセスするためのCメカニズムです。、リフレクションは、オブジェクトタイプおよびオブジェクトにするをするためにされます。たとえば、RESTアプリケーションでは、リフレクションをして、シリアライズされたレスポンス·オブジェクトをすることができます。

MSのガイドラインによれば、パフォーマンスのなコードはされるべきではありません。https://msdn.microsoft.com/en-us/library/ff647790.aspxをしてください。

リフレクションをすると、にアセンブリ、モジュール、およびタイプにするにコードがアクセスできるようになりますプログラム。これをさらにして、タイプをに、、またはアクセスすることができます。タイプには、プロパティ、メソッド、フィールド、およびがまれます。

C

.NET Frameworkのリフレクション

Examples

System.Typeをする

のインスタンスの

```
var theString = "hello";
var theType = theString.GetType();
```

から

```
var theType = typeof(string);
```

のメンバーをする

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

public class Program
{
  public static void Main()
  {
    var members = typeof(object)
                  .GetMembers(BindingFlags.Public |
                              BindingFlags.Static |
```

```
                                BindingFlags.Instance);

    foreach (var member in members)
    {
      bool inherited = member.DeclaringType.Equals( typeof(object).Name );
      Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                        $"it has {(inherited ? "":"not")} been inherited.");
    }
  }
}
```

アウトプットアウトプットオ―ダについてのをさらにてください

```
GetType is a Method, it has not been inherited.
GetHashCode is a Method, it has not been inherited.
ToString is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
ReferenceEquals is a Method, it has not been inherited.
.ctor is a Constructor, it has not been inherited.
```

また、 `BindingFlags`をさずに`GetMembers()`をすることもできます。これは、そののタイプのすべてのパブリックメンバ―をします 。

`GetMembers`がのでメンバ―をさないので、 `GetMembers`されるにはしてしないことにしてください。

デモをる

メソッドをしてびす

インスタンスメソッドをしてびす

```
using System;

public class Program
{
    public static void Main()
    {
        var theString = "hello";
        var method = theString
                     .GetType()
                     .GetMethod("Substring",
                                new[] {typeof(int), typeof(int)}); //The types of the method
arguments
         var result = method.Invoke(theString, new object[] {0, 4});
         Console.WriteLine(result);
    }
}
```

デモをる

メソッドをしてびす

、メソッドがである、インスタンスをびすはありません。

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2});//Pass null as the first argument (no need
for an instance)
Console.WriteLine(result); //You'll get e^2
```

7.38905609893065

デモをる

プロパティのと

な

```
PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.setValue(myInstance, newValue);
```

みりのプロパティをするには、そのバッキングフィールドをしますバッキングフィールドの.NET Frameworkは "k__BackingField"です。

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

カスタム

カスタム - `MyAttribute` プロパティをする

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

のプロパティのすべてのカスタムをする

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

カスタムをつすべてのクラスをする - `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
```

```
    }
}
```

にカスタムのをみる

```
public static class AttributeExtensions
{

        /// <summary>
        /// Returns the value of a member attribute for any member in a class.
        ///     (a member is a Field, Property, Method, etc...)
        /// <remarks>
        /// If there is more than one member of the same name in the class, it will return the
first one (this applies to overloaded methods)
        /// </remarks>
        /// <example>
        /// Read System.ComponentModel Description Attribute from method 'MyMethodName' in
class 'MyClass':
        ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName",
(DescriptionAttribute d) => d.Description);
        /// </example>
        /// <param name="type">The class that contains the member as a type</param>
        /// <param name="MemberName">Name of the member in the class</param>
        /// <param name="valueSelector">Attribute type and property to get (will return first
instance if there are multiple attributes of the same type)</param>
        /// <param name="inherit">true to search this member's inheritance chain to find the
attributes; otherwise, false. This parameter is ignored for properties and events</param>
        /// </summary>
        public static TValue GetAttribute<TAttribute, TValue>(this Type type, string
MemberName, Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute :
Attribute
        {
            var att =
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),
inherit).FirstOrDefault() as TAttribute;
            if (att != null)
            {
                return valueSelector(att);
            }
            return default(TValue);
        }
    }
```

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

クラスのすべてのプロパティをループする

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
```

```
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

ジェネリックのインスタンスのの

ジェネリックのインスタンスがありますが、なんらかののでのがわからないは、このインスタンスのにされたのをべることをおめします。

たとえば、かが`List<T>`インスタンスをし、それをメソッドにすとしましょう

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

`ShowGenericArguments`はこのシグネチャがあります。

```
public void ShowGenericArguments(object o)
```

コンパイルに、。をするためになながわれているかどうかはわかりません。 Reflectionは、ジェネリックをするためのくのメソッドをします。に、。のタイプがタイプであるかどうかをできます。

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType`し`true`がジェネリックとされている`false`のではありません。

しかし、これはたちがりたいだけではありません。 `List<>`もです。しかし、のされたジェネリックのインスタンスをべたいだけです。されたジェネリックは、えば、すべてのジェネリックパラメータにしてのをつ`List<int>`です。

`Type`クラスは、これらのされたジェネリックをジェネリックとするために、さらに2つのプロパティ`IsConstructedGenericType`と`IsGenericTypeDefinition`します。

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

インスタンスのをするために、ジェネリックをむ`Type`をす`GetGenericArguments()`メソッドをできます。

```
public void ShowGenericArguments(object o)
```

```
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

したがって、からのびし `ShowGenericArguments(myList)` のはのようになります。

```
Int32
```

ジェネリックメソッドをしてびす

ジェネリックメソッドをつクラスがあるとしましょう。そして、そのをリフレクションでびすがあります。

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

GenericMethodをstringでびすとします。

```
Sample sample = new Sample();//or you can get an instance via reflection

MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null);//Since there are no arguments, we are passing null
```

メソッドの、インスタンスはありません。したがって、のもnullになります。

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

のインスタンスをし、それをびす

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

インタフェースをするクラスをインスタンスするえば、プラグインのアクティブ

アプリケーションがプラグインシステムをサポートするようにする、たとえば`plugins`フォルダにあるアセンブリからプラグインをロードするは、のようにします。

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

このクラスはのdllにあります

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

アプリケーションのプラグインローダーはdllファイルをつけ、 `IPlugin`をしているアセンブリですべてのをし、それらのインスタンスをします。

```
    public IEnumerable<IPlugin> InstantiatePlugins(string directory)
    {
        var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name =>
new FileInfo(name).FullName).ToArray();
        //load the assemblies into the current AppDomain, so we can instantiate the types
later
        foreach (var fileName in pluginAssemblyNames)
            AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
        var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
        var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
        var pluginTypes = typesInAssembly.Where(type => typeof
(IPlugin).IsAssignableFrom(type));
        return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
    }
```

のインスタンスの

もなは、 `Activator`クラスをすることです。

しかし、 `Activator.CreateInstance()`をして、 `Activator`パフォーマンスが.NET 3.5にされたとしても、 テスト1 、 テスト2 、 テスト3の パフォーマンスがいため、

`Activator`クラスをする

```
Type type = typeof(BigInteger);
```

```
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an
'int' compatible argument.
Console.WriteLine(result); //Output: 123
```

のパラメ―タがある、オブジェクトを`Activator.CreateInstance`することができます。

```
// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });

Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

ジェネリックの

`MakeGenericType`メソッドは、いたジェネリック `List<>` をタイプをすることによって `List<string>` にします。

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

`List<>`は、 `typeof`のではされていません。

`Activator` クラスなし

`new`キ―ワ―ドをするパラメ―タのないコンストラクタの

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

びしメソッドの

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

ツリーの

ツリーは、ノードがであるツリーデータのコードをす。 MSDNがするように

> は、1つのオペランドと0ののシーケンスであり、の、オブジェクト、メソッド、またはにされます。は、リテラル、メソッドびし、とそのオペランド、またはなでできます。シンプルなには、、メンバー、メソッドパラメーター、またはをできます。

```
public class GenericFactory<TKey, TType>
    {
       private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; //
dictionary, that holds constructor functions.
       private object _locker = new object(); // object for locking dictionary, to guarantee
thread safety

        public GenericFactory()
        {
            _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
        }

        /// <summary>
        /// Find and register suitable constructor for type
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key">Key for this constructor</param>
        /// <param name="parameters">Parameters</param>
        public void Register(TKey key, params Type[] parameters)
        {
            ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] {
}); // Get the instance of ctor.
            if (ci == null)
                throw new InvalidOperationException(string.Format("Constructor for type '{0}'
was not found.", typeof(TType)));

            Func<object[], TType> ctor;

            lock (_locker)
            {
                if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor
already been registered
                {
                    var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
parameter Expression
                    var ctorParams = ci.GetParameters(); // get parameter info from
constructor

                    var argExpressions = new Expression[ctorParams.Length]; // array that will
contains parameter expessions
                    for (var i = 0; i < parameters.Length; i++)
                    {

                        var indexedAcccess = Expression.ArrayIndex(pExp,
Expression.Constant(i));

                        if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
parameter is a value type
                        {
                            var localVariable = Expression.Variable(parameters[i],
```

```
"localVariable"); // if so - we should create local variable that will store paraameter value

                            var block = Expression.Block(new[] { localVariable },
                                    Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
                                        Expression.Assign(localVariable,
Expression.Default(parameters[i])),
                                        Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
                                    ),
                                    localVariable
                                );

                            argExpressions[i] = block;

                        }
                        else
                            argExpressions[i] = Expression.Convert(indexedAcccess,
parameters[i]);
                    }
                    var newExpr = Expression.New(ci, argExpressions); // create expression
that represents call to specified ctor with the specified arguments.

                    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp
}).Compile() as Func<object[], TType>); // compile expression to create delegate, and add
fucntion to dictionary
                }
            }
        }

        /// <summary>
        /// Returns instance of registered type by key.
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key"></param>
        /// <param name="args"></param>
        /// <returns></returns>
        public TType Create(TKey key, params object[] args)
        {
            Func<object[], TType> foo;
            if (_registeredTypes.TryGetValue(key, out foo))
            {
                return (TType)foo(args);
            }

            throw new ArgumentException("No type registered for this key.");
        }
    }
```

このようにできます

```
public class TestClass
{
        public TestClass(string parameter)
        {
            Console.Write(parameter);
        }
}
```

```
public void TestMethod()
{
       var factory = new GenericFactory<string, TestClass>();
       factory.Register("key", typeof(string));
       TestClass newInstance = factory.Create("key", "testParameter");
}
```

FormatterServices.GetUninitializedObjectの

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

FormatterServices.GetUninitializedObjectコンストラクタをする、フィールドはびされません。シリアライザやリモーティングエンジンでされることをしています

ネームスペースででをする

これをうには、をむアセンブリへのがです。なものとじアセンブリにあることがわかっているのタイプがあるは、これをうことができます

```
typeof(KnownType).Assembly.GetType(typeName);
```

- ここで、 typeNameは、しているタイプのをむですKnownTypeは、じアセンブリにあることがわかっているタイプです。

それほどではありませんが、よりです。

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

のスピードをげるために、システムアセンブリのスキャンをするチェックにしてください。タイプがにCLRタイプのは、これらの2をするがあります。

アセンブリをむにアセンブリでされたがあるは、にそれをることができます

```
Type.GetType(fullyQualifiedName);
```

リフレクションをしてメソッドまたはプロパティにくけされたデリゲートをする

パフォーマンスがされる、リフレクションつまり、 MethodInfo.Invokeメソッドをしてをしてメソッドをびすことはではありません。ただし、 Delegate.CreateDelegateをして、よりのあるくけされたデリゲートをするのはです。リフレクションをするのパフォーマンスのは、デリゲートのプ

ロセスでのみします。デリゲートがされると、それをびすためのパフォーマンスのペナルティはほとんどありません。

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int,
int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

このはプロパティにもできます。 `MyIntProperty`というの`int`プロパティをつ`MyClass`というクラスがある、くけされたgetterをするコードはのようになりますのでは、 'target'は`MyClass`なインスタンスです。

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

...セッターについてもじことができます

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/28/

127: のキーワード

き

ステートメントでyieldキーワードをするは、それがれるメソッド、、またはアクセサーをイテレータとしてします。 yieldをしてイテレータをすると、カスタムコレクションのIEnumerableおよびIEnumeratorパターンをするときに、ななクラスのをするクラスがになります。

- yield return [TYPE]
-

りのが`IEnumerable` 、 `IEnumerable<T>` 、 `IEnumerator` 、または`IEnumerator<T>`メソッドに`yield`キーワードをすると、りの `IEnumerable`または`IEnumerator` のがされます。それぞれのをるためにそれぞれの ""までの

`yield`キーワードは、にのシーケンスの「の」をすにです。にシーケンスをすることはです。また、されるになシーケンスをすると、ユーザーにとってましくないにつながります。

`yield break`をして、いつでもをすることができます。

`yield`キーワードは`IEnumerable<T>`ようなりとしてイテレータのインターフェイスをとするため、これを`Task<IEnumerable<T>>`オブジェクトをすのでメソッドではできません。

- https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

Examples

な

`yield`キーワードは、 `IEnumerable`または`IEnumerator` `IEnumerator`したバリアントをすをするためにされます。このは、びしがされたコレクションをするときににされます。 にのをおみください。

のは、 `for`ループのにあるyield returnステートメントをっています。

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

に、それをびすことができます

```
foreach (int value in Count(start: 4, count: 10))
{
```

```
    Console.WriteLine(value);
}
```

コンソール

```
4
5
6
...
14
```

.NET Fiddleのライブデモ

`foreach`ステートメントのでは、`Count`がびされます。イテレーターのびしは、の`for`ループのりしにする`yield return`ステートメントののにみます。

よりな

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name
FROM Users"))
    {
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

もちろん、SQLデータベースから`IEnumerable<User>`をするのもあります。これは`yield`をして "のシーケンス"セマンティクスをつものをかがりしできる`IEnumerable<T>`できることをしています。

の`yield`メソッドのをするには、ループのをする`yield break`に`yield break`をびして、でをできる1つのまたはをします。

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
```

```
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

のメソッドは、 `earlyTerminationSet`のの1つがされるまで、された`start`からします。

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

1
2
3
4
5
6

.NET Fiddleのライブデモ

をしくチェックする

ィテレータメソッドは、りがされるまでされません。したがって、イテレータのでをすることはです。

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));

    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

コールサイドコード

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

1
2
3
4
5
6
7
8
9
10

.NET Fiddleのライブデモ

メソッドが`yield`をしてをすると、コンパイラはにコードを`yield`までするステートマシンをします。に、したアイテムをし、そのをします。

これは、のをしてアクセスしたときにのみ、メソッドをにびすときにな`null`をすなどがつからないことをします。メソッドはマシンによってされます。をにチェックするのメソッドでラップすることによって、メソッドがびされたときにをチェックすることができます。これはのです。

C7 `CountCore`をする、`CountCore`は、ローカルとしてに`Count`にすことができます。ここのをしてください。

Enumerableをすメソッドのの Enumerableをす

```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

レイジー

`foreach`がのにするときのみ、イテレータブロックはの`yield`までされます。

のをえてみましょう。

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}
```

これはされます

の
イテレータ0
foreach0
イテレータ1
foreach1
イテレータ2
foreach2

デモをる

として

- "Starting iteration"は、にイテレータメソッドがびされたにもかかわらずされます `Integers().Take(3);`これは、 `Integers().Take(3);`にをしません `IEnumerator.MoveNext()`へのびしはわれません
- コンソールにするは、イテレータメソッドのものと`foreach`のものとのでにされます。
- このプログラムは、iteratorメソッドがれない`while true`をっていても、 `.Take()`メソッドのためにします。

して...に

iteratorメソッドが`try...finally`にyieldをつ、された`IEnumerator`は、ののが`try`ブロックにあるり、`Dispose`がびされたとき`finally`ステートメントをします。

えられた

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

するとき

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

に、それはされます

    1

デモをる

するとき

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

に、それはされます

    1
    2
    にされた

デモをる

yieldをしてIEnumeratorをする IEnumerableをするとき

`IEnumerable<T>`インターフェイスは、の、する`GetEnumerator()`をし、 `IEnumerator<T>`。

`yield`キーワードは`IEnumerable<T>`をするためにできますが、まったくじで`IEnumerator<T>`をすることもできます。されるのは、メソッドのりのだけです。

これは、 `IEnumerable<T>`をするのクラスをするにです。

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }

    // This method returns an IEnumerator<T>, rather than an IEnumerable<T>
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
    {
        foreach(var item in _wrapped)
        {
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

このののはなるであり、 `IEnumerable<T>`すのイテレータメソッドをしてよりにできます。

な

`yield`キーワードは、コレクションのをにします。コレクションをにメモリにロードすることを**eager evaluation**とびます。

のコードはこれをしています

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
...
```

```
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

ToList 、 ToDictionaryまたはToArrayをびすと、のがされ、すべてのがコレクションにされます。

レ イジ―のフ ィボナッチ

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numberics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

どのようにそれがフ―ドのでするは、IL Disaamblerツ―ルでの.exeファイルをコンパイルすることをおめします

1. Cコンパイラは、 IEnumerable<BigInteger>とIEnumerator<BigInteger> <Fibonacci>d__0 d__0をするクラスをします。
2. このクラスはマシンをします。 Stateは、メソッドののとロ―カルのからされます。
3. もいコ―ドはbool IEnumerator.MoveNext()メソッドにあります。に、 MoveNext()はをします
    - のをします。 prevやcurrentなどのは、クラスのフィ―ルドになります <current>5__2と<prev>5__1 5__1。 々のでは、2つのポジション <>1__state 1つの<>1__state があります。 はをき、2はyield returnです。
    - のyield returnかyield break / } yield breakまでコードをします。
    - yield returnはがされるので、 Currentプロパティはそれをすことができます。 trueが

されます。このので、のは、のMoveNextびしのためにびされます。
- yield break / }メソッドのは、がわれたfalseするfalseをします。

また、10001のは468バイトであることにもしてください。ステートマシンは、 currentとprevのみをフィールドとしてします。シーケンスのすべてのをから10000までしたい、されるメモリサイズは4メガバイトをえます。したがって、にされると、レイジーは、によってはメモリフットプリントをらすことができます。

とのい

してyield breakするとはにbreak 1がえることほどではないかもしれません。インターネットには2つのいががあり、そのいをしていないいがたくさんあります。

するは、のキーワードまたはキーフレーズがループでのみがあるということです foreach 、 while ...。

あるメソッドでyieldキーワードをうと、メソッドをにイテレータにえることができます 。このようなメソッドののは、またはのコレクションをりしし、そのをすることです。がされると、メソッドのをけるはありません。には、メソッドののじでにします。しかし、には、メソッドをでしたいとします。のしないメソッドでは、 returnキーワードをします。しかし、イテレータでreturnをうことはできません。yield yield breakをするがあります。つまり、イテレータのyield breakは、メソッドのreturnとじです。、 breakステートメントはもいループをさせるだけです。

いくつかのをてみましょう

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
```

```
public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
            break;
        }
    }
    // Execution continues
    yield return 10;
}
```

オンラインでのキーワードをむ https://riptutorial.com/ja/csharp/topic/61/のキーワード

128: き

Examples

きは、コードをよりにすることができます

このなクラスをえてみましょう

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

C3.0よりは

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

きをしてこのメソッドをさらににびすことができます 。

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

きとオプションのパラメータ

きをオプションのパラメータとみわせることができます。

このをてみましょう

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

`retryCount`をせずにこのメソッドをびすは、のようにします。

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

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

のはありません

きは、のでできます。

サンプルメソッド

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

びしサンプル

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

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

きは、オプションのパラメータのバグをける

メソッドがされたときのなバグをけるために、にオプションにきをしてください。

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

    public string Title { get; set; }

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

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

のコードは、コンストラクタがいつかのようにされるまで、コンパイルしてにします

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

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

「チームののかがっている」ときのバグをけるためのベストプラクティス

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

オンラインできをむ https://riptutorial.com/ja/csharp/topic/2076/き

129: きとオプション

き

*RefMSDN*きをすると、パラメータリストのパラメータのではなくパラメータのにをけることによって、のパラメータのをできます。

MSDNによると、きは、

- パラメータのをけることによって、をにすことができます。
- たちがにづいていないパラメータのをえておくはありません。
- びされたのパラメータリストのパラメータのをるはありません。
- のパラメータは、そのでできます。

オプションの

*RefMSDN*メソッド、コンストラクタ、インデクサ、またはデリゲートのでは、そのパラメータがであるか、オプションであるかをできます。すべてのびしは、なすべてのパラメータのをするがありますが、オプションのパラメーターのはできます。

オプションのであるMSDNによると、

- そのがオプションのである、びしのをすることができます
- すべてのオプションにはのデフォルトがあります
- をしないと、デフォルトがされます
- Optional Argumentのデフォルトは、a
  - 。
  - enumやstructなどのでなければなりません。
  - フォームのデフォルトvalueTypeのであるがあります。
- パラメータリストのにするがあります

Examples

き

のことをえてみましょう。

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

この、のはさつまり120で、2のはつまり56です。そして、そのでをしています。はのとおりです。

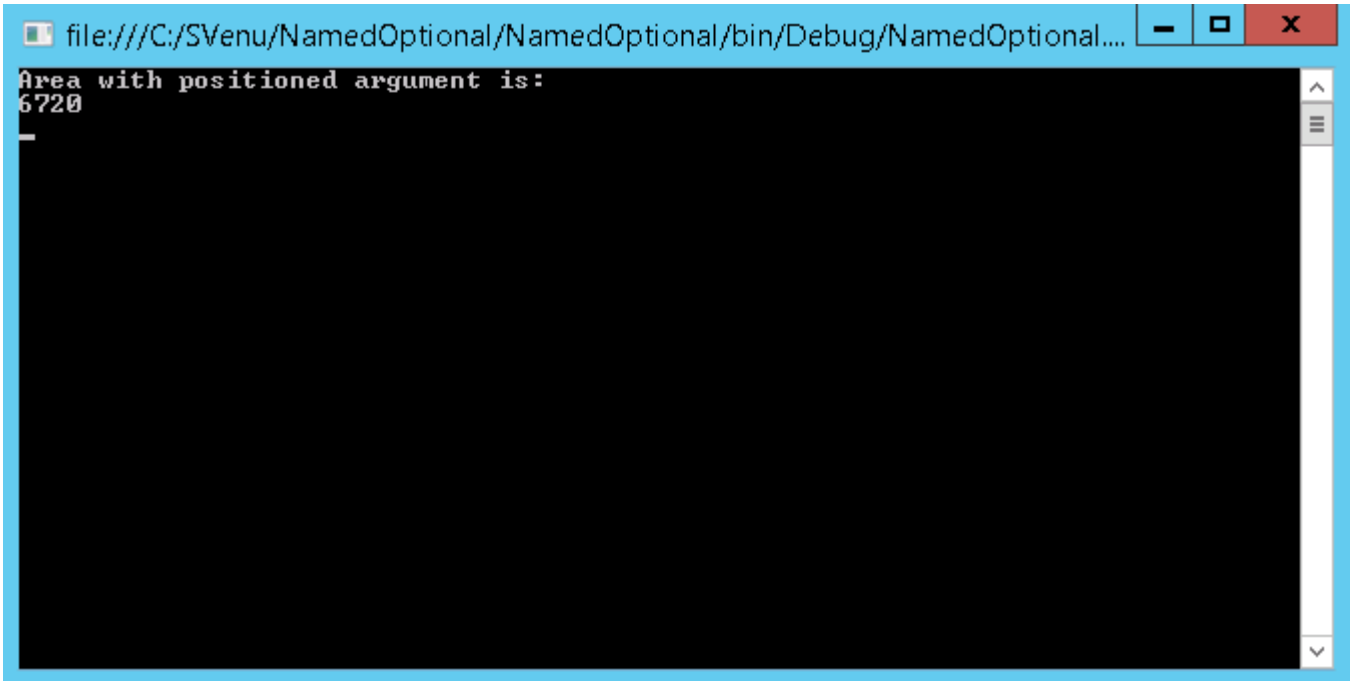
```
private static double FindArea(int length, int width)
       {
```

```
            try
            {
                return (length* width);
            }
            catch (Exception)
            {
                throw new NotImplementedException();
            }
        }
```

したがって、のびしでは、をそのでしました。

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

これをすると、のようながられます。



ここではきのがあります。のびしをしてください。

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

ここでは、メソッドびしできをしています。

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

このプログラムをすると、じがられます。きをしているは、メソッドびしでをすることができます。

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

きのないの1つは、プログラムでこれをすると、コードのがすることです。あなたのがをするのか、それともであるのかをにえます。

のもできます。つまり、ときのをみわせたものです。

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

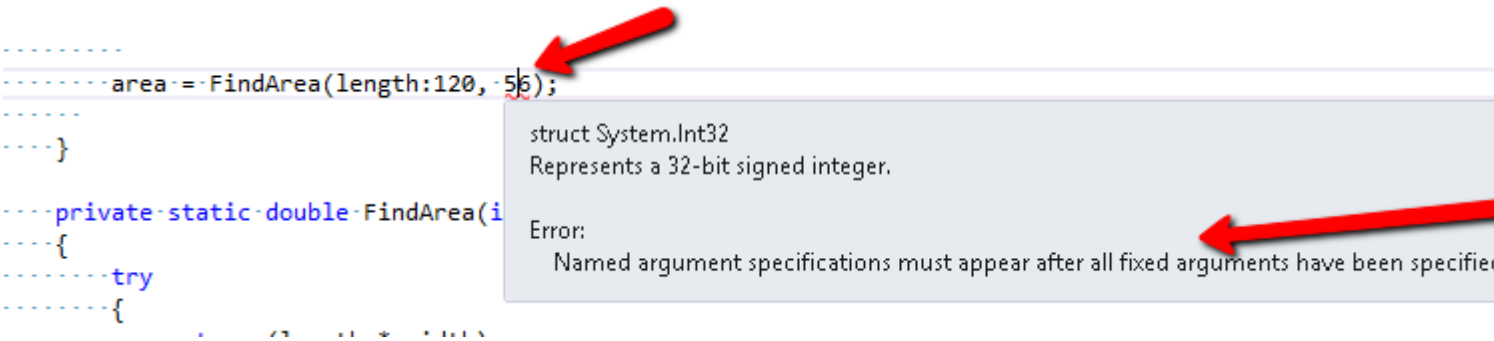
のでは、さとして120をし、パラメータのきとして56をしました。

いくつかのもあります。ここでは、きのについてします。

きのの

きのは、すべてのがされたでなければなりません。

のにきをすると、のようにコンパイルエラーがします。



きのは、すべてのがされたでなければなりません

オプションの

するものはオプションのをつです。

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

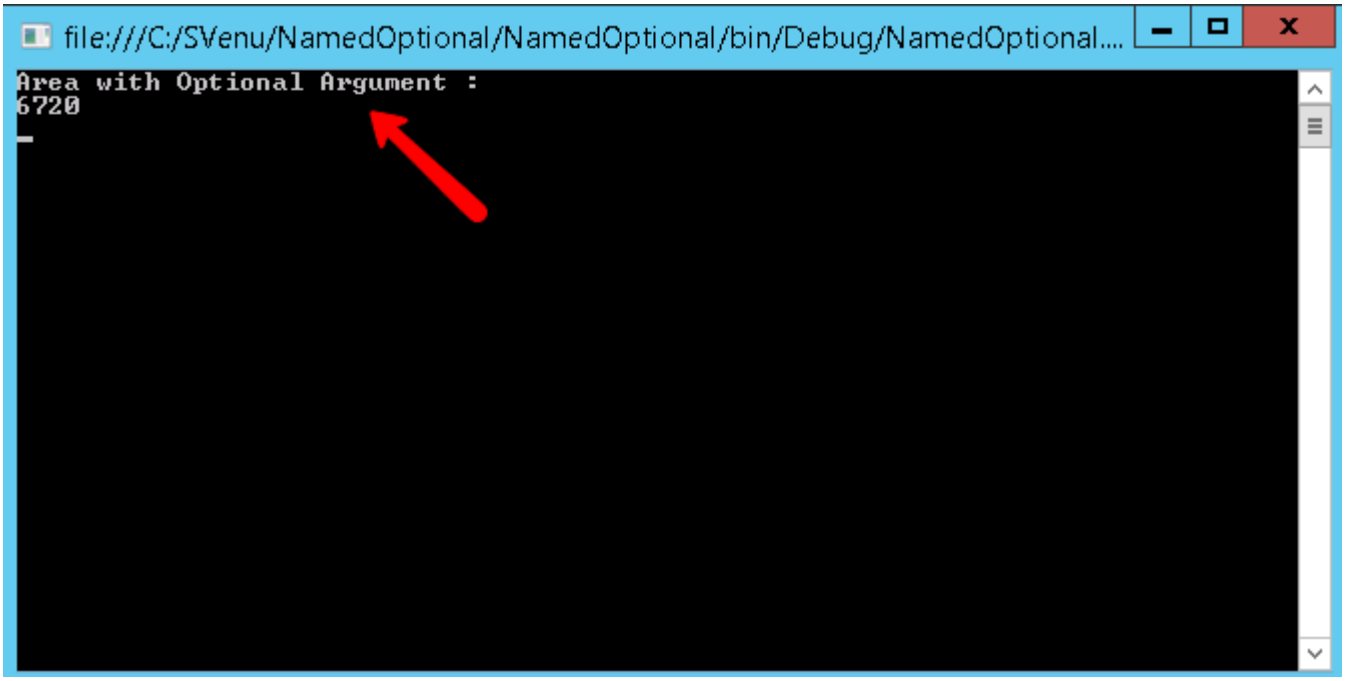
ここでは、widthのをオプションとしてし、として56をえました。メモをとると、IntelliSenseには、ののようにオプションのがされます。

```
area=FindAreaWithOptional(
```

double Program.FindAreaWithOptional(**int length**, [int width = 56])

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

コンパイルにエラーがしなかったことにしてください。のようにされます。



オプションの。

オプションのをするのは、 `[Optional]`キーワードをすることです。オプションののをさないと、そのデータのデフォルトがそのにりてられます。 `Optional`キーワードは、 "Runtime.InteropServices"にします。

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

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

をびすと、2のがされず、intのデフォルトが0なのでが0になるので、は0になります。

オンラインできとオプションをむ https://riptutorial.com/ja/csharp/topic/5220/きとオプション

130:

き

このトピックでは、Cでするにされるなについてします。すべてのと、コンパイラによってされるのではなく、のをします。

な.NETフレームワークガイドラインについては、 docs.microsoft.com/dotnet/standard/ design-guidelinesをしてください。

みやすいをする

たとえば、HorizontalAlignmentというプロパティは、AlignmentHorizontalよりもでみやすくなります。

よりみやすさをする

プロパティ `CanScrollHorizontally`は、 `ScrollableX` Xへのあいまいなよりもれています。

アンダースコア、ハイフン、またはそののののはしないでください。

ハンガリーをしない

ハンガリーは、のデータえば、 `string strName` のような、パラメータにするいくつかのメタデータをするためににをめるである。

また、Cですでにされているキーワードとするをしないでください。

と

に、やはしないでください。これらはあなたのをみにくくします。に、がくされているとすることがいつであるかをることはです。

Examples

の

のは、とをするをします。

パスカルケーシング

ののとそのにされるののがになります。 3つののには、パスカルのをできます。 `BackColor`

キャメルケーシング

ののはで、そのにされるののはになります。 `backColor`

のすべてのがになります。たとえば、 `IO`

ルール

がのでされている、のにアンダースコア "_"やハイフン " - "などのセパレータはしないでください。わりに、のをすために、caseをします。

のは、ののをし、さまざまなののをしています。

ローカル	キャメル	carName
クラス	パスカル	AppDomain
	パスカル	ErrorLevel
	パスカル	なり
イベント	パスカル	ValueChanged
クラス	パスカル	WebException
みりフィールド	パスカル	RedValue
インタフェース	パスカル	IDisposable
	パスカル	ToString
	パスカル	System.Drawing
パラメータ	キャメル	
プロパティ	パスカル	バックカラー

は、 MSDNをしてください 。

インターフェイス

インタフェースには、や、またはをするでをけるがあります。たとえば、 `IComponent`はなをし、

`ICustomAttributeProvider`はをし、 `IPersistable`はをします。

インタフエースは`I`でをけて、がインタフエースであることをし、パスカルのをするがあります。

はしくきのインターフェイスです

```
public interface IServiceProvider
public interface IFormatable
```

プライベートフィールド

プライベートフィールドには、 `camelCase`と`_camelCaseWithLeadingUnderscore` 2つののがあります。

キャメルケース

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

きのキャメルケース

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;

    public Rational(int numerator, int denominator)
    {
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

のなはのとおりです。

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

としては、

```
Fabrikam.Math
```

```
Litware.Security
```

のにをけると、なるののがじになることはありません。

ほとんどのにのをする

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

ビットフィールドであるにはのをする

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

に*FlagsAttribute*をビットフィールドの*Enum*にしてください。

として'enum'をしないでください

```
public enum VolumeEnum // Incorrect
```

エントリに**enum**をしないでください

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

として''をする

カスタムのに "-Exception"をけるがあります。

はしくがけられたです

```
public class MyCustomException : Exception
public class FooException : Exception
```

オンライ ンでをむ https://riptutorial.com/ja/csharp/topic/2330/

131:

Examples

のの

はOOPののつです。ポリはギリシャであり、をしています。

はをすです。 `Vehicle`クラスは、のフォームをクラスとしてります。

Derivedクラス`Ducati`と`Lamborghini`は`Vehicle`からし、クラスの`Display()`メソッドをオーバーライドして、の`NumberOfWheels`をします。

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
}
```

は、がされているコードスニペットです。オブジェクトは、Line 1で`Vehicle`をしてタイプ`Vehicle`にされます`vehicle`でクラスメソッド`Display()`をびし、をします。

```
static void Main(string[] args)
{
   Vehicle vehicle = new Vehicle();    //Line 1
   vehicle.Display();                  //Line 2
   vehicle = new Ducati();             //Line 3
   vehicle.Display();                  //Line 4
   vehicle = new Lamborghini();        //Line 5
   vehicle.Display();                  //Line 6
}
```

3では、 `vehicle`オブジェクトはクラス`Ducati`けられ、 `Display()`メソッドがびされ、がされます。ここでがる、オブジェクトにもかかわらず`vehicle`である`Vehicle`は、クラスのメソッドびし`Display()`タイプなどの`Ducati`クラスのオーバーライド`Display()`ので、このを`vehicle`オブジェクトがかってっている`Ducati`。

`Lamborghini`タイプの`Display()`メソッドをびすもじがされます。

アウトプットはのりです

```
The number of wheels for the Vehicle is 0        // Line 2
The number of wheels for the Ducati is 2         // Line 4
The number of wheels for the Lamborghini is 4    // Line 6
```

のタイプ

とは、をののにすることもできます。

にはのタイプがあります。

- アドホック
  `function overloading`がまれてい`function overloading`。 は、メソッドがジェネリックでなくてもなるタイプでできるということです。
- パラメトリック
  ジェネリックのです。 ジェネリックスを
- サブタイプ
  のをするクラスをしている

アドホック

`Ad hoc polymorphism`のは、びしやジェネリックのをとせずに、なるデータでびすことができるメソッドをすることです。のメソッド`sumInt(par1, par2)`は、なるデータでびすことができ、のみわせごとにのをっています。

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

に、コールのをします。

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 ));  //  3
    Console.WriteLine(sumInt("3","4"));  //  7
    Console.WriteLine(sumInt("5", 6 ));  // 11
    Console.WriteLine(sumInt( 7 ,"8"));  // 15
}
```

サブタイプ

サブタイプは、クラスからしてのをすることです。

```
public interface Car{
    void refuel();
}

public class NormalCar : Car
{
    public void refuel()
    {
```

```
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

`NormalCar`と`ElectricCar`のクラスには、、のがありますが、のです。ここにがあります

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

はのようになります。

ガソリンで
バッテリーの

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1589/

132:

Examples

カスタムの

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using
System.AttributeUsage Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

をする

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

をみむ

メソッド`GetCustomAttributes`は、メンバーにされるカスタムのをします。このをした、1つまたはののをできます。

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

またはそれらをりす

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`System.Reflection.CustomAttributeExtensions` `GetCustomAttribute`メソッドは、されたのカスタムをし、の`MemberInfo`できます。

```
var attribute = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute`には、するのをするためのシグネチャもあります。

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

ブールの`inherit`は、これらのメソッドのにす`inherit`ができます。このを`true`すると、のもされます。

DebuggerDisplay

`DebuggerDisplay`をすると、 `DebuggerDisplay`がにったときにデバッガがそのクラスをするがされます。

`{}`まれたは、デバッガによってされます。これは、のサンプルやよりなロジックのようななプロパティにすることができます。

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```

じの,`nq`すると、をするときにがされます。

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

`{}`ではながされていますが、されません。 `DebuggerDisplay`は、アセンブリメタデータにとしてき

まれます。 {}ののはチェックされません。したがって、よりなロジックをむDebuggerDisplay、つまりCではうまくいくかもしれませんが、VB.NETでされたじははではなく、デバッグにエラ―がします。

DebuggerDisplayをにしないようにするは、メソッドやプロパティにをしてわりにびすことです。

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

   private string DebuggerDisplay()
    {
        return $"{StringProperty} - {IntProperty}"";
    }
}
```

DebuggerDisplayがプロパティのまたはをし、オブジェクトのタイプもデバッグしてべることができます。
のは、 DebuggerDisplayがデバッグでされるため、ヘルパ―メソッドを#if DEBUGみます。

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

#if DEBUG
   private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
Type:{this.GetType()}";
    }
    #endif
}
```

びしをして、びしにするをびされたメソッドにすことができます。はのようになります。

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
                         [CallerMemberName]string callerMemberName = "",
                         [CallerLineNumber]int callerLineNumber = 0,
                         [CallerFilePath]string callerFilePath = "")
{
    //perform logging
}
```

びしはのようになります。

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

のパラメ―タのみがに`LogException`メソッドにされ、りはコンパイルにするとともにされることにしてください。

`callerMemberName`パラメ―タは、びしのメソッドのである`"Save"`というをけり`"Save"`。

`callerLineNumber`パラメ―ター―は、いずれのののける`LogException`メソッドびしができまれます。

また、 'callerFilePath'パラメ―タは`Save`メソッドがされているファイルのフルパスをけり`Save`。

インタフェ―スからをみる

クラスはインタフェ―スからをしないため、インタフェ―スからをするなはありません。インタ―フェイスをしたり、クラスのメンバ―をオ―バ―ライドするはに、をするがあります。したがって、のでは、3つのケ―スすべてで`True`がされます。

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

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True
```

```
            var attribute2 =
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}
```

インターフェイスをする1つのは、クラスによってされたすべてのインターフェイスをじてインターフェイスをすることです。

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

された

System.Obsoleteは、よりいバージョンをつまたはメンバーをマークするためにされるであるため、しないでください。

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

のクラスがされている、コンパイラは "This class is obsolete。SomeOtherClassをわりにする"というをします。

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1062/

133: メソッド

- public static ReturnType MyExtensionMethodこのTargetTypeターゲット
- public static ReturnType MyExtensionMethodこのTargetTypeターゲット、TArg1 arg1、...

パラメーター

パラメータ	
この	メソッドののパラメータのにはず`this`キーワードがき、するオブジェクトの「の」インスタンスをするがきます

メソッドは、オブジェクトインスタンスでスタティックメソッドをのメンバのようにびすことをにするシンタックスシュガーです。

メソッドには、なターゲットオブジェクトがです。キーワードそのものからメソッドにアクセスするには、 `this`キーワードをするがあります。

メソッドはされ、クラスにしなければなりません。

どのですか

メソッドクラスのネームスペースのは、とのトレードオフです。

もにされているオプションは、メソッドのカスタムをつことです。しかし、これには、コードのユーザーがメソッドがすること、およびそれらをつけるをるように、コミュニケーションがです。

のとして、がIntellisenseをしてメソッドをできるようにをするがあります。したがって、 `Foo`クラスをしたいは、メソッドを`Foo`とじにするのがにかなっています。

"か"のをすることをげるものはもないことをすることがです。したがって、 `IEnumerable`をするは、 `System.Linq`にメソッドをできます。

これはずしもいえではありません。たとえば、あるのケースでは、な `bool IsApproxEqualTo(this double value, double other)`をすることができますが、 `System`を'する'ことはできません。この、ローカルののをすることがましいです。

に、メソッドをにくれないこともです

いの メソッドのをどうやってしていますか

メソッドをして、なすべてのにしていることをし、のにするだけでなく、メソッドをするときは

がです。たとえば、 `string`などのシステムクラスをして、しいコーードをのでできるようにすることができます。コーードがドメインのでドメインのロジックをするがある、そのがシステムののをうびしをさせるため、メソッドはではありません。

のリストに、メソッドのなとプロパティをします

1. これはメソッドでなければなりません。
2. クラスにするがあります。
3. これは、 "this"キーワードを.NETのをつのパラメータとしてし、このメソッドはクライアントのされたインスタンスによってびされます。
4. それはまた、VS intellisenseによってされた。たちは、ドットをすと.タイプインスタンスの、それはVS intellisenseでる。
5. メソッドは、されているのとじにあるか、usingステートメントでクラスのをインポートするがあります。
6. メソッドをつクラスにはのをけることができますが、クラスはでなければなりません。
7. にしいメソッドをしたいが、そのためのソースコードがないは、そののメソッドをしてすることです。
8. しているとじシグネチャメソッドをつメソッドをした、メソッドはしてびされません。

Examples

メソッド -

メソッドはC3.0でされました。メソッドは、しいをしたり、コンパイルしたり、のをせずに、のにをしたりしたりします。 これらは、するためにしているタイプのソースをできないににちます。メソッドは、システムタイプ、サードパーティによってされたタイプ、およびでしたタイプにしてすることができます。メソッドは、ののメンバーメソッドであるかのようにびすことができます。これにより、 **Fluent Interface**のにされる**Method Chaining**がになります。

メソッドは、されるのとはなるクラスに メソッドをすることによってされます。メソッドをするクラスは、メソッドをするののためにされることがよくあります。

メソッドは、されるのをするな1パラメータをとります。こののパラメータは`this`というキーワードでられています `this`は、Cでにいけられ`this`いますが、のオブジェクトインスタンスのメンバをできるようにするためにしてください。

のでは、されるのはクラス`string`です。 `Shorten()`メソッドによって`String`がされました。ショートニングのをします。クラス`StringExtensions`は、メソッドをするためにされています。メソッド`Shorten()`は、にマークされたのパラメータをして`string`をしたものであることをしています。`Shorten()`メソッドが`string`していることをすために、のパラメータに`this`とマークされて`this`ます。したがって、のパラメータのなは`this string text`。 ここで、 `string`はされるので、 `text`はされたパラメータです。

```
static class StringExtensions
{
```

```
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

.NET Fiddleのライブデモ

メソッドのの `this`キーワードがいているとしてされたオブジェクトは、メソッドがびされたインスタンスです。

たとえば、このコードがされると、のようになります。

```
"some string".Shorten(5);
```

のはのとおりです。

```
text: "some string"
length: 5
```

メソッドは、それらのとじにある、がメソッドをしてコードによってにインポートされた、またはクラスがなしであるにのみできます。 .NET Frameworkのガイドラインでは、クラスをのにすることをしています。ただし、これはのにつながるがあります。

これにより、するのあるがにきまれないり、メソッドとされているライブラリとのにはじません。たとえば、 LINQ

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
```

```
    }
}
```

.NET Fiddleのライブデモ

C6.0、メソッドをむクラスに `using static`ディレクティブを`using static`することもできます。たとえば、 `using static System.Linq.Enumerable;`。 これにより、じののをスコープにたせることなく、そののクラスのメソッドをできるようになります。

じシグネチャをつクラスメソッドがな、コンパイラはメソッドびしよりします。えば

```
class Test
{
   public void Hello()
   {
       Console.WriteLine("From Test");
   }
}

static class TestExtensions
{
    public static void Hello(this Test test)
    {
        Console.WriteLine("From extension method");
    }
}

class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

.NET Fiddleのライブデモ

じシグネチャをつ2つのがあり、そのうちの1つがじにある、そのがされることにしてください。がによってアクセスされる、 `using` 、コンパイルエラーがメッセージとにとしてきるであろう。

> このびしは、のメソッドまたはプロパティでです

`originalTypeInstance.ExtensionMethod()`をしてメソッドをびすののはオプションです。メソッドは、のでびすこともできます。そのため、な1パラメータがメソッドのパラメータとしてされます。

つまり、のの

```
//Calling as though method belongs to string--it seamlessly extends string
```

```
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

にメソッドをする

メソッドは、のクラスメソッドのようにすることもできます。メソッドをびすこのは、よりですが、によってはです。

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

メソッドをメソッドとしてびすタイミング

メソッドをメソッドとしてするがあるシナリオはまだあります。

- メンバーメソッドとのの。これは、ライブラリのしいバージョンでじシグネチャをつしいメンバメソッドがされたにします。この、メンバメソッドはコンパイラによってされます。
- じをつのメソッドとのの。これは、2つのライブラリにのメソッドがまれていて、メソッドをつのクラスのがじファイルでされているにします。
- メソッドグループとしてのメソッドをパラメータにします。
- `Reflection`をしてあなたのをしています。
- Visual Studioのイミディエイトウィンドウでメソッドをする。

な

`using static`ディレクティブをしてクラスのメンバーをグローバルスコープにすると、メソッドはスキップされます。

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

`Shorten`メソッドののから`this`をすると、のがコンパイルされます。

ヌルチェック

メソッドは、インスタンスメソッドのようにするメソッドです。しかし、のインスタンスメソッドをびすときにがこるかとはって`null`メソッドをしてびされ、、 `null`、それはスロ―されません`NullReferenceException`。 これは、いくつかのシナリオではにです。

たとえば、のクラスをえてみましょう。

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}
```

```
string nullString = null;
string emptyString = nullString.EmptyIfNull();// will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null
```

.NET Fiddleのライブデモ

メソッドは、クラスのパブリックまたはメンバ―のみをできます

```
public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
```

```
}
```

メソッドはなるなであり、にするクラスのメンバーではありません。これは、カプセルをることができないことをします。 `public` またはじアセンブリ、 `internal` されているフィールド、プロパティ、およびメソッドにのみアクセスできます。

メソッド

のメソッドとじように、メソッドはジェネリックをできます。えば

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

それをびすのはのようなものです

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

デモをる

にのの

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

それをびすことはのようになります

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

デモをる

また、にバインドされたのメソッドを、のでることもできます。

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
```

```
    }
}
```

それをびすことはのようになります

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

デモをる

あなたはまた、とのをすることができます `where`

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

びしコードド

```
int number = 5;
var IsDefault = number.IsDefault();
```

デモをる

にづいたメソッドのディスパッチ

ではなく、コンパイルがされ、パラメータがします。

```
public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
    {
        return "Derived";
```

```
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}
```

.NET Fiddleのライブデモ

また、にづくディスパッチでは、 `dynamic`オブジェクトにしてメソッドをびすことはできません。

```
public class Person
{
    public string Name { get; set; }
}

public static class ExtenionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

メソッドはコードではサポートされていません。

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));
```

```
        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

は[ダイナミックコードからのメソッドのびし]はしません。なぜなら、ではないコードメソッドは、コンパイラがっているすべてのクラスをして、するメソッドをつクラス。はネストスペースネストにづいてにみ、ネームスペースでディレクティブを`using`になります。

それは、しくメソッドのびしをるために、とかDLRは、すべてののネストとどのようなにっていなければならないことを`using`ディレクティブは、ソースコードにありました。たちは、そのをすべてコールサイトにエンコードするためのメカニズムをしていません。たちはそのようなみをしたとっていましたが、コストがぎて、それにするスケジュールリスクがあまりにもきかったとしました。

ソース

なきラッパーとしてのメソッド

メソッドは、のようなオブジェクトのためのくけされたラッパーをくためにすることができます。たとえば、キャッシュ、 `HttpContext.Items` at cetera ...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

このアプローチでは、コードベースのキーとしてリテラルをするがなくなり、みりになにキャストするがなくなります。には、よりでくけされた、などのオブジェクトとやりりするをします。

のメソッド

メソッドが`this`とじをつをすとき、それはのあるで1つのメソッドびしを「する」ためにできます。これは、やプリミティブのにであり、メソッドがのようにめば、いわゆる「」なAPIをすることができます。

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

またはこのように

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6};
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

インタフェースとみわせたメソッド

インプリメンテーションはクラスのにすることができ、クラスにいくつかのをするためになのは、インタフェースをつクラスをデコレートすることだけです。

```
public interface IInterface
{
   string Do()
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
      //does something with IInterface instance
    }
}
```

```
public class Classy : IInterface
{
   // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
   // provided you import the namespace containing the extension method
   public Do(){
       return this.DoWith();
   }
}
```

のようにします。

```
 var classy = new Classy();
 classy.Do(); // will call the extension
 classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

IList メソッドの2リストの

のメソッドをして、じの2つのIListインスタンスのをできます。

デフォルトでは、はリストのとにづいてされ、falseを`isOrdered`パラメータにすと、になくがされます。

このがするために、ジェネリックタイプ `T` オーバーライドするがあり`Equals`と`GetHashCode`を。

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2);//this gives false
list1.Compare(list2, false);//this gives true. they are equal when the order is disregarded
```

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                 (l1 == null && l2 != null) ||
                 (l1 != null && l2 == null) ||
                 (!l1.Equals(l2)))
            {
                    return false;
            }
        }
        return true;
    }
    else
```

```
    {
        List<T> list2Copy = new List<T>(list2);
        //Can be done with Dictionary without O(n^2)
        for (int i = 0; i < list1.Count; i++)
        {
            if (!list2Copy.Remove(list1[i]))
                return false;
        }
        return true;
    }
}
```

Enumerationをしたメソッド

メソッドは、をにするのにです。

1つのなは、をすることです。

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }
    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

これで、enumをのタイプにくできます。このはブーールです。

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

あるいは、メソッドをしてメソッドのようなプロパティをすることができます。

```
public enum Element
{
    Hydrogen,
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
    Nitrogen,
    Oxygen
    //Etc
}
```

```
public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen:  return 1.00794;
            case Element.Helium:    return 4.002602;
            case Element.Lithium:   return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:     return 10.811;
            case Element.Carbon:    return 12.0107;
            case Element.Nitrogen:  return 14.0067;
            case Element.Oxygen:    return 15.9994;
            //Etc
        }
        return double.Nan;
    }
}

var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

とインターフェースにより、DRYコードとミックスインのようながになります

メソッドをすると、インターフェイスにコアをめるだけで、インターフェイスをでき、なメソッドとオーバーロードをメソッドとしてできるようになります。メソッドのがないインタフェースは、しいクラスでするがです。オーバーロードをインターフェイスにめるのではなく、としてそのまますることで、コードをすべてのにコピーすることがなくなり、コードのDRYをできます。これはにはCがサポートしていないmixinパターンにています。

`System.Linq.Enumerable`の`IEnumerable<T>`へのは、これのらしいです。 `IEnumerable<T>`では、クラスではとの2つのメソッド`GetEnumerator()`をするがあります。しかし、 `System.Linq.Enumerable`は、`IEnumerable<T>`でなをにするとして、のなユーティリティをします。

は、としてされるなオーバーロードをえたになインターフェースです。

```
public interface ITimeFormatter
{
   string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
public class SecondsTimeFormatter : ITimeFormatter
{
```

```
   public string Format(TimeSpan span)
   {
       return $"{(int)span.TotalSeconds}s";
   }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is rougly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}
```

なケ―スをうためのメソッド

エクステンションメソッドは、そうでなければif / thenステ―トメントでびしをにするのある、されていないビジネスル―ルのを「す」ためにできます。これは、メソッドでnullをするとです。えば、

```
public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty
but are known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() :
cake;
    }
}
```

```
Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake);//Eat the cake, confident that it is not a lie.
```

メソッドとコ―ルバックでのメソッドの

のコ―ドをラップするとしてメソッドをすることをしてください。ここでは、Try Catchをラップするためのメソッドとメソッドのをするれたです。あなたのコ―ドのをる...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
```

```
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
```

```
            {
                //error is the already logged and handled exception
                //determine the base result
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

インタフェースのメソッド

メソッドのなの1つは、インターフェイスのなメソッドをできることです。、インタフェースはをつことはできませんが、メソッドをすることができます。

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

このでは、 `FeetDriven`メソッドはの`IVehicle`でできます。こののこのロジックはすべての`IVehicle`されるため、このでうことができ、 `IVehicle`に`FeetDriven`がなくても、すべてのにしてじでされます。

メソッドをってしいマッパークラスをする

メソッドをってよりいマッパークラスをすることができます。もしがいくつかのDTOクラスをっていたら

```
 public class UserDTO
 {
        public AddressDTO Address { get; set; }
 }
```

```
 public class AddressDTO
 {
        public string Name { get; set; }
 }
```

はするビュ―モデルクラスにマップするがあります

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

はのようにのマッパ―クラスをすることができます

```
public static class ViewModelMapper
{
      public static UserViewModel ToViewModel(this UserDTO user)
      {
            return user == null ?
                null :
                new UserViewModel()
                {
                    Address = user.Address.ToViewModel()
                    // Job = user.Job.ToViewModel(),
                    // Contact = user.Contact.ToViewModel() .. and so on
                };
      }

      public static AddressViewModel ToViewModel(this AddressDTO userAddr)
      {
            return userAddr == null ?
                null :
                new AddressViewModel()
                {
                    Name = userAddr.Name
                };
      }
}
```

その、はマッパをのようにびすことができます

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
        };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

ここのしさは、すべてのマッピングメソッドがToViewModelをち、いくつかのでできることです

メソッドをしてしいコレクションタイプをするDictList

`List<T>`をつ`Dictionary`ようなネストされたコレクションのユーザビリティをさせるメソッドをすることができます。

のメソッドをえてみましょう。

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
            where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection>
dict, TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }

        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

のようにメソッドをできます。

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15
```

```
dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False
```

デモをる

オンラインでメソッドをむ https://riptutorial.com/ja/csharp/topic/20/メソッド

134: の

`using`キーワードは、ディレクティブこのトピックとステートメントのです。

`using`ステートメントつまり、 `IDisposable`オブジェクトのスコープをカプセルし、そのスコープのにオブジェクトがきれいにされることをするについては、 Using Statementをしてください。

Examples

な

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console;  //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

をする

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace.  i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

ネームスペースにエイリアスをける

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
//prefixing them with only the defined alias and not the full namespace.  i.e:

//...
var sb = new st.StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

クラスのメンバーへのアクセス

6.0

のタイプをインポートし、タイプでせずにそのタイプのスタティックメンバーをできます。これは、メソッドをしたをしています。

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

これは、なプロパティとメソッドをしたをしています。

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

コンフリクトをするためのエイリアスのけ

じのクラス `System.Random`や`UnityEngine.Random`などをつのあるのをしているは、をして、 `Random`がびしのをすることなく、。

えば

```
using UnityEngine;
using System;

Random rnd = new Random();
```

これにより、しいをする`Random`がコンパイラによってになります。わりに、のことができます。

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

これはのでのものをびすことをげるものではありません。

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd`は`System.Random`になり、 `unityRandom`は`UnityEngine.Random`になります。

ディレクティブの

`using`すると、またはのエイリアスをできます。はこちらをごください 。

```
using <identifier> = <namespace-or-type-name>;
```

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

オンラインでのをむ https://riptutorial.com/ja/csharp/topic/52/の

135: エスケープシーケンス

- \' - 0x0027
- \" - 0x0022
- \\ - バックスラッシュ0x005C
- \0 - ヌル0x0000
- \a - アラート0x0007
- \b - バックスペース0x0008
- \f - フォームフィード0x000C
- \n - 0x000A
- \r - キャリッジリターン0x000D
- \t - タブ0x0009
- \v - タブ0x000B
- \u0000 - \uFFFF - Unicode
- \x0 - \xFFFF - Unicodeのコード
- \U00000000 - \U0010FFFF - Unicodeサロゲートをするため

エスケープシーケンスは、コンパイルにするにされます。スラッシュをむのはされません。

えば、の`notEscaped`と`notEscaped2`のはにされませんが、2つのなる`'\'`と`'n'`としてります。

```
string escaped = "\n";
string notEscaped = "\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

Examples

Unicodeエスケープシーケンス

```
string sqrt = "\u221A";      // √
string emoji = "\U0001F601"; // 
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

リテラルのをエスケープする

アポストロフィ

```
char apostrophe = '\'';
```

バックスラッシュ

```
char oneBackslash = '\\';
```

リテラルのをエスケープする

バックスラッシュ

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

2のでは、バックスラッシュをエスケープとしてわないリテラルをしています。

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @"""Hello World!"", said the quick brown fox.";
```

のにじテキストがまれます。

"ハローワールド"、クイックブラウンキツネはった。

ニューラインズ

なリテラルは、をむことができます

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

のにじテキストがまれます。

できないエスケープシーケンスがコンパイルエラーをする

のはコンパイルされません。

```
string s = "\c";
char c = '\c';
```

わりに、コンパイルに「`Unrecognized escape sequence`というエラーがされます。

にエスケープシーケンスをする

エスケープシーケンスは、`string`と`char`リテラルにされません。

サードパーティメソッドをオーバーライドするがあるとします。

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

あなたのCソースファイルにするエンコーディングで`Œ`ができないとします。あなたはいですが、

コードのに\u####または\U########エスケープをすることはされています。それで、くのはです

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

Cコンパイラは、Œと\u0152がじであることを\u0152ます。

ただし、すべてのをできるUTF-8またはのエンコーディングにりえることをおめします。

オンラインでエスケープシーケンスをむ https://riptutorial.com/ja/csharp/topic/39/エスケープシーケンス

136: の

- $ "コンテンツ{}のコンテンツ"
- $ "コンテンツ{フォーマット}コンテンツ"
- $ "content {expression} {{のコンテンツ}}}}"
- $ "コンテンツ{フォーマット} {{のコンテンツ}}コンテンツ}" "

は`string.Format()`メソッドので、にとのをむをしやすくします。

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);  // returns "Hello, World"
var newWay = $"Hello, {name}!";                   // returns "Hello, World"
```

Examples

なは、されたでもできます。

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

.NET Fiddleのライブデモ

のの

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

`DateTime.ToString`メソッドをして、`DateTime`オブジェクトをフォーマットすることもできます。これにより、のコードとじがされます。

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

20151111、いを

.NET Fiddleのライブデモ

DateTime.ToStringをしたライブデモ

`MM`はをし、`mm`はをします。いがしにくいバグをするがあるため、これらをするはにしてください。

な

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

には

```
$"Hello, {name}!"
```

このようなものにコンパイルされます

```
string.Format("Hello, {0}!", name);
```

をめむ

は、されたがするのをするパディングパラメータをけれるようにフォーマットできます。

```
${value, padding}
```

のパディングはのパディングをし、のパディングはのパディングをします。

パディング

パディング5numberののに3つのスペースがされるため、のには5のがです。

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

```
The answer to life, the universe and everything is    42.
```

.NET Fiddleのライブデモ

パディング

のパディングをするパディングでは、ののにがされます。

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42   .";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

```
The answer to life, the universe and everything is 42   .
```

.NET Fiddleのライブデモ

によるパディング

のをパディングとみわせてすることもできます。

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//                                                          ^^^^^
```

.NET Fiddleのライブデモ

のの

コロンとをして、 のをできます。

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;

var prefixedIfNecessary = $"{integerValue:D5}";
// String value is "00057"
```

.NET Fiddleのライブデモ

オンラインでのをむ https://riptutorial.com/ja/csharp/topic/22/の

137: をのにするのFormatExceptionの

Examples

をにする

に`string`を`integer`するには、 `integer`ようなさまざまながあります。

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

しかし、にのがまれる、これらのメソッドはすべて`FormatException`をスローします。そのためには、 `try..catch` をしてするがあります。

の

だから、たちのはのようになります

```
string inputString = "10.2";
```

1 `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

じのされたメソッドのためにく - *`Convert.ToInt16();`*と*`Convert.ToInt64();`*

2 `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

どうやってこれをするのですか

のように、をするには、、のように`try..catch`がです。

```
try
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
```

```
{
    //Display some message, that the conversion has failed.
}
```

しかし、してtry..catchどこでもすることはいではありません、とたちはえたかったいくつかのシナリオがあるかもしれ0がっている、 々はのにったは々がりてるが0までconvertedIntからキャッチブロック。このようなシナリオをするために、 .TryParse()というなメソッドをすることができます。

をつ.TryParse()メソッドは、 outパラメータへのをし、ステータスをすブールをしますがしたは*true* 、したは*false* 。 りについて、ステータスをすることができます。 をてみましょう。

1りをブールにする

```
 int convertedInt; // Be the required integer
 bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

にisSuccessConversionをチェックしてステータスをできます。 falseの、 convertedIntのは0になり0エラーが0はりをチェックするはありません。

2 if りをチェックする

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

3りをチェックせずに、りをにしないされているかどうか、 0はです 、をできます。

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

オンラインでをのにするのFormatExceptionのをむ https://riptutorial.com/ja/csharp/topic/2886/をのにするのformatexceptionの

138:

Examples

のの/の

System.Stringクラスは、のとのでするいくつかのメソッドをサポートしています。

- System.String.ToLowerInvariantは、Stringオブジェクトをにしてし System.String.ToLowerInvariant 。
- System.String.ToUpperInvariantは、にされたStringオブジェクトをすためにされます。

これらのメソッドのバージョンをするは、しないカルチャののをぐためです。これについてはここでしくします 。

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

あなたがのするかをできることにしてくださいをしてとにするString.ToLowerのCultureInfoをとString.ToUpperのCultureInfoにじた。

のをつける

System.String.Containsをすると、のがにするかどうかをべることができます。このメソッドはbooleanをします。がするはtrueをし、そうでないはfalseをします。

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

System.String.IndexOfメソッドをすると、ののののをつけることができます。
されるは0からまり、がつからないは-1がされることにしてください。

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

のからのをするには、 System.String.LastIndexOfメソッドをしSystem.String.LastIndexOf 。

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

からをトリミングする

System.String.Trimメソッドをして、からとのをすべてできます。

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

えて

- のからのみをするには System.String.TrimStart
- のからのみをするには、 System.String.TrimEnd

のをする。

System.String.Substringメソッドをすると、のをできます。

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

のをする

System.String.Replaceメソッドをすると、のをのできえることができます。

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

すべてのがされます。

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

String.Replaceは、としてのをすることによって、のをするためにもできます。

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

りをしてをする

System.String.Splitメソッドをして、されたりについてされたののをむをしSystem.String.Split。

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

1

のをのにする

`System.String.Join`メソッドをすると、にされたセパレータをして、のすべてのをでき`System.String.Join`。

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

は、 `System.String.Concat`メソッドをするか、 `+`をしてもっとにうことができます。

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/3599/

139:

あなたがなをしている、それはのためにふことをおめし`StringBuilder` +またはして、クラスではなく、する`Concat`+ /のようなを`Concat`それがされ、たなオブジェクトをされますが。

Examples

+

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

System.Text.StringBuilderをしてをする

StringBuilderをしてをすると、 `+`をしたなよりもパフォーマンスのがられます。これはメモリのりてによるものです。ストリングはでてされ、のブロックがいくされたときにのみりてされるように、StringBuildersはブロックでメモリーをりてます。これは、さなのくをうときにきないをることができます。

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

`Append()`へのびしは、 `StringBuilder`へのをすため、ディジーチェーンにすることができます

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

String.JoinをしたConcat

`String.Join`メソッドは、からのをするためにできます。

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

のをします。 "orange、grape"

このでは、セパレータとのにインデックスとカウントをする`String.Join(String, String[], Int32, Int32)`オーバーロードをします。

startIndexとcountのオーバーロードをしないは、されたすべてのにできます。このような

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

それはする;

リンゴ、オレンジ、ブドウ、

$をした2つのの

$は、のをするためのでなメソッドをします。

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/3616/

140: System.Security.Cryptography

Examples

されたのの

はかにしく、さまざまなをんで、らかのをするのがどれほどかをて、くのをやした、はにいとう@jbtuleによってかれたえをつけました。をしむ

「のためのなベストプラクティスは、するデータAEADでをすることですが、これはの.NETライブラリのではありません。のは、していますのでAES256 、そのHMAC256 、2つのステップの、をMAC 、くのオーバーヘッドとよりくのがです。

2のでは、オープンソースのBouncy Castleナゲットをしたaes256- GCMのよりなをしています。

のには、のメッセージ、キーおよびオプションのペイロードと、データがプリペンドされたとされたをけるながあります。には、これらを`NewKey()`のキーでランダムにし、 `NewKey()`してください。

のには、キーをするためにパスワードをするヘルパーメソッドもあります。これらのヘルパーメソッドは、のとマッチするためののためにされていますが、パスワードのは256ビットのキーよりもはるかにいので、 ははるかにくなります 。

アップデートされた`byte[]`オーバーロード、そしてGistだけがStackOverflowののために4スペースのげとapiドキュメントでなをしています。

.NETビルトインAES-Then-MACHMAC [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
```

```
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
HMac-Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
                        int nonSecretPayloadLength = 0)
```

```
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }
```

```
    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }

      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
```

```
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!",
KeyBitSize), "cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
```

```
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
              binaryWriter.Write(
                encryptedMessage,
                nonSecretPayloadLength + iv.Length,
                encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
              );
            }
            //Return Plain Text
            return plainTextStream.ToArray();
          }
        }
      }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length ==0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      int payloadIndex = nonSecretPayload.Length;

      byte[] cryptKey;
      byte[] authKey;
      //Use Random Salt to prevent pre-generated weak password attacks.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);

        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
      }

      //Deriving separate key, might be less efficient than using HKDF,
```

```
      //but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      }

      return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
authSalt.Length + nonSecretPayloadLength);
    }
  }
}
```

バウンシーキャッスル**AES-GCM** []

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
```

```
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33
Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
```

```
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
```

```
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
```

```
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");
```

```
      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +
nonSecretPayloadLength);
    }
  }
}
```

との

をすることで、データのやのセキュリティをさせることができます。に、
*System.Security.Cryptography*をする、 との 2つのアプローチがあります 。

このメソッドは、データをするためにをします。

- アルゴリズムは、よりないリソースをし、アルゴリズムよりもです。
- できるデータのはです。

- とはじをします。キーがされた、かがあなたのデータをすることができます。
- なるデータになるをすることをした、くのなるですることができます。

System.Security.Cryptographyでは、をするさまざまなクラスがあり、 ブロックとばれます 。

- AesManaged AESアルゴリズム。
- AesCryptoServiceProvider AESアルゴリズムFIPS 140-2クレーム 。
- DESCryptoServiceProvider DESアルゴリズム。
- RC2CryptoServiceProvider Rivest Cipher 2アルゴリズム。
- RijndaelManaged AESアルゴリズム。 RijndaelManagedはFIPS-197のではありません 。
- TripleDES TripleDESアルゴリズム。

このメソッドは、データをするためにキーとキーのみわせをします。

- それはアルゴリズムよりもきなキーをするので、なをってひびれをけにくい。
- 2つのとにしているため、がデータをおよびできるかをするがです。

- できるデータにはがあります。このはアルゴリズムごとになり、はアルゴリズムのキーサイズにします。たとえば、キーが1,024ビットのRSACryptoServiceProviderオブジェクトは、128バイトよりさいメッセージのみをできます。
- アルゴリズムは、アルゴリズムとしてにい。

System.Security.Cryptographyでは、をするさまざまなクラスにアクセスできます。

- DSACryptoServiceProvider デジタルアルゴリズム
- RSACryptoServiceProvider RSAアルゴリズムアルゴリズム

パスワードのハッシュ

パスワードをしてプレーンテキストとしてしないでくださいらは、パスワードのハッシュアルゴリズムがいことをして、にされたのにしてするためでハッシュするがあります。ブルートフォースをさせるためには、い> 10kをできます。ログインしたユーザーは100msのがされますが、いパスワードをることはです。くのをするは、アプリケーションのをし、コンピュータのパフォーマンスがするにつれてそのをやすがあります。また、DoSとしてされるのあるりしをすることもするがあります。

にハッシュをすると、されたハッシュとソルトをファイルにすることができます。

```
private void firstHash(string userName, string userPassword, int numberOfItterations)
{
```

```
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberOfItterations);
//Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberOfItterations); //Store the hashed
password with the salt and number of itterations to check against future password entries
}
```

のユーザーのパスワードをし、ファイルからハッシュとをみり、されたパスワードのハッシュとします

```
private bool checkPassword(string userName, string userPassword, int numberOfItterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
numberOfItterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMach = comparePasswords(usersHash, hashedPassword);    //Compares byte
arrays
    return passwordsMach;
}
```

シンプルなファイル

のコードサンプルは、AESアルゴリズムをしてファイルをおよびするでなをしています。

このコードは、ファイルがされるたびにランダムにベクタをします。つまり、じパスワードでじファイルをするとになるにつながります。 saltとIVはファイルにきまれ、するためにパスワードだけがとなります。

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
```

```
            fsIn.Read(salt, 0, saltLength);
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);

        // Encrypt or decrypt the file
        using (var cryptoTransform = encryptMode
            ? cypher.CreateEncryptor(key, iv)
            : cypher.CreateDecryptor(key, iv))
        using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}
```

でなランダムデータ

ジェネレータにづいているため、フレームワークのRandomクラスはにランダムではないがあります。ただし、フレームワークのCryptoクラスは、RNGCryptoServiceProviderのでよりなものをします。

のコードサンプルは、Cryptographically Secureバイト、およびをするをしています。

ランダムバイト

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

ランダム

```
public static int GenerateRandomInt(int minVal=0, int maxVal=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxVal - minVal + 1) + minVal);
}
```

ランダムな

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);

    // Generate the output string
    var allowable = allowableChars.ToCharArray();
    var l = allowable.Length;
    var chars = new char[length];
    for (var i = 0; i < length; i++)
        chars[i] = allowable[rnd[i] % l];

    return new string(chars);
}
```

のファイル

は、メッセ―ジをにするために、よりもしばしばましいとなされます。これはに、のにするくのリスクをにし、をつもがしたのメッセ―ジをできるにそのだけができることをするためです。ながら、アルゴリズムのなは、それらがないとこよりもかなりいことです。このように、ファイルの、になは、になプロセスになるがあります。

セキュリティとパフォ―マンスのをするために、ハイブリッドアプロ―チをとることができます。これは、 のためのキ―とベクトルのなランダムをとする。これらのは、 *Asymmetric*アルゴリズムをしてされ、ソ―スファイルをにしてにするにファイルにきまれます。

このアプロ―チは、デ―タがアルゴリズムをしてされ、キ―とivのがランダムにされ、アルゴリズムでされるというで、また、のにされたじペイロ―ドには、がランダムにされるため、になるがするというのがあります。

のクラスは、ハイブリッドファイルとに、とバイトのをします。

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more
robust but takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion
```

```
    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }

    public static string DecryptString(string value, string privateKey)
    {
        return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value),
privateKey));
    }

    #endregion

    #region Hybrid File Encryption and Decription

    public static void EncryptFile(string inputFilePath, string outputFilePath, string
publicKey)
    {
        using (var symmetricCypher = new AesManaged())
        {
            // Generate random key and IV for symmetric encryption
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(key);
                rng.GetBytes(iv);
            }

            // Encrypt the symmetric key and IV
            var buf = new byte[key.Length + iv.Length];
            Array.Copy(key, buf, key.Length);
            Array.Copy(iv, 0, buf, key.Length, iv.Length);
            buf = EncryptData(buf, publicKey);

            var bufLen = BitConverter.GetBytes(buf.Length);
```

```
            // Symmetrically encrypt the data and write it to the file, along with the
encrypted key and iv
            using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
            using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsOut.Write(bufLen,0, bufLen.Length);
                fsOut.Write(buf, 0, buf.Length);
                fsIn.CopyTo(cs);
            }
        }
    }

    public static void DecryptFile(string inputFilePath, string outputFilePath, string
privateKey)
    {
        using (var symmetricCypher = new AesManaged())
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        {
            // Determine the length of the encrypted key and IV
            var buf = new byte[sizeof(int)];
            fsIn.Read(buf, 0, buf.Length);
            var bufLen = BitConverter.ToInt32(buf, 0);

            // Read the encrypted key and IV data from the file and decrypt using the
asymmetric algorithm
            buf = new byte[bufLen];
            fsIn.Read(buf, 0, buf.Length);
            buf = DecryptData(buf, privateKey);

            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            Array.Copy(buf, key, key.Length);
            Array.Copy(buf, key.Length, iv, 0, iv.Length);

            // Decript the file data using the symmetric algorithm
            using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsIn.CopyTo(cs);
            }
        }
    }

    #endregion

    #region Key Storage

    public static void WritePublicKey(string publicKeyFilePath, string publicKey)
    {
        File.WriteAllText(publicKeyFilePath, publicKey);
    }
    public static string ReadPublicKey(string publicKeyFilePath)
    {
        return File.ReadAllText(publicKeyFilePath);
    }

    private const string SymmetricSalt = "Stack_Overflow!"; // Change me!
```

```
    public static string ReadPrivateKey(string privateKeyFilePath, string password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        var cypherText = File.ReadAllBytes(privateKeyFilePath);

        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var decryptor = cypher.CreateDecryptor(key, iv))
            using (var msDecrypt = new MemoryStream(cypherText))
            using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
                return srDecrypt.ReadToEnd();
            }
        }
    }

    public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string
password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var encryptor = cypher.CreateEncryptor(key, iv))
            using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
            using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor,
CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(privateKey);
            }
        }
    }

    #endregion
}
```

```
private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey,
privateKeyPassword);
```

```
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}
```

オンラインでSystem.Security.Cryptographyをむ https://riptutorial.com/ja/csharp/topic/2988/-system-security-cryptography-

141:

Examples

If-Elseステートメント

に、プログラミングは、コードがなるまたはでどのようにするかをするために、コードの`decision`または`branch`とすることがあります。 Cプログラミングおよびこののほとんどのプログラミングでは、プログラムにブランチをするもでにはもなは、 `If-Else`ステートメントによるものです。

100までのスコアをすintパラメータをとるメソッドがあるとし、メソッドはかかをすメッセージをします。

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

このメソッドをると、 `If`のにこのコード `score >= 50` があることにくかもしれません。これは`boolean`と`boolean`ことができます。 が`true`としいとされた`if` 、 `if { }`にあるコードがされます。

たとえば、このメソッドがのようにびされたとします`PrintPassOrFail(60);`このメソッドのは、「パス 」というコンソールプリントになります。 60のパラメータが50であるためです。

ただし、メソッドがのようにびされた `PrintPassOrFail(30);` 、メソッドのは「 ***Fail*** 」とされます。 。これは、30が50でないため、 `If`のわりに`else { }`のコードがされるため`else` 。

このでは、 スコアは100になるはずですが、これはまったくされていません。 100をえていないスコア、またはおそらく0をるスコアをするには、 **If-Else If-Else Statement**のをしてください。

If-Else If-Elseステートメント

If-Elseステートメントのにいて、 `Else If`ステートメントをするがました。 `Else If`ステートメントは、 **If-Else If-Else**の`If`ステートメントのにありますが、 `If`ステートメントとじをちます。これは、な**If-Else**ステートメントよりもくのブランチをコードにするためにされます。

If-Else Statementのでは、スコアは100までがるとされています。しかし、これにするはもなかった。これをするには、 **If-Else Statement**のメソッドをのようにします。

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

これらのはすべて、がたされるまでからにされます。このメソッドのこのしいアップデートでは、スコアのにする2つのしいブランチをしました。

たとえば、コードで`PrintPassOFail(110);`というメソッドをびしたとします`PrintPassOFail(110);`、は「***Errorscore is 100***よりきい」というConsole Printとなります。;`PrintPassOrFail(-20);`ようなコードでこのメソッドをびした`PrintPassOrFail(-20);`、は***Error***スコアが**0**よりさいとうでしょう。

スイッチ

switchをすると、のリストとのをテストできます。はケースとばれ、スイッチオンされているはスイッチのケースごとにチェックされます。

`switch`は、のにしてのなをテストする`if...else if... else..`、`if...else if... else..`よりもでできることがよくあります。

はのとおりです

```
switch(expression) {
   case constant-expression:
      statement(s);
      break;
   case constant-expression:
      statement(s);
      break;

   // you can have any number of case statements
   default : // Optional
      statement(s);
      break;
}
```

switchをしているときにするがあるものがあります

- switchでされるは、または、またはクラスがまたはへののをつクラスでなければなりません。
- スイッチには、いくつでもcaseをできます。それぞれののに、されるとコロンがきます。するは、switchででなければなりません。
- switchステートメントには、なデフォルトのをできます。デフォルトケースは、いずれのケースもてはまらないときにタスクをするためにできます。
- それぞれのケースは、のステートメントでないり、 `break`ステートメントでするがあります。その、はそののケースでされます。 `return`、 `throw`または`goto case`がされている`goto case`は、breakをすることもできます。

はなでえることができます

```
char grade = 'B';

switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent!");
        break;
    case 'B':
    case 'C':
        Console.WriteLine("Well done");
        break;
    case 'D':
        Console.WriteLine("You passed");
        break;
    case 'F':
        Console.WriteLine("Better try again");
        break;
    default:
        Console.WriteLine("Invalid grade");
        break;
}
```

ステートメントがブールおよびである

のステートメント

```
if (conditionA && conditionB && conditionC) //...
```

には

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

いえれば、ifののはのブールをするだけです。

をするのよくあるいは、 `true`と`false`とをにすること`true`。

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

これは、のようにきえることができます。

```
if (conditionA && !conditionB && conditionC)
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/3144/

142:

クラスとはなり、 `struct`はであり、 デフォルトではマネージヒープではなくローカルスタックにされます 。 つまり、のスタックがになると、 `struct`りてがされます。 りてられていない`struct`の`struct`も、 `struct`によってされていないとGCがすると、されます。

`struct` sはすることができず、のとなることもできません。にシール`protected`います。また、`protected`メンバーをめることもできません。しかし、クラスがうように、 `struct`はインタフェースをすることができます。

Examples

の

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct`インスタンスフィールドは、パラメータされたコンストラクタをして、または`struct`ににできます。
- プライベートメンバーはコンストラクターによってのみできます。
- `struct`は、System.ValueTypeからにされるをします。
- はのからすることはできませんが、インタフェースをできます。
- はりてにコピーされます。つまり、すべてのデータがしいインスタンスにコピーされ、そのいずれかのはのインスタンスにされません。
- は`null`でもかまいませんが、nullなとしてでき `null`。

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- は、 `new`をして、またはせずにインスタンスできます。

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

ただし、イニシャライザをするには、 `new`をするがあります。

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

は、いくつかのをいて、クラスができるすべてをできます。

- は、パラメータのないコンストラクタをすることはできません。 `struct`インスタンスフィールドは、パラメータされたコンストラクタをして、または`struct`ににできます。プライベートメンバはコンストラクタによってのみできます。
- はにシールされているため、メンバをprotectedとしてすることはできません。
- フィールドは、constまたはstaticのにのみできます。

の

コンストラクタで

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=0,Y=2,Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

コンストラクタなし

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

をコンストラクタとともにすると、りてフィールドにはじませんりてフィールドにはNULLがあります。

クラスとはなり、をするはありません。つまり、コンストラクタの1つをびすがないは、しいキーワードをするはありません。は、value-typeでnullにはなれないため、しいキーワードはありません。

をする

```
public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{
    public decimal Length { get; set; }
    public decimal Width { get; set; }

    public decimal Area()
    {
        return Length * Width;
    }
}
```

にがコピーされる

シンセは、すべてのデータがにコピーされるであり、しいコピーをしてものコピーのデータはされません。のコードスニペットは、 $_{p1}$が$_{p2}$ コピーされ、 $_{p1}$えられたが$_{p2}$インスタンスにしないことをしています。

```
var p1 = new Point {
    x = 1,
    y = 2
};
```

```
Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/778/

143: パターン

き

パターンは、オブジェクトとクラスがどのようにされ、きなをするかをし、エンティティのをするなをすることによってをにするパターンです。 7つのパターンがされています。らはのとおりですアダプタ、ブリッジ、コンポジット、デコレータ、ファサード、フライウェイト、プロキシ

Examples

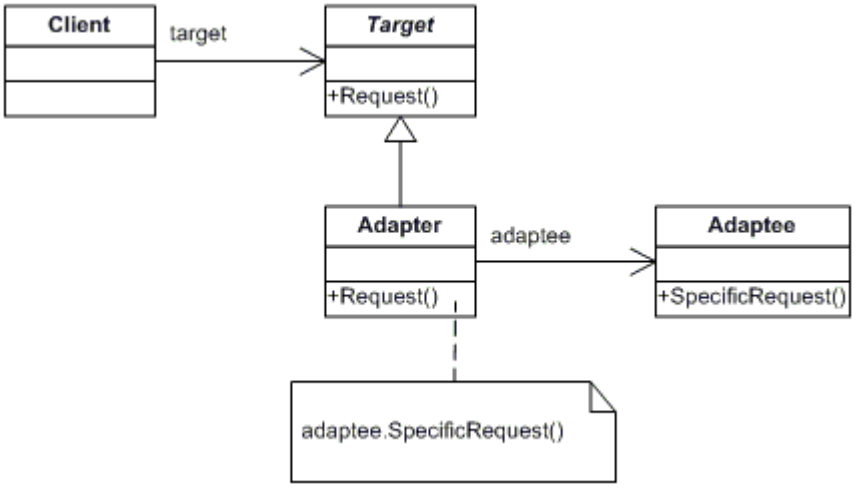
アダプタパターン

> として「アダプタ」がしているのは、にのない2つのインターフェイスがにできるようにするオブジェクトです。
>
> たとえば、あなたがIphone 8またはのAppleをするは、くのアダプタがです。デフォルトのインターフェイスはオーディオjacまたはUSBをサポートしていないためです。これらのアダプタでは、イヤホンをワイヤでることができます。また、のイーサネットケーブルをすることもできます。したがって、「にのない2つのインターフェイスはにします」。
>
> つまり、には、のことをします。クラスのインターフェースを、クライアントがするのインターフェースにする。アダプターは、のないインターフェースのためにクラスがにすることをにします。このパターンにするクラスとオブジェクトはのとおりです。

アダプターパターンは、4つの

1. **ITarget**これは、クライアントがをするためにするインターフェイスです。
2. **Adaptee**これは、クライアントがむですが、そのインタフェースはクライアントとがありません。
3. クライアントこれは、アダプターのコードをしていくつかのをしたいクラスです。
4. アダプタ ITargetをするクラスであり、クライアントがびたいAdapteeコードをびします。

UML



のコードのな。

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

2のコードの

```
/// <summary>
///  Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
```

```
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not
compatible with the client.
/// </summary>
public class CompanyEmplyees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];

        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee’s
code (list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a thirt party and you do'n have any control over it.
     * But it requires a Emplyee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the clietn list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("######### Employee List ##########");
        foreach (var item in employee)
        {
            Console.Write(item);
        }

    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code
which the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmplyees, ITarget
```

```
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}
```

いつするか

- システムが、それとのないのシステムのクラスをすることをする。
- にしたシステムとシステムとののをにする
- Ado.Net SqlAdapter、OracleAdapter、MySqlAdapterは、Adapter Patternののです。

オンラインでパターンをむ https://riptutorial.com/ja/csharp/topic/9764/パターン

144: プログラミング

Examples

とアクション

Funcは、パラメータされたのをします。のはであり、のはにりです。

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

アクションオブジェクトはvoidメソッドとていますので、タイプのみです。スタックにはありません。

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

はプログラミングではよくあり、オブジェクトプログラミングではまれです。

たとえば、のアドレスタイプをします。

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
}
```

コードののは、のオブジェクトののプロパティをするがあります。

に、のアドレスタイプをします。

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City  = city;
    }
}
```

みりのコレクションをつことは、をするものではないことにしてください。えば、

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

オブジェクトのユーザがコレクションをすることができるのでオブジェクトをまたはすることができるため、ではありません。にするには、IEnumerableなどのインターフェイスをするか、するメソッドをしないか、またはReadOnlyCollectionにするがあります。

```
public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }
}

List<Students> list = new List<Student>();
// add students
Classroom c = new Classroom(list.AsReadOnly());
```

オブジェクトの、のようながあります。

- それはのになりますのコードではできません。
- スレッドセーフです。
- コンストラクタは、のためののをします。
- オブジェクトをできないことがわかっているので、コードをしやすくなります。

NULLをける

Cはくのnullをします。 Fは、Optionタイプをっているので、そうではありません。オプション<>おそらくとして<>をむものでは、SomeとNoneがされます。これは、メソッドがnullレコードをすがあることをします。

えば、あなたはをんで、あなたがヌルをわなければならないかどうかをることはできません。

```
var user = _repository.GetUser(id);
```

なnullについてっていれば、それにするコードをすることができます。

```
var username = user != null ? user.Name : string.Empty;
```

Option <>がわりにされたらどうなりますか

```
Option<User> maybeUser = _repository.GetUser(id);
```

このコードでは、Noneレコードがされ、SomeまたはNoneをチェックするためのコードがであることがされています。

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

のメソッドは、Option <>をすをしています。

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Option <>のです。

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();
```

```
            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

のをすために、 avoidNull.csxはCREPLでできます。

のように、これはのです。 「」NuGetパッケージをすると、くのれたライブラリがします。

とは、のをとしてるか、をすですまたはその。

これは、LINQのWhereにをすなど、lambdaでにわれます。

```
var results = data.Where(p => p.Items == 0);
```

Whereには、かなりのをえるくのなるをけることができます。

メソッドをのメソッドにすことは、ストラテジーデザインパターンをするときにもられます。たとえば、のにじて、さまざまなソートをし、オブジェクトのSortメソッドにすことができます。

なコレクション

`System.Collections.Immutable` NuGetパッケージは、のコレクションクラスをします。

アイテムのと

```
var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

ビルダーをしてする

のなコレクションには、きなインスタンスをにするためにできる`Builder`クラスがあります。

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

のIEnumerableからする

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

すべてのなコレクションののリスト

- System.Collections.Immutable.ImmutableArray<T>
- System.Collections.Immutable.ImmutableDictionary<TKey,TValue>
- System.Collections.Immutable.ImmutableHashSet<T>
- System.Collections.Immutable.ImmutableList<T>
- System.Collections.Immutable.ImmutableQueue<T>
- System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>
- System.Collections.Immutable.ImmutableSortedSet<T>
- System.Collections.Immutable.ImmutableStack<T>

オンラインでプログラミングをむ https://riptutorial.com/ja/csharp/topic/2564/プログラミング

145: の

- `new Regex(pattern);` // されたパターンでしいインスタンスをします。
- `Regex.Match(input);` // ルックアップをし、*Match*をします。
- `Regex.Matches(input);` // をし、*MatchCollection*をします。

パラメーター

パターン	ルックアップにする`string`パターン。 msdn
RegexOptions [オプション]	ここでのなオプションは`Singleline`と`Multiline`です。 `Multiline-Mode` `SingleLine-Mode` `Multiline-Mode`なく、 `SingleLine-Mode` `Multiline-Mode`での `NewLine` \ nをカバーしないドット。のようなパターンのをしてい `SingleLine-Mode`。 の msdn
タイムアウト[オプション]	パターンがよりになるところでは、はよりくのをするがあります。これは、ネットワークプログラミングでられているように、ルックアップにされたタイムアウトです。

するがあります

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

いいね

- ここでをるためにソリューションをコンパイルするなく、パターンをオンラインでテストできます Click me
- Regex101 をクリックしてください

は、パワフルで、なテキストベースののためのなにいるため、でをにするがあります。これは、*`XmlDocument`*ようなこのタスクのためににしたクラスがするがあるかどうかをねることなしに、々がで*XML*をしようとするポイント*`XmlDocument`*。

はさをりくのでなければなりません。なくとも20のパターンをきめるに*`right way`*をすをれることはなくともありません。

Examples

シングルマッチ

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

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

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

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

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

の

using System.Text.RegularExpressions;

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

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

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

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

オンラインでのをむ https://riptutorial.com/ja/csharp/topic/3774/の

146: ラムダクエリビルダ

クラスは`ExpressionBuilder`とばれます。それは3つのをする

```
 private static readonly MethodInfo ContainsMethod = typeof(string).GetMethod("Contains",
new[] { typeof(string) });
 private static readonly MethodInfo StartsWithMethod = typeof(string).GetMethod("StartsWith",
new[] { typeof(string) });
 private static readonly MethodInfo EndsWithMethod = typeof(string).GetMethod("EndsWith",
new[] { typeof(string) });
```

ラムダをす1つのパブリックメソッド`GetExpression`、および3つのプライベートメソッド

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

すべてのメソッドについては、でしくします。

Examples

QueryFilterクラス

このクラスは、フィルタをします。

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

をする

```
 public enum Operator
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
```

```
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

GetExpressionメソッド

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param
part
    // To create a ParameterExpression need the type of the entity that the query is against
an a name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good parctice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put I in the query two at a time
            while (filters.Count > 0)
            {
                // Retreive the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration
finds the next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
                if (filters.Count == 1)
                {
                    exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
                    filters.RemoveAt(0);
                }
            }
```

```
        }
    }
    else
        // It is result from direct call.
        exp = GetExpression<T>(param, filters[0]);

           // converts the Expression into Lambda and retuns the query
    return Expression.Lambda<Func<T, bool>>(exp, param);
}
```

GetExpressionプライベートなオーバーロード

1つのフィルタの

ここでクエリがされ、パラメータとフィルタをけります。

```
private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for
example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
            return Expression.Call(member, StartsWithMethod, constant);

        case Operator.EndsWith:
            return Expression.Call(member, EndsWithMethod, constant);
    }
```

```
    return null;
}
```

2つのフィルタの

これは、なのわりにBinaryExpresionインスタンスをします。

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter
filter1, QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

ConstantExpression メソッド

`ConstantExpression`は、じタイプの`MemberExpression`でなければなりません。 こののは、`ConstantExpression`インスタンスをするにされるです。

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if(type == typeof(string))
    {
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
        }
        constant = Expression.Constant(flag);
    }
    else if (type == typeof(decimal))
    {
        decimal number;
```

```
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

コレクションフィルタ=しいList; QueryFilter filter =しいQueryFilter ""、 "バ—ガ—"、 Operator.StartsWith; filters.Addフィルター;

```
 Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

この、Foodエンティティにするクエリで、そのに "Burger"でまるすべてのをしたいとします。

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

オンラインでラムダクエリビルダをむ https://riptutorial.com/ja/csharp/topic/6721/ラムダクエリビルダ

147:

き

Cでは、 はまたはの1つのオペランドにされるプログラムです。インクリメント++やnewなどの1つのオペランドをとるは、とばれます。+、 - 、*、/などの2つのオペランドをるは、2とばれます。 1つの、き:)は3つのオペランドをとり、Cの3つの3です。

- public static OperandTypeoperatorSymbolOperandType operand1
- パブリックstatic OperandTypeoperatorSymbolOperandType operand1、OperandType2 operand2

パラメーター

パラメータ	
operatorSymbol	オペレータがになっている、えば+、 - 、/、*
オペランドタイプ	オーバーロードされたによってされる。
オペランド1	をするにされるのオペランド。
オペランド2	バイナリをうときにをするにされる2オペランド。
ステートメント	をすにをするためになオプションコード。

すべてのは`static methods`としてされ、 `virtual`ではなく、されません。

オペレータの

すべてのオペレータは、オペレータがどのグループにしているかじグループのオペレータがじをつにじて、の「」をっています。のはののにされることをします。にすのは、もいにべえられたグループそれぞれのをむのリストです。

-
  - `ab` - メンバーアクセス。
  - `a?.b` - Nullきメンバーアクセス。
  - `->` - メンバーアクセスとみわされたポインタ。
  - `f(x)` - びし。
  - `a[x]` - インデクサー。
  - `a?[x]` - Nullきインデクサ。
  - `x++` - Postfixインクリメント。
  - `x--` - デクリメント。

    - `new` - インスタンスの。
    - `default(T)` - タイプ`T`デフォルトのされたをします。
    - `typeof` - オペランドの`Type`オブジェクトをします。
    - `checked` - のオ―バ―フロ―チェックをにします。
    - `unchecked` - のオ―バ―フロ―チェックをにします。
    - `delegate` - デリゲ―トインスタンスをしてします。
    - `sizeof` - オペランドのサイズをバイトでします。

-
    - `+x` -りの`x`。
    - `-x` - の。
    - `!x` - 。
    - `~x` -ビットの/デストラクタをします。
    - `++x` - プレフィックスインクリメント。
    - `--x` - プレフィックスの。
    - `(T)x` - キャスティング。
    - `await` - `Task`つ。
    - `&x` -のアドレスポインタをします`x`。
    - `*x` - ポインタ。

-
    - `x * y` - 。
    - `x / y` - 。
    - `x % y` - モジュラス。

-
    - `x + y` - 。
    - `x - y` - 。

- ビットシフト

    - `x << y` - にシフトビット。
    - `x >> y` - にシフトビット。

- リレ―ショナル/タイプテスト

    - `x < y` - よりさい。
    - `x > y` - よりきい。
    - `x <= y` - よりさいかしい。
    - `x >= y` - よりきいかしい。
    - `is` - 。
    - `as` -

-
    - `x == y` - 。
    - `x != y` - しくない。

- **AND**

    - `x & y` - /ビットAND。

- **XOR**
  - `x ^ y` - /ビットごとの。
- **OR**
  - `x | y` - /ビットごとのOR。
- きAND
  - `x && y` - ANDをします。
- きOR
  - `x || y` - ORをします。
- ヌル
  - `x ?? y` - nullでないは`x`します。それのは`y`します。
- き
  - `x ? y : z` - `x`がtrueのは`y`/す。そうでなければ、 `z`する。

のあるコンテンツ

- ヌル
- Nullき
- の

Examples

オーバーロード

Cでは、 `operator`キーワードをしてメンバーをすることにより、ユーザーでをオーバーロードすることができます。
のは、 `+`のをしています。

をす`Complex`クラスがある

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

このクラスに`+`をするオプションをします。すなわち

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

クラスの`+`をオーバーロードするがあります。これは、と`operator`キーワードをしてわれます。

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

`+`、`-`、`*`、`/`などのはすべてオーバーロードされます。これにはじをさないもまれますたとえば、`==`と`!=`はブールをすにもかかわらずオーバーロードされますここでは、ペアにするルールもされます。

は、ペアでオーバーロードするがあります`<`がオーバーロードされているは、`>`もオーバーロードするがあります。

オーバーロードなおよびオーバーロードなといくつかのオーバーロードなにされたのリストは、MSDN - オーバーロードCプログラミングガイドをしてください 。

7.0

`operator is`オーバーロードはC7.0のパターンマッチングメカニズムでされました。については、パターンマッチングをしてください。

のようにされた`Cartesian`

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

オーバーロードな`operator is`えば、`Polar`

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

このようにすることができます

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

このは[Roslyn Pattern Matching Documentation]からったものです

しい

されたオペランドがしいかどうかをチェックします。

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

Javaとはなり、はネイティブにでします。

は、のキャストがからへする、なるタイプのオペランドでします。ななキャストがしないは、なキャストをびすか、のあるにするメソッドをできます。

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

Visual Basic.NETとはなり、はとじではありません。

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

= としないでください。

の、のオペランドのがしい、 `true` します。
のために、オペレータは、り`true`のオペランドが ないにしい。ただし、オブジェクトはのしいものとされます。

しくない

されたオペランドがしくないかどうかをチェックします。

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
```

```
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

このは、 == のとにのをします

よりきい

のオペランドが2のオペランドよりきいかどうかをチェックします。

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

のオペランドが2のオペランドよりさいかどうかをチェックします。

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.

var x = 12;
var y = 22;
x < y    //Returns true.
y < x    //Returns false.
```

しいよりきい

のオペランドが2のオペランドとしいかどうかをチェックします。

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

しくない

のオペランドが2のオペランドとしいかどうかをチェックします。

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

、は、*1*オペランドがのなをできないにのみ、*2*オペランドをする。

それはあなたが*firstCondition && secondCondition*として&&をしている、それは*firstCondition*がで

あるにのみ、secondConditionをし、なが*firstOperand*と*secondOperand*のがtrueにされているにのみtrueになりますofcourceだろう、ということをします。これはくのシナリオでです。たとえば、リストに3つのがありますが、リストが*NullReferenceException*にされないようにされているかどうかをチェックするがあります。あなたはこれをのようにすることができます

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mLList = nullがたされるまで、 *mList.Count> 3*はチェックされません。

AND

`&&`は、ブーAND `&` ののです。

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

OR

`||`ブーOR `|` ののです。

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

のサイズ

バイトの*のサイズをする`int`をします。

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
```

```
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

* なコンテキストでのみ、のプリミティブをサポートします。

でないコンテキストでは、 `sizeof`をしてのプリミティブやのサイズをすことができます。

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

の オ―バ―ロ―ド

のオ―バ―ロ―ドだけではです。なるでは、のすべてをびすことができます

1. `object.Equals`と`object.GetHashCode`
2. `IEquatable<T>.Equals` オプション、ボクシングをけることができます
3. `operator ==`および`operator !=` オプション、のを

`Equals`オ―バ―ライドする、 `GetHashCode`もオ―バ―ライドするがあります。 `Equals`する、なケ―スがあります。なるタイプのオブジェクトとの、とのなどです。

オ―バ―ライドされないには`Equals`をし、 `==`は、クラスやのためにがなります。クラスの、はされ、の、プロパティのはリフレクションをしてされ、パフォ―マンスにをえることがあります。 `==`は、オ―バ―ライドされないり、のにはできません。

に、はのにわなければなりません

- をスロ―することはできません。
- にしいのためにではないかもしれない`A A NULL`いくつかのシステムでの。
- トランジット `A`が`B`にしく、 `B`が`C`にしいならば、 `A`は`C`しい。
- `A`が`B`にしい、 `A`と`B`はしいハッシュコ―ドをつ。
- ツリ―のは`B`と`C`のインスタンスである`Class2`からされた`Class1 Class1.Equals(A,B)` 、にへのびしとじをすがあります`Class2.Equals(A,B)`

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
```

```
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

クラスメンバ—オペレ—タ—メンバ—アクセス

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

クラスメンバ—Nullきメンバ—アクセス

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

クラスメンバ—びし

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

クラスメンバオブジェクトのけ

```
var letters = "letters".ToCharArray();
char letter = letters[1];
```

```
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

クラスメンバNullきインデックス

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

「」または「」

"exclusive or"いXORのはのとおりです。^

このは、されたboolのうち1つだけがであるにtrueをします。

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

ビットシフト

シフトをすると、プログラマは、ビットをすべてまたはにシフトすることでをできます。のは、を1にシフトしたときのをしています。

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

シフト

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

のキャストとなキャスト

Cでは、 `explicit`および`implicit`キーワードをして、ユーザーのでりておよびキャストをできます。メソッドのシグネチャはのをとります。

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

このメソッドはそれをることはできませんし、インスタンスメソッドにすることもできません。ただし、でされたのプライベートメンバにアクセスすることはできます。

`implicit` `explicit`キャストと`explicit`キャストのの

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
}
```

のキャストをする

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

キャストがあなたのタイプからくとあなたのタイプにく、のをかせることができます。

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

に、のキャストにするがある`as`キーワードは、このではです。 `explicit`または`implicit`キャストのいずれかをしたでも、のことはできません。

```
ColorImage cm = myBinaryImage as ColorImage;
```

コンパイルエラーがします。

をつ2

Cには、のをし、そのをのにするために、 `=`とみわせることができるのがあります。

```
x += y
```

とじです

```
x = x + y
```

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<=
- >>=

ブーールのにじて2つののいずれかをします。

```
condition ? expression_if_true : expression_if_false;
```

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

はであり、をすることができます。これは、のまたはののいずれかにのをえることによってわれる。みやすくするためにするがありますが、これはによってはするとです。

このでは、は`clamp`をし、にあるはのをし、のは`min`をし、をえているは`max`をします。

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

は、のようにネストすることもできます。

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

3ステートメントをするときは、みやすさをさせるためにやインデントをするのがです。

*expression_if_true*と*expression_if_false*のはでなければならないか、またはからへののがです。

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

タイプとコンバ―ジョンのは、あなたのクラスにもされます。

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit
conversion from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough
to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate
to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

タイプ

の`System.Type`オブジェクトをし`System.Type`。

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

のをするには、 `GetType`メソッドをして、のインスタンスの`System.Type`をします。

`typeof`は、コンパイルにされるパラメ―タとしてをとります。

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
```

```
Assert.IsTrue(animal is Animal);                    // pass, animal implements Animal
```

デフォルト

のT

char 、 int 、 floatなどのみみプリミティブデータ、 struct 、 enumされたユーザー。デフォルトはnew T()です。

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Tクラス

のclass 、 interface 、またはデリゲート。デフォルトはnullです。

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

の

variable 、 type 、またはmemberのをすをします。

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

nameofはC6.0でされました。コンパイルにされ、されたはコンパイラによってインラインでされるため、をできるほとんどのにできます switch、アトリビュートなどのcaseラベルなど。 。これは、や、MVCアクションリンクなどをさせたりロギングするなどにです。

Null

6.0

C6.0でされたNull?. NullReferenceExceptionをスローするのではなく、のがnullとされたは、すぐにnullしnull 。 そのがnullとされる、それはのようにわれ.オペレーター。 nullすがあるので、そ

のりのはにNULLです。これは、またはプリミティブの、 `Nullable<T>`ラップされることをします。

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

これは、イベントをさせるときにです。は、ifステートメントでイベント・コールをラップして`null`をチェックし、そのイベントをさせるがあります。これにより、がするがあります。 Nullをすると、のようにできます。

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

PostfixとPrefixのインクリメントとデクリメント

Postfixインクリメント`x++`は`x`に`1`をえます

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

デクリメント`x--`は1をく

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x`はインクリメントとばれ、xのをインクリメントしてからxをし、 `x++`は`x++`のをし、インクリメントします

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

while

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

どちらもforループでにされます

```
for(int i = 0; i < 10; i++)
{
}
```

=>ラムダ

3.0

=>は、=とじをち、です。

ラムダをするためにされ、 LINQクエリでもくされています。

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

LINQのやクエリでする、、オブジェクトのはコンパイラによってされるのでスキップできます。

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

ラムダのなはのとおりです。

```
(input parameters) => expression
```

ラムダのパラメータは、 =>のにされ、されるの//ブロックはのにあります。

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

このをすると、なメソッドをすることなく、デリゲートをにできます。

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

のわりに

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

'='

=オペランドのをオペランドのにし、そのをします。

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

??ヌル

ヌル?? NULLでないときはをします。 nullのは、をします。

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

??はチェインされ、 ifチェックのがです。

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/18/

148: の

き

nameofをすると、 、 またはメンバのをリテラルとしてハードコードすることなく、でできます。

はコンパイルにされます。つまり、IDEのをしてののをすることができ、がされます。

-

Examples

ないの

nameofをすると、、またはメンバのをリテラルとしてハードコードすることなく、でできます。
はコンパイルにされます。つまり、IDEのをして、されたをし、をそのでします。

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

するか

myString

のが "myString"であるためです。をリファクタリングするとがされます。

でびされた、 nameofは、になるオブジェクトのまたはではなく 、ののをします。えば

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

パラ メータの

スニペット

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

コンソール

paramValue

PropertyChanged イベントの

スニペット

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

コンソール

PropertyChanged イベントの

スニペット

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...
```

```
private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

コンソール

タイトルがすべてにわった

ステータスがShowStopperにされました

ジェネリックパラメータにされます。

スニペット

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}

...

var myClass = new SomeClass<int>();
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

コンソール

TItem

SomeClass

されたにされる

スニペット

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

コンソール

- MyNamespace
- のクラス
- MyNestedClass
- MyStaticProperty

チェックとガード

む

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

オーバー

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

`nameof`フィーチャをすると、メソッドパラメータをにリファクタリングできます。

くけされた**MVC**アクションリンク

のやかなけのわりに

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

くけされたアクションリンクをできるようになりました

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

あなたのコーードをリファクタリングし、をしたいは`UserController.LogIn`にメソッドを`UserController.SignIn`、あなたはすべてののををするはありません。コンパイラがそのをいます。

オンラインでのをむ https://riptutorial.com/ja/csharp/topic/80/の

149:

Cをした**Win32 API**の

Windowsは、Win32 APIのでくのをしています。これらのAPIをすると、ウィンドウでき、アプリケーションのパフォーマンスがします。

WindowsはいAPIをしています。さまざまなAPIにするをるには、ピンボケのようなサイトをチェックできます。

Examples

アンマネージC ++ DLLからのインポート

に、アンマネージC ++ DLLでされているをインポートするのをします。 "myDLL.dll"のC ++ソースコードでは、`add`がされています。

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

その、のようにCプログラムにみむことができます。

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

`extern "C"`と`__stdcall`がなについては、 `extern "C"` とC ++ネームマングリングの びし 」をしてください。

ライブラリの

externメソッドがにびされると、Cプログラムはなdllをしてロードします。 DLLのをすると、にどのようにするかについては、 このstackoverflowのをしてください。

comのクラスをするなコード

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

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }

        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

C ++ネームマングリング

C ++コンパイラは、なるをつオーバーロードをにするために、などのエクスポートされたのにをエンコードします。このプロセスをネームマングリングといいます。これは、のとして、Cのとにはのとののをインポートするとがする`int add(int a, int b)`は、もはや`add`、それはすることができ、`?add@@YAHHH@Z`、`_add@8`コンパイラおよびびしにじて、`_add@8`またはそののものをします。

のマングリングのをするはいくつかあります。

- `extern "C"`をしてをエクスポートして、Cのマングリングをするcリンケージにりえます。

  ```
  extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
  ```

```
[DllImport("myDLL.dll")]
```

はまだされますが `_add@8` 、 `StdCall` + `extern "C"`のマングリングはCコンパイラによってされます。

- `myDLL.def`モジュールファイルにエクスポートされたをする

```
EXPORTS
  add
```

```
int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

この、はな`add`になります。

- されたをインポートする。されたをするには、いくつかのDLLビューアがです。にすることができます

```
__declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

びし

コールのいくつかのがあります。びしびしまたはびしがスタックからをりす、をす、をするがあります。 C ++ではデフォルトで`Cdecl`びしがされていますが、CではWindows APIでされる`StdCall`がです。あなたはどちらかをするがあります

- C ++のびしを`StdCall`にする

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- または、Cのびしを`Cdecl`にします。

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

`Cdecl`びしとされたのをする、コードはのようになります。

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** **__thiscall** はにクラスのメンバーであるでされます。
- が**thiscall** **__thiscall** をすると、クラスへのポインタがのパラメータとしてされます。

アンマネージDLLのロードとアンロード

`DllImport`をするときは、コンパイルにしいdllとメソッドをっておくがあります。あなたがよりになり、ロードするDLLとメソッドをにしたいは、Windows APIメソッド`LoadLibrary()`、`GetProcAddress()`、および`FreeLibrary()`をできます。これは、するライブラリがのにするにちます。

`GetProcAddress()`によってされたポインタは、`Marshal.GetDelegateForFunctionPointer()`をしてデリゲートにキャストできます。

のコードサンプルは、のの`myDLL.dll`これをしています。

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module);  // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
typeof(AddDelegate));
```

```
        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

Win32エラーの

interopメソッドをするは、 **GetLastError** APIをして、APIびしにするをできます。

DllImportSetLastError

SetLastError = true

びしがSetLastErrorWin32 APIをびすことをします。

SetLastError = false

びしが SetLastErrorWin32 APIをびさないことをします。したがって、エラーはされません。

- SetLastErrorがされていない、falseデフォルトにされます。
- Marshal.GetLastWin32Errorメソッドをしてエラーコードをできます。

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

しないmutexをオープンしようとすると、GetLastErrorは**ERROR_FILE_NOT_FOUND**をします。

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

システムエラーコードはここにあります

https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx

GetLastError API

ネイティブの**GetLastError** APIもあります。

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- マネージコードからWin32 APIをびすときは、に**Marshal.GetLastWin32Error**をするがあります。

はのとおりです。

エラーをし、あなたのWin32コールSetLastErrorをびしのに、CLRは、のWin32がびすことができ**SetLastError**としても、このがあなたのエラーをきすることができたびしをびすことができます。このシナリオでは、 **GetLastError**をびすとなエラーがします。

SetLastError = trueをすると、のWin32びしをするにCLRがエラーコードをすることをします。

オブジェクト

GC ガベージコレクターはゴミをするがあります。

GCはゴミをしながら、されていないオブジェクトをのヒープからし、ヒープのをきこします。**GC**は、をったときに、ヒープのオブジェクトのをうヒープdefusionintationをします。

GCはではないので、オブジェクト/ポインタをネイティブコードにすとき、 **GC**はいつでもできます.Ineropびしのにすると、オブジェクトネイティブにされたがされたヒープですることができます。その、でながされます。

このシナリオでは、ネイティブコードにすにオブジェクトをするがあります 。

オブジェクト

されたオブジェクトは、GCによってできないオブジェクトです。

Gcピンめハンドル

Gc.Allocメソッドをしてピンオブジェクトをできます

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- オブジェクトにされた**GCHandle**をすると、ハンドルをするまで、 **GC**によってできないオブジェクトとしてのオブジェクトがマークされます

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finaly
    {
```

```
        handle.Free()
    }
}
```

- ピンニングにきなものオブジェクトは、ヒープのをするため、された**GcHandle**をできるだけくしようとします。
- あなたが**GcHandle**をするのをれるなら、もしません。なコードセクションでしますなものなど

マーシャルでをむ

Marshalクラスには**PtrToStructure**というのがまれています。このは、アンマネージポインタでをみることができます。

PtrToStructureにはくのオーバーロードがありますが、それらはすべてじをっています。

な**PtrToStructure**

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - 。

ptr - されていないメモリブロックへのポインタ。

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- ネイティブをみながらオブジェクトをう、オブジェクトをすることをれないでください:)

```
T Read<T>(byte[] buffer)
    {
        T result = default(T);

        var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

        try
        {
            result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
        }
        finally
        {
            gch.Free();
        }

        return result;
    }
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/3278/

150: しいとGetHashCode

Equalsでは、のをたすがあります。

- オブジェクトはそれとじでなければならない。
  x.Equals(x)はtrueしtrue。
- xとy、またはyとxをxとすると、いはありません。はじです。
  x.Equals(y)はy.Equals(x)とじをします。
- 1つのオブジェクトがのオブジェクトとしく、このオブジェクトが3のオブジェクトにしい、のオブジェクトは3のオブジェクトにしくなければなりません。
  if (x.Equals(y) && y.Equals(z))がtrueしたtrue、 x.Equals(z)はtrueしtrue。
- オブジェクトをのものともすると、はにじになります。
  するx.Equals(y)びしは、xとyによってされるオブジェクトがされていないり、じをします。
- **null**との オブジェクトはnullはなりません。
  x.Equals(null)はfalseしfalse。

GetHashCode

- **Equals**があります2つのオブジェクトがしい Equalsがtrueをす、 GetHashCode はそれぞれのオブジェクトにしてじをすがあります。
- い2つのオブジェクトがしくない Equalsう、それらのハッシュコードがであるいがすべきです。なのがられているため、 なハッシングはです。
- **Cheap** すべてのケースでハッシュコードをするのはでなければなりません。

オーバーロードのためのガイドラインEqualsとオペレータ==

Examples

デフォルトはとじです。

Equalsは Objectクラスでされています。

```
public virtual bool Equals(Object obj);
```

デフォルトでは、 Equalsはのとおりです。

- インスタンスがのは、がじにのみ、 Equalsはtrueをします。
- インスタンスがの、とがじにのみ、 Equalsはtrueをします。

- `string`はなケーースです。それはのようにします。

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

い**GetHashCode**オーバーライドをく

`GetHashCode`はDictionary <>およびHashTableにきなパフォーマンスをもたらします。

い`GetHashCode`メソッド

- なをつべきである
  - すべてのは、ランダムなインスタンスのりとほぼじになるはずです
  - メソッドごとにインスタンスごとにじ '999'がされると、パフォーマンスがします
- くなければならない
  - これらは、さがであるハッシュではありません
  - ハッシュがいほど、のがくなります
- `Equals`される2つのインスタンスでじHashCodeをすがあります
  - `GetHashCode`がをすなどので、 `List` 、 `Dictionary`などにアイテムがつからないがありま

す。

GetHashCodeをするためのいは、として1つのをし、のでされたのフィールドのハッシュコードをそれにすることです。

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Equalsメソッドでされるフィールドのみがハッシュにされます。

Dictionary / HashTablesのなるでじをうがあるは、IEqualityComparerをできます。

カスタムでのEqualsとGetHashCodeのオーバーライド

Personクラスの

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

しかし、のようにEqualsとGetHashCodeをする

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }

    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }
```

```
    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

また、LINQをしてにしてさまざまなクエリをすると、 `Equals`と`GetHashCode`がチェックされます。

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

IEqualityComparatorのEqualsとGetHashCode

えられた`Person`

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

しかし、 `Equals`と`GetHashCode`を`IEqualityComparator`する

```
public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when
comparing two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
```

```
Count = 2
```

このクエリでは、 `Equals`がtrueをし、 `GetHashCode`が2のユーザーにしてじハッシュコードをした、2つのオブジェクトがしいとみなされていることにしてください。

オンラインでしいとGetHashCodeをむ https://riptutorial.com/ja/csharp/topic/3429/しいと-gethashcode

151:

Examples

cとのの

Cでは、とのという2つのなるのがあります。のは、にされるのです。つまり、2つのオブジェクトにじがまれていることをします。たとえば、2のをつ2つのは、のしいをちます。とは、するオブジェクトが2つしないことをします。わりに、2つのオブジェクトがあり、どちらもじオブジェクトをします。

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

されたの、==は、そのオペランドのがしいはtrueをし、そうでないはfalseをします。 stringのの、==は、2つのオペランドがじオブジェクトをするにtrueをします。の、==はのをします。

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1491/

152: みみのエイリアス

Examples

みみテーブル

のは、みみのC#のキーワードをしています。これらのは、システムののみのエイリアスです。

Cタイプ	.NET Frameworkの
ブール	System.Boolean
バイト	System.Byte
sbyte	System.SByte
チャー	System.Char
の	System.Decimal
ダブル	System.Double
く	System.Single
int	System.Int32
uint	System.UInt32
いです	System.Int64
ulong	System.UInt64
オブジェクト	System.Object
ショート	System.Int16
ushort	System.UInt16
	System.String

C#タイプのキーワードとそのエイリアスはがあります。たとえば、のいずれかのをしてをできます。

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

オンラインでみみのエイリアスをむ https://riptutorial.com/ja/csharp/topic/1862/みみのエイリアス

153:

- クラスDerivedClassBaseClass
- クラスDerivedClassBaseClass、IExampleInterface
- クラスDerivedClassBaseClass、IExampleInterface、IAnotherInterface

クラスは1つのクラスからすることができますが、わりに、またはに1つのインタフェースをできます。

はインタフェースをできますが、どのからもにすることはできません。それらは`System.ValueType`からにされ、 `System.Object`からします。

クラスはインターフェイスをできません 。

Examples

クラスから

コードのをけるには、なクラスのなメソッドとをベースとしてします。

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

は`Animal`をすクラスがありますので、ののをすクラスをできます

```
public class Cat : Animal
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
```

```
}
```

Catクラスは、そのににされているメソッドだけでなく、な`Animal`クラスでされているすべてのメソッドにもアクセスできます。どのであろうとなかろうとはべることができます。しかし、それがでもないり、はスクラッチすることができませんでした。その、のをするのクラスをすることができます。 Gopherのような、のとSlothをするは、なをたない。

クラスからし、インタフェースをする

```
public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

クラスからし、のインタフェースをする

```
public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
```

```
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

のテストと

```
interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name);  //BaseClass
Console.WriteLine(baseType.BaseType.Name);  //Object
Console.WriteLine(typeof(object).BaseType);  //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance));  //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance));  //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface

Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

クラスの

のためのとしてできるインタフエースとはなり、クラスはのためののをたします。

クラスをインスタンスすることはできません。するがあり、のクラスまたはクラスをインスタン

スできます。

クラスはをするためにされます

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

のは、Carをするすべてのクラスが、でにHonkHornメソッドをけるをしています。これは、しいカーをするは、どのようにそれがホーンをどのようにらすかするはないことをします。

サブクラスのコンストラクタ

あなたは、クラスのサブクラスをするときは、してクラスをすることができます：`base`サブクラスのコンストラクタのパラメータのに。

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;

    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

。コンストラクタのびしシーケンス

クラスの`Dog`をつ`Animal`クラスがあるとえてください

```
class Animal
{
```

```
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

デフォルトでは、すべてのクラスはに`Object`クラスをします。

これはのコーードとじです。

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

`Dog`クラスのインスタンスをするとき、クラスののコンストラクタへのなびしがない 、 クラスのデフォルトのコンストラクタパラメーータなしがびされます。たちの、は`Object's`コンストラクタ、に`Animal's`、そしての`Dog's`コンストラクタとばれます。

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

は

のコンストラクタで
のコンストラクタで

デモをる

のコンストラクタをにびします。

のでは、 `Dog`クラスのコンストラクタが`Animal`クラスのデフォルトのコンストラクタをびします。にじて、びすコンストラクタをできます。つまり、クラスでされているコンストラクタをびすことができます。

これら2つのクラスがあるとえてください。

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

ここにはがありますか

クラスには2つのコンストラクタがあります。

baseどういうですか

baseはクラスへのです。たちの、このようなDogクラスのインスタンスをするとき

```
Dog dog = new Dog();
```

ランタイムはまず、パラメータのないコンストラクタであるDog()びします。しかし、そのはすぐにはしません。コンストラクタのかっこのに、 base()というようなびしがあります。つまり、デフォルトのDogコンストラクタをびすと、のデフォルトコンストラクタがびされます。のコンストラクタがされると、 Dog()コンストラクタがされ、にされます。

はのようになります

    のデフォルトコンストラクタ
    のデフォルトコンストラクタ

デモをる

Dog'sコンストラクタをパラメータでびすとどうなりますか

```
Dog dog = new Dog("Rex");
```

プライベートではないクラスのメンバーは、クラスによってされます。つまり、 `Dog`は`name`フィールドもあります。
このケースでは、コンストラクタにをしました。は、をクラスのコンストラクタにして、 `name`フィールドをします。

は

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

すべてのオブジェクトのはクラスからされます。では、にあるクラスがされます。すべてのクラスは`Object`からしているため、 `Object`にびされるのコンストラクタは`Object`クラスのコンストラクタです。に、チェーンののコンストラクタがびされ、それらのすべてがびされたでのみ、オブジェクトがされます

ベースキーワード

1. baseキーワードは、クラスからクラスのメンバーにアクセスするためにされます。
2. のメソッドでオーバーライドされたクラスのメソッドをびします。クラスのインスタンスをするときにびすベースクラスコンストラクタをします。

するメソッド

メソッドのにはいくつかのがあります

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }

    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
```

```
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

アンチパターン

な

2つのクラスのクラス`Foo`と`Bar`ます。 `Foo`は`Do1`と`Do2` 2つのがあります。 `Bar`は`Foo` `Do1`をするがありますが、 `Do2`とせず、 `Do2`とのがですが、まったくなるをとします。

い `Foo`バーチャルで`Do2()`をし、 `Bar`それをオーバーライドするか、 `Do2()` `Bar`に`throw Exception`を`throw Exception`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

い

`Foo`から`Do1()`をりし、しいクラス`Baz`れ、 `Baz`から`Foo`と`Bar`をし、 `Do2()`々にします

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
```

```
    }
}
```

なぜのがいのですか2はいですnr2が`Foo`でをうがある、 `Bar`は`Foo`からりせないので、 `Bar`をるがあります。のでうと、 `Foo`と`Bar`は`Baz`され、いにしませんそうすべきではありません。

をつクラス

をつクラスのの。ノ―ドには、1つのノ―ドとのノ―ドがあります。

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }
    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }
    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}
```

オブジエクトのツリ―をするがあるたびにをすることができます。ツリ―のノ―ドオブジエクトは、ベ―スクラスからするがあります。

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

ノ―ドクラスはのどこにオブジエクトがあるのか、オブジエクトがであるかをっています。いくつかのみみは、 `Control`や`XmlElement`などのツリ―をし、の`Tree<T>`はコ―ドののののクラスとしてできます。

たとえば、 すべてののウエイトからウエイトをするパ―ツのをするには、のようにします。

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); }
}
}
```

としてする

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//    \
//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

もう1つのは、フレームのにあります。この、フレームののは、 すべてのフレームのにします。

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null,
PointF.Empty) { };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
```

```
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}
```

としてする

```
// Define the following coordinate system hierarchy
//
// o--> [A1] --+--> [B1] -----> [C1]
//             |
//             +--> [B2] --+--> [C2]
//                         |
//                         +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/29/

154:

き

は、のデータでされたです。ツリーのノードはのであり、はコードです。ラムダのインメモリは、クエリののすなわちコードをするが、そのはしないツリーになります。ツリーは、ラムダのをかつにする。

- <TDelegate> name = lambdaExpression;

パラメーター

パラメータ	
TDelegate	にされるデリゲート
ラムダ	ラムダ `num => num < 5`

の

たちがたところ

ツリーは、すべてに "ソースコード"をします。`decimal CalculateTotalTaxDue(SalesOrder order)`づいてをするをえてみましょう。 .NETプログラムでそのメソッドをするのはです - に`decimal taxDue = CalculateTotalTaxDue(order);`。 リモートクエリSQL、XML、リモートサーバなどのすべてのにするはどうすればよいですかこれらのリモートクエリソースはメソッドをびすことはできませんには、これらすべてのケースでフローをさせるがあります。クエリをメモリにし、をループしてごとにをします。

フローインバージョンのメモリとレイテンシのをする

ツリーは、ノードがをするツリーのデータです。それらは、データベースクエリのようなプログラムのでできるでコンパイルされたデータをフィルタリングするためにされるメソッドのようなをするためにされます。

ここでのは、リモートクエリがたちのメソッドにアクセスできないことです。わりに、メソッドのをリモートクエリにした、このをできます。 `CalculateTotalTaxDue`では、このをします。

1. をするをする
2. のすべてのをループする
3.

について、がかどうかをする
4. そうであるは、にをけ、そのをにします
5. それはもしない

これらので、リモートクエリはデータをするときにをできます。

これをするには2つのがあります。コンパイルされた.NETメソッドをのリストにすると、リモートシステムでできるでをどのようにフォーマットするのですか

がなければ、MSILののをすることしかできませんでした。 MSILは.NETコンパイラによってされたアセンブラのようなコードです。MSILのはですが 、ではありません。たとえそれをしくしたとしても、のプログラマーのがのルーチンであったかどうかをするのはしいでしょう。

ツリーはそのをする

ツリーは、これらのなにします。それらはプログラムをツリーデータでし、ノードは*1*つのをし、そのをするためになすべてのをします。たとえば、 `MethodCallExpression`は、1びす`MethodInfo` 、2そのメソッドにす`Expression`のリスト、3メソッドのメソッド、メソッドをびす`Expression`をします。あなたは "ツリーをいて"あなたのリモートクエリにをすることができます。

ツリーの

ツリーをするもなは、ラムダをするです。これらのは、のCメソッドとほぼじです。これがコンパイラのであることをすることがです。にラムダをすると、コンパイラはそのラムダにをりてるかをチェックします。 `Delegate Action`または`Func`をむの、コンパイラはラムダをデリゲートにします。 `LambdaExpression` またはにされた`LambdaExpression`の`Expression<Action<T>>`または`Expression<Func<T>>` の、コンパイラは`LambdaExpression`します。では、コンパイラはツリー*API*をってラムダを`LambdaExpression`にします。

ラムダは、あらゆるタイプのツリーをすることはできません。そのようなは、Expressions APIをでして、なツリーをすることができます。 APIののでは、 APIをして`CalculateTotalSalesTax`をします。

はここでしします。 ラムダ 2つの、は、 `=>`きのコードブロックをします。これはCでのメソッドをし、 `Delegate`または`Expression`されます。 *`LambdaExpression`* 1つの、PascalCaseは、なメソッドをすExpression APIのノードをします。

ツリーとLINQ

ツリーのもなの1つは、LINQとデータベースクエリです。 LINQはツリーをクエリプロバイダとペアにして、のリモートクエリにをします。たとえば、LINQ to Entity Frameworkクエリプロバイダは、ツリーをSQLにし、データベースにしてします。

すべてのをまとめてみると、LINQののパワーをることができます。

1. ラムダをしてクエリをく `products.Where(x => x.Cost > 5)`
2. コンパイラは、そのをツリーにし、 "パラメータのCostプロパティが5よりきいかどうかをする"というをします。
3. せプロバイダはツリーをし、なSQLせをします`SELECT * FROM products WHERE Cost > 5`
4. ORMはすべてのをPOCOにし、オブジェクトのリストをします

ノート

- ツリーはです。ツリーをするには、しいツリーをするがあります。のツリーをしいツリーにコピーし `ExpressionVisitor` Treeをして、 `ExpressionVisitor`をできます、なをいます。

Examples

APIをしたツリーの

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

ツリーのコンパイル

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

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

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

// Prints True.

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

の

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

```
// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

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

ラムダによるツリーの

は、ラムダによってされたもなツリーです。

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

'で'ツリーをするには、 `Expression`クラスをするがあります。

のはのようになります。

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

APIの

ツリーAPIをして`CalculateSalesTax`ツリーをします。なで、ツリーのにするステップのをにします。

1. がかどうかをする
2. そうであれば、にをけてそのをします
3. そうでなければ0をす

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

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

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

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

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

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

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

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

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

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

ツリー

ツリーは、ツリーのデータのコードをし、ノードは

Expression Treesをすると、コードの、さまざまなデータベースでのLINQクエリの、およびクエリのがになります。ツリーでされるコードをコンパイルしてすることができます。

これらはランタイムDLRでされ、と.NET Frameworkのをし、コンパイラライターがMicrosoft MSILのわりにツリーをできるようにします。

をすることができます

1. ラムダ、
2. でSystem.Linq.Expressionsをします。

ラムダからの

ラムダがExpressionにされると、コンパイラはラムダをすツリーをするコードをします。

のコードは、Cコンパイラがラムダnum => num <5をすツリーをするをしています。

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

APIをした

ツリーは、クラスをしてもされます。このクラスには、のタイプのツリーノードをするファクトリメソッドがまれています。

は、いくつかのタイプのツリーノードです。

1. ParameterExpression
2. MethodCallExpression

のコードは、APIをしてラムダnum => num <5をすツリーをするをしています。

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive,new
ParameterExpression[] { numParam });
```

ビジターをってのをべる

ExpressionVisitorのいくつかのメソッドをオーバーライドして、しいクラスをします。

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

のでこのをするには`Visit`をびす

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

オンラインでをむ https://riptutorial.com/ja/csharp/topic/75/

155: のりをつ

これにはCにのえはありません - いわゆるです。それにもかかわらず、このニーズをたすためのがあります。

が "いわゆる"というをめるは、いプログラミングのにしたときにるためには、2つのをつメソッドしかないということです。にの 。

したがって、2つのをすがなときにをけて、をするがいでしょう。

Examples

「オブジェクト」+「キーワード」ソリューション

からオブジェクトをすことができます

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

をオブジェクトにりて、そののをみみます。

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

タプル

`Tuple<string, MyClass>` 2つのテンプレートパラメータをして、から`Tuple`クラスのインスタンスをすことができます。

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

そして、のようなをんでください

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

パラメータとパラメータ

`ref`キーワードは、 Argument as Referenceをすためにされます。 `out`は`ref`とじようにしますが、をびすにびしによってりてられたをとしません。

Ref Parameter - refパラメータとしてをすは、refパラメータとしてメソッドにすにするがあります。

Out Parameterをoutパラメータとしてすは、outパラメータをoutにすにするはありません。

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int
b, out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

オンラインでのりをつをむ https://riptutorial.com/ja/csharp/topic/3908/のりをつ

156:

Examples

Debug.WriteLine

アプリケーションがデバッグでコンパイルされるときに、Listenersコレクションのトレースリスナーにきみます。

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

Visual StudioまたはXamarin Studioでは、これが[アプリケーション]ウィンドウにされます。これは、TraceListenerCollectionにデフォルトのトレースリスナーがするためです 。

TraceListenersでログをリダイレクトする

Debug.Listenersコレクションに TextWriterTraceListenerをすると、デバッグをテキストファイルにリダイレクトできます。

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

ConsoleTraceListenerをして、デバッグをコンソールアプリケーションのストリームにリダイレクトできます。

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/2147/

157: な

- @ "というは、そのがエスケープされないです。この、\ nはではなく、\とnの2つのです。ののに@
- @ "をエスケープするには、""をします。"

リテラルをするには、のに@をします。

```
var combinedString = @"\t means a tab" + @" and \n means a newline";
```

Examples

の

```
var multiLine = @"This is a

multiline paragraph";
```

これは

の

.NET Fiddleのライブデモ

をむのは、そのままのであるため、にあるものとじようにエスケープすることができます。

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

.NET Fiddleのライブデモ

ここでは、2と3ののはにはのにすることにしてください。なについてこのをチェックしてください。

をエスケープする

のが2つのシーケンシャルをしてエスケープすることができます""つのをすために"のに。

```
var str = @"""I don't think so,"" he said.";
Console.WriteLine(str);
```

"はそうはわない"とはった。

.NET Fiddleのライブデモ

されたなストリング

は、C6のしいとみわせることができます。

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

テスト\ｎ１２３

.NET Fiddleのライブデモ

なからされるように、バックスラッシュはエスケープとしてされます。されたからされるように、のは、そののにされるにされます。

は、エスケープをしないようにコンパイラにします

のでは、バックスラッシュはエスケープです。これは、ののをするためにのをるようにコンパイラにします。 エスケープのなリスト

では、にはのエスケープはありません""になっている" 。をするには、にに、 @のに。

このな

```
var filename = @"c:\temp\newfile.txt"
```

c\ temp \ newfile.txt

のそのままのをするのとはに、

```
var filename = "c:\temp\newfile.txt"
```

それはされます

```
c:    emp
ewfile.txt
```

エスケープをします。 \tはタブにきえられ、 \nはできえられます。

.NET Fiddleのライブデモ

オンラインでなをむ https://riptutorial.com/ja/csharp/topic/16/な

158:

のプロセスは、 Cの 、セクション7.5.3でされています。セクション7.5.2と7.6.5しもしています。

どのようにのがC7でおそらくされるでしょうかノートは、Microsoftがどのがなシナリオでいかをするためのしいシステムをすることをしています。

Examples

なオーバーロードの

このコードには、 **Hello**というのオーバーロードされたメソッドがまれています。

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Mainメソッドがびされると、それはされます

```
int
double
```

コンパイルに、コンパイラがメソッドびし`Hello(0)`をすると、 `Hello`というのすべてのメソッドがされます。この、それらのうちの2つがつかります。に、どのメソッドがれているかをしようとします 。どちらのがいかをするためのアルゴリズムはですが、は「できるだけなをなくする」ということになります。

したがって、のに`Hello(0)`はのためにとされない`Hello(int)`が、なは、のためにとされる`Hello(double)`。 したがって、のはコンパイラによってされます。

`Hello(0.0)`の、 `0.0`を`int`にするはないため、 `Hello(int)`メソッドはのためにされません。メソッドだけがり、コンパイラによってされます。

がないり、「params」はされません。

のプログラム

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

されます

```
5
two
3
```

`Method(objectArray)`のびしは、2つのでできます`Method(objectArray)`つの`Object`はになりますつまり、のまたはのになるため、プログラムは`1`をします。メソッド`Method`はキーワード`params`がありませんが、のされていないフォームがにされるので、プログラムは`5`します。

2の`Method(objectArray, objectArray)`では、のメソッドのとの2のメソッドのがです。このもされていないがされるため、プログラムは`two`します。

3の`Method(objectArray, objectArray, objectArray)`では、のオプションはのメソッドのされたをするため、プログラムは`3`します。

の1つとしてnullをす

あなたがっている

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

らかのでF1ののオーバーロードをびすがありますが、 `x = null`、に

```
F1(null);
```

びしがあいまいであるためコンパイルされません。これにするには、

```
F1(null as MyType1);
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/77/

159: なクラスとメソッド

き

クラスは、クラスをのとのソースファイルにするオプションをします。コンパイルには、すべてのパートが1つのクラスにまとめられます。すべてのにキーワード`partial`がまれていて、じアクセシビリティであるがあります。コンパイルにすべてのパーツをじアセンブリにめるがあります。

- パブリッククラスMyPartialClass {}
- なクラスは、じアセンブリでし、はするクラスとじようにするがあります。
- クラスのすべてので`partial`キーワードをするがあります。
- クラスのすべてのがじアクセシビリティをたなければなりません。 `public` / `protected` / `private`など。
- いずれかのが`abstract`キーワードをする、されたはとなされます。
- いずれかのパートが`sealed`キーワードをする、されたタイプはシールされたとみなされます。
- いずれかのがをする、はそのからします。
- は、すべてのクラスでされたすべてのインタフェースをします。

Examples

クラス

クラスは、クラスをは々のファイルにするをします。クラスでできるのは、コードをしたにがきされることをれずに、ユーザーがコードをできるようにすることです。また、のがじクラスまたはメソッドですることができます。

```
using System;

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

    public partial class PartialClass
```

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

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

な

パ―シャルメソッドは、1つのクラスのなシナリオ - されたものと、のクラスのでされます。

```
using System;

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

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

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

クラスからするクラス

いずれかのクラスからするは、1つのクラスのみがクラスをするがあります。

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

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

のクラスでじクラスをできます。それはいくつかのIDEツールでとしてフラグがてられますが、しくコンパイルされます。

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

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

のクラスになるクラスをすることはできません。コンパイルエラーがします。

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

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

オンラインでなクラスとメソッドをむ https://riptutorial.com/ja/csharp/topic/3674/なクラスとメソッド

160:

- の

  <タイプ> [] <>;

- **2**の

  <タイプ> [、] <> =しい<タイプ> [<>、<>];

- ジグザグの

  <タイプ> [] <> =しい<タイプ> [<>];

- **Jagged Array**のサブアレイをする

  <name> [<value>] =しい<type> [<value>];

- のないの

  <name> =しい<type> [<length>];

- をでする

  <name> =しい<type> [] {<value>、<value>、<value>、...};

- をつ**2**をする

  <name> = new <type> [、] {{<value>、<value>}、{<value>、<value>}、...};

- インデックス**i**のへのアクセス

  <> [i]

- のさをする

  <> .Length

Cでは、はです。つまり、 *null*です。

は、あなたがカントをし、である`.Add()`、それに`.Remove()`それから。これらをするには、Array `List`または`ArrayList`が`ArrayList`。

Examples

の

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

このはセーフではありません。のコードは、ランタイムをさせます。

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0];   // would have been bad if above assignment had succeeded
```

のと

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

の

`[]` をして、をしてデフォルトでめることができます。たとえば、10ののをするとします。

```
int[] arr = new int[10];
```

Cのインデックスはゼロベースです。ののインデックスは09です。えば

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

つまり、システムはインデックスを0からえめます。さらに、のへのアクセスはのにわれます。つまり、ののへのアクセスには、2の、3のなどをアクセスするのとじコストがかかります。

また、をインスタンスせずにへののをすることもできます。

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

は、コレクションをしてカスタムをしておよびすることもできます。

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

`new int[]`は、をするときにすることができます。これはのではないため、のびしのとしてすると

しませんそのために、 `new`バージョンをしてください。

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

にされた

あるいは、 `var`キーワードとみわせて、のをして、のをすることもできます。

```
// same as int[]
var arr = new [] { 1, 2, 3 };
// same as string[]
var arr = new [] { "one", "two", "three" };
// same as double[]
var arr = new [] { 1.0, 2.0, 3.0 };
```

をする

```
int[] arr = new int[] {1, 6, 3, 3, 9};

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

foreachをって

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

ポインタででないアクセスをするhttps://msdn.microsoft.com/en-ca/library/y31yhkeb.aspx

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
            pInt++;// move pointer to next element
        }
    }
}
```

1
6
3

3
9

はのをつことができます。のでは、1010の2をします。

```
int[,] arr = new int[10, 10];
```

3つのの

```
int[,,] arr = new int[10, 10, 10];
```

にをすることもできます

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]);  // 4
```

ジ グザ グ

ジグザグは、プリミティブのわりにまたはのコレクションをむです。ののようなものです。にはのがまれています。

それらはにていますが、はのとにされ、ギザギザではすべてのがなるのをつことができるため、わずかないがあります。

ギザギザのをする

たとえば、8のギザギザのをするとのようになります。

```
int[][] a = new int[8][];
```

2の[]はなしでされます。サブアレイをするには、にうがあります。

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

の*I*

、サブアレイの1つをするのはです。 a 3ののすべてのをしましょう

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

のをする

```
a[<row_number>][<column_number>]
```

のをする

```
a[<row_number>][<column_number>] = <value>
```

にギザギザののではなくマトリックスをすることをおめします。するのがよりくてです。

のにする

1の$_{\text{int}}$の5の3をえてみましょう。これはCでかれています

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

にように、CLRシステムでは、ブラケットのけのためのは、され$_{\text{arr}}$インスタンス々がっています

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

に

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

あるにのがまれているかどうかをする

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
        return false;
```

```
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// サンプル

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

デフォルトのをりしてをする

わかっているように、デフォルトをつをできます

```
int[] arr = new int[10];
```

これにより、のが`0` `int`のデフォルトをつ10ののがされます。

デフォルトのでされたをするには、 `System.Linq`の`Enumerable.Repeat`をし`Enumerable.Repeat` 。

1. **"true"**でたされたサイズ10の`bool`をするには、

   ```
   bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
   ```

2. **"100"**でめられたサイズ5の`int`をするには、

   ```
   int[] intArray = Enumerable.Repeat(100, 5).ToArray();
   ```

3. **"C"**でたされたサイズ5の`string`をするには

   ```
   string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
   ```

のコピー

ソースとのでインデックス0からまる、な`Array.Copy()`メソッドを`Array.Copy()`したのコピー

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);
```

```
// destinationArray will have 11,12 and 3
```

ソースのインデックス0からし、コピーのされたインデックスから`CopyTo()`インスタンスメソッドをしてをコピーする

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone`をしてオブジェクトのコピーをします。

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

`CopyTo`と`Clone`どちらも、いコピーをし`CopyTo`つまり、にはののとじオブジェクトへのがまれています。

のをする

LINQは、したでいっぱいのコレクションをにできるをします。たとえば、1100のをむをできます。

`Enumerable.Range`メソッドをすると、されたとのからのシーケンスをでき`Enumerable.Range`。

このメソッドは、とするのの2つのをとります。

```
Enumerable.Range(int start, int count)
```

*`count`*はであってはいけません。

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

これにより、1100の `[1, 2, 3, ..., 98, 99, 100]` をむがされます。

`Range`メソッドは`IEnumerable<int>`すため、のLINQメソッドをできます。

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

これは、 `4 [4, 9, 16, ..., 100, 121]` 4,9,16 `[4, 9, 16, ..., 100, 121]`からまる10ののをむをします。

ののの

LINQには、2つの`IEnumerable`のをチェックするためのみみがされており、そのをでできます。

`SequenceEqual`がされ`true`はじさをし、するインデックスのがしい、としている`false`そうでありません。

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

これはされます

```
Arrays equal? True
```

をIEnumerable <> インスタンスとして

すべてのは、ジェネリック`IList`インターフェイスしたがってジェネリック`ICollection`および`IEnumerable`ベースインターフェイスをします。

さらになのは、1は、そのにまれるデータの`IList<>`および`IReadOnlyList<>`ジェネリックインターフェイスおよびベースインターフェイスをすることです。つまり、それらをのとしてい、ににすることなく、さまざまなメソッドにすことができます。

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

このコードをすると、リスト`listOfIntegers`に3,5、および7をむ`List<int>`がまれます。

`IEnumerable<>`サポートは、がLINQでできることをします。たとえば、 `arr1.Select(i => 10 * i)`です。

オンラインでをむ https://riptutorial.com/ja/csharp/topic/1429/

161:

キャスティングはコンバートとじではありません。`"-1"`を `-1` にすることはですが、`Convert.ToInt32()`や`Int32.Parse()`などのライブラリメソッドをしてうがあります。キャストをすることはできません。

Examples

オブジェクトをにキャストする

のがえられた

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

オブジェクトをにキャストする

```
    MyClass obj = new MyClass();

    IMyInterface1 myClass1 = (IMyInterface1)obj;
    IMyInterface2 myClass2 = (IMyInterface2)obj;

    Console.WriteLine("I am : {0}", myClass1.GetName());
    Console.WriteLine("I am : {0}", myClass2.GetName());

    // Outputs :
    // I am : IMyInterface1
    // I am : IMyInterface2
```

キャスティング

がのであることがわかっている、そのがなコンテキストでするために、そのににキャストできます。

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

たちはしようとしたは`valueMath.Abs()`ので、たちは、コンパイルのになるだろう`Math.Abs()`けるオ―バ―ロ―ドがない`object`パラメ―タとしてを。

`value` `int`にキャストできない、このの2は`InvalidCastException`をスロ―します

キャスト `as`

あなたがうタイプのであるかどうかなは、 `as`をってにキャストすることができます。がそのでない、のは`null`になり`null`。

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

`null`にはがないので、 `null`をキャストするときに`as`キ―ワ―ドがに`null` `as`すことにしてください。

のキャスティング

コンパイラがにそのにできることがわかっている、はになにキャストされます。

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

このでは、なキャストをするはありませんでした。なぜなら、コンパイラはすべての`int`が`object`sにキャストできることをっているからです。には、のをけ、 `-1`を、 `object`をする`Console.WriteLine()`としてすことができ`object`。

```
Console.WriteLine(-1);
```

キャストなしののチェック

のがされたをまたはするかどうかをるがあるが、にをそのとしてキャストしたくないは、 `is`をできます。

```
if(value is int)
```

```
{
   Console.WriteLine(value + "is an int");
}
```

な

なキャストをして、のをすることはできますが、それらはいにまたはしません。

```
double value = -1.1;
int number = (int) value;
```

タイプのがのタイプよりもい、はわれます。たとえば、のではdoubleとして`-1.1`として`-1`なります。

また、のはコンパイルのにしているため、がオブジェクトに "まれている"はしません。

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

コンバ―ジョン

Cでは、はカスタムのをできます。これにより、またはなキャストをしてをのとのでることができます。たとえば、JavaScriptをすクラスをえてみましょう。

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

2つのJavaScriptのをすJsExpressionをするは、のようにします。

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

しかし、`JsExpression`なをすることで、なキャストをするときになをうことができます。

```
public static explicit operator JsExpression(int value)
{
```

```
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

あるいは、これらのをにして、をはるかににすることができます。

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

LINQキャスティング

のようなタイプがあるとします。

```
interface IThing {  }
class Thing : IThing {  }
```

LINQでは、 `Enumerable.Cast<>()`および`Enumerable.OfType<>()`メソッドをして、 `IEnumerable<>`のコンパイルジェネリックをするプロジェクションをできます。

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

とき`things2`され、 `Cast<>()`メソッドは、にすべてのをキャストしようとする`things`に`Thing`。キャストできないが`InvalidCastException`と、 `InvalidCastException`がスロ―されます。

`things3`がされるとき、 `OfType<>()`メソッドは、キャストできないにしたにをスロ―するのではなくにそのをするというをいてじことをいます。

これらのメソッドのタイプのため、をびしたりをすることはできません。

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

として`.Select()`でにキャストをすることができます

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

オンラインでをむ https://riptutorial.com/ja/csharp/topic/2690/

162: クラス

Examples

キーワード

staticキーワードは2つのことをします

1. このはオブジェクトごとにするのではなく、クラスでします
2. プロパティとメソッドはインスタンスをとしません。

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

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

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

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

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

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

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

    }
}
```

クラス

クラスをするときの「static」キーワードには、3つのがあります。

1. クラスのインスタンスをすることはできません これにより、デフォルトのコンストラクタもされます

2. クラスのすべてのプロパティとメソッドはでなければなりません 。
3. `static`クラスは`sealed`クラスです。 つまり、することはできません。

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

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

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

クラスの

`static`クラスは、メンバーアクセスににされ、アプリケーションドメインのします。

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

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

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

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

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

オンラインでクラスをむ https://riptutorial.com/ja/csharp/topic/1653/クラス

163: -

き

Cでは、 `async`されたメソッドは、I / Oベースの Webアクセス、ファイルのなどをしている、プロセスでブロックされません。そのようなマークされたメソッドのは、 `await`キーワードをして`await`ことができます。

`async`メソッドは、 `void` 、 `Task`または`Task<T>`すことができ`void`。

りのの`Task`は、メソッドがするまでし、は`void`になり`void`。 `Task<T>`は、メソッドのにタイプ`T`からをします。

`async`メソッドは、ほとんどすべてので、 `void`ではなく`Task`または`Task<T>`をすがあり`void`。 `async void`メソッドはedを`await`ことができません。これはさまざまなをきこします。 `async`で`void`をすのシナリオは、イベントハンドラのです。

`async` / `await` 、 `async`メソッドをマシンにし`await`します。これは、のとコンテキストローカルなどをするシーンのにをし、っているちがするたびに`MoveNext()`メソッドをしてをめするコードをするようにします。

Examples

なびし

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

ここですべきなものは、すべてがいるということです`await`て、そののためにがってシステムにもたらしているびす--メソッドでのれがであるとのためになをとしない-edメソッドをにびされました。いずれかのメソッドがした、は "どおり"にされます。この、メソッドのがされ、がスタックにがります。

してみる/キャッチする/に

6.0

C6.0、 `await`キーワードは`catch`および`finally`ブロックでできるようになりました。

```
try {
```

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

5.0 6.0

C6.0は、のにってかをするがありました。 6.0では、 Null Propagatingでヌルチェックもクリーンアップされています。

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

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

`async`されていないタスク `Task.Run`されたタスク`Task.Run` をっている、のtry / catchによってわれても、タスクによってスローされたがデバッガによってすることがあります。これは、デバッガがユーザコードにしてであるとみなすためにします。 Visual Studioには「Just My Code」というオプションがあり、このようなでデバッガがれるのをぐためににすることができます。

しいのために4.5をターゲットにするWeb.configセットアップ。

web.config system.web.httpRuntimeは、メソッドをするにスレッドがコンテキストをにリネームするように4.5をターゲットにするがあります。

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

AsyncとAwaitは、4.5よりのASP.NETでのをしています。 Async / awaitは、コンテキストをたないのスレッドでします。ロードのアプリケーションは、ってからHttpContextにアクセスするnullがすると、ランダムにします。 WebApiでHttpContext.Currentをすると、のためです

のawaitableタスクをし、それらをっていることにより、にのびしをつことがです。

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

```
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

また、 `Task.WhenAll`をして、のタスクをの`Task`にグループすることができます。タスクは、されたすべてのタスクがしたでします。

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

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

また、ループでこれをうこともできます。たとえば、のようにします。

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

await Task.WhenAll(tasks);
```

Task.WhenAllでのタスクをってからタスクのをするには、にタスクをちます。タスクはすでにしているので、はされます

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

await Task.WhenAll(task1, task2);

var result = await task2;
```

また、 `Task.WhenAny`に、にのタスクをするためにすることができる`Task.WhenAll`されるタスクのいずれかがするときに、このがしと、。

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

`firstTask` 、 `secondTask` 、または`thirdTask`いずれかがすると、 `RunConcurrentTasksWhenAny`によってされた`Task`がします。

とキーワードをつ

`await`オペレータと`async`キーワードがにる

awaitをするメソッドは、 **async**キーワードでするがあります。

のことはずしもではありません。メソッドを`await`をにせずに`async`としてマークすることができます。

`await`にうことはのタスクがするまで、コードのをすることです。どんなタスクもつことができます。

もさないメソッドvoidをつことはできません。

には、がするだけでなく、スレッドがのをするためににになるがあるため、「サスペンド」というはをくことがあります。ボンネットのに、 `await`コンパイラののビットによってされます。それはつのにするを-と`await`。のは、っているタスクがしたときにされます。

なをすると、コンパイラはおおまかにあなたのためにこれをいます

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

のようになる

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

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

のは、のようににできます。

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

これは、UIをフリーズせずにUIスレッドでするメソッドをするがあるにです。

しかし、ここではにながあります。 はにまたはマルチスレッドをするとはりません。のスレッド

であっawaitも、 async - awaitはコードをします。たとえば、このカスタムタスクスケジューラをしてください。このような「クレイジー」タスクスケジューラは、タスクをにメッセージループでびされるにえることができます。

々はにねるがありますどんなスレッドがたちのメソッドDoIt_Continuationをするのでしょうか

デフォルトでは、 awaitオペレータはのSynchronizationコンテキストでのをスケジュールします。これは、UIスレッドでWinFormsとWPFがデフォルトでされることをします。なんらかのでこのをするがあるは、 Task.ConfigureAwait() メソッドをします。

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

たずにタスクをす

をするメソッドは、のにawaitをするはありません。

- メソッドにはびしが1つしかありません
- びしはメソッドのにある
- タスクでするかもしれないキャッチ/ハンドリングはありません

Taskをすこのメソッドについてえてみましょう。

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

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

GetByKeyAsyncじするGetUserAsync Task<User> 、をすることができます。

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

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

この、メソッドはをしていてもasyncとマークするはありません。タスクによってさGetByKeyAsyncそれがされ、びしのメソッドにされawait。

TaskをするわりにTaskをすと、 Taskをするメソッドでをスローしないが、っているメソッドでがスローされないため、メソッドのがされます。

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
```

```
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

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

コンパイラがなステートマシンをするのをぐので、パフォーマンスがします。

コードをブロックするとデッドロックがすることがある

コンテキストをつでデッドロックがするがあるため、びしをブロックするのはいです。ベストプラクティスはasyncをすることです。たとえば、のWindowsフォームコードはデッドロックをきこします。

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

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

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

に、コールがすると、コンテキストがになるのをちます。しかし、イベントハンドラは、`TryThis()`メソッドがするのをっているにコンテキストに「ホールド」し、ちをきこします。

これをするには、コードをのようにするがあります。

```
private async void button1_Click(object sender, EventArgs e)
{
```

```
  bool result = await TryThis();
  Trace.TraceInformation("Done with result");
}
```

イベントハンドラは、のです`async void`あなたがつことができないため、すべきである`async void`。

Async / awaitは、マシンがのをうことができるにのみパフォーマンスをさせます

のコードをえてみましょう

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

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

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

これは、

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

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

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

/のなは、マシンがのをできるようにすることです。たとえば、あるI / Oのをっているにびしのスレッドがのをできるようにすることです。この、びしスレッドはしてできなかったよりくのをうことはできません。したがって、に`MethodA()`、 `MethodB()`、および`MethodC()`にびすだけではパフォーマンスはしません。

オンラインで - をむ https://riptutorial.com/ja/csharp/topic/48/---

164: /、バックグラウンドワーカー、タスク、スレッドの

これらのをするには、のようにびします。

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

Examples

ASP.NET

ASP.NETがをすると、スレッドプールからスレッドがりてられ、コンテキストがされます。リクエストコンテキストには、な`HttpContext.Current`プロパティでアクセスできるのリクエストにするがまれています。のコンテキストは、をするスレッドにりてられます。

のコンテキストは、に1つのスレッドでしかアクティブではない。

が`await`と、メソッドがされているにをするスレッドがスレッドプールにされ、のスレッドがするためにコンテキストがされます。

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

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

タスクがすると、スレッドプールはのスレッドをりててのをします。コンテキストは、このスレッドにりてられます。これはのスレッドかもしれません。

ブロッキング

`async`メソッドびしのがにすると、デッドロックがするがあります。たとえば、のコードでは、`IndexSync()`がびされたときにデッドロックがし`IndexSync()`。

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

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

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

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

    return result;
}
```

これは、デフォルトでは`db.Products.ToListAsync()`しているタスク`db.Products.ToListAsync()`このは`db.Products.ToListAsync()`がコンテキストASP.NETのはコンテキストをし、にしようとするためです。

コ―ルスタックがであるは、 `await`になるとのスレッドがされ、コンテキストがされるため、はありません。

`Task.Result`または`Task.Wait()` またはのブロッキングメソッドをしてにブロックすると、のスレッドはとしてアクティブであり、コンテキストをします。しているメソッドはとしてでし、コ―ルバックがしようとすると、つまりしているタスクがったら、コンテキストをしようとします。

したがって、がするのをしているコンテキストをつブロックスレッドでは、がするためにコンテキストをしようとしているため、デッドロックがします。

ConfigureAwait

デフォルトでは、しているタスクをびすと、のコンテキストがされ、したらコンテキストでをしようとします。

`ConfigureAwait(false)`をすることで、これをぐことができ、デッドロックをできます。

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

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

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

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

    return result;
```

```
}
```

これは、コードをブロックするがあるときにデッドロックをすることができますが、これはのコンテキストをうことになりますびしちのコード。

ASP.NETでは、これは、あなたのコードが`await someTask.ConfigureAwait(false);`を`await someTask.ConfigureAwait(false);`く、そのことをします`await someTask.ConfigureAwait(false);`えば、`HttpContext.Current.User`などのコンテキストからにアクセスしようとすると、がわれてしまいます。この、`HttpContext.Current`はnullです。えば

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

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

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

    return View();
}
```

`ConfigureAwait(true)`をするとConfigureAwaitをまったくたない`user`とじです、`user`と`user2`にじデータがされます。

このため、コンテキストがもはやされていないライブラリコードで`ConfigureAwait(false)`をすることをおめします。

/

async / awaitをしてバックグラウンドプロセスでのかかるものをするのなについては、をしてください。

ただし、のかかるメソッドのをでするがあるは、をつことでこれをうことができます。

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

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

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

private async Task<int> TimeintensiveMethod(object file)
```

```
{
    Console.WriteLine("Start TimeintensiveMethod.");

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

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

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

BackgroundWorker

BackgroundWorkerオブジェクトをしてBackgroundWorkerグラウンドスレッドでのかかるをするのなについては、をしてください。

がある

1. のかかるをうワ―カ―メソッドをし、 BackgroundWorker DoWorkイベントのイベントハンドラからコールします。
2. RunWorkerAsyncをしRunWorkerAsync。 ワ―カ―でとされるのDoWorkですことができるDoWorkEventArgsへのパラメ―タRunWorkerAsync。

DoWorkイベントにえて、 BackgroundWorkerクラスは、ユ―ザ―インタ―フェイスとするためにされる2つのイベントもします。これらはオプションです。

- RunWorkerCompletedにイベントがトリガされDoWorkハンドラがしました。
- ProgressChangedイベントは、 ReportProgressメソッドがびされたときにReportProgressれます。

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

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

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

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

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

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

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

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

バックグラウンドプロセスでをする`Task`をうために`Task`をするのなについては、をしてください。

`Task.Run()`びしで、のかかるメソッドをラップするだけです。

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

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

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

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

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

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

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

Threadをしてバックグラウンドプロセスでのかかるをうのなについては、をしてください。

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

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

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

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

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

TimeIntensiveMethod、 Threadはパラメータとしてvoidメソッドをしているので、TimeIntensiveMethodからをすことはできません。

Threadからりをするには、イベントまたはのいずれかをします。

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

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

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

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

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

タスク"してれる"

によってはえばロギング、タスクをしてをたずにむかもしれません。のでは、タスクをしてりのコーードのをできます。

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

は、メソッドでのみたれます。 `async` / `await` がされるので、をし、それをするためのオプションのメソッドをびすことができます。

のいの

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

オンラインで/、バックグラウンドワーカー、タスク、スレッドのをむ
https://riptutorial.com/ja/csharp/topic/3824/--バックグラウンドワーカー-タスク-スレッドの

165: ソケット

き

ソケットをすることで、サ―バはをリッスンし、ソケットとはに、のロジックをすることができます。リスンするとメインスレッドがブロックされ、アプリケ―ションがしなくなりクライアントがするまでフリ―ズします。

ソケットとネットワ―ク

のネットワ―クのにあるサ―バ―にアクセスするにはこれはなであり、されたときににトピックとしてフラグがてられます。

サ―バ

サ―バ―のネットワ―クで、ル―タ―をサ―バ―にするがあります。

サ―バ―がされているPCの

ロ―カルIP = `192.168.1.115`

サ―バ―はポ―ト1234をリッスンしています。

`Port 1234`ルータのを`192.168.1.115`する

クライアント

するがあるのはIPだけです。ル―プバックアドレスにするのではなく、サ―バ―がしているネットワ―クからのパブリックIPにするがあります。このIPはここでできます 。

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

したがって、このエンドポイントでリクエストをします`10.10.10.10:1234 10.10.10.10:1234`プロパティポ―トをルータにした、サ―バとクライアントはなくされます。

ロ―カルIPにするは、ル―プバックアドレスを`192.168.1.178`などにするだけで、portforwartをするはありません。

データの

データはバイトでされます。データをバイトにパックし、もうにするがあります。

ソケットにれているは、するにバイトをすることもできます。これは、もあなたのパッケ―ジをむのをぎます。

Examples

ソケットクライアント/サーバーの

サーバーサイドの

サーバーのリスナーをする

するクライアントをするサーバーをし、するをします。したがって、これをするListenerクラスをします。

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming
connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    }
 }
```

まず、リスナーソケットをして、をできるようにするがあります。たちはSocketType.StreamをうであるTcpソケットをうつもりです。また、々はサーバーがくべきのポートをします

に、をします。

ここでするツリーメソッドはのとおりです。

1. ListenerSocket.Bind;

   このメソッドは、ソケットをIPEndPointにバインドします。このクラスには、アプリケーションがホストのサービスにするためになホストおよびローカルまたはリモートのポートがまれます。

2. ListenerSocket.Listen10;

   backlogパラメータは、けれのためにキューにれることができるのをします。

3. ListenerSocket.BeginAccept;

   サーバーはのリスンをし、のロジックにみます。があると、サーバーはこのメソッドにり、AcceptCallBackメソッドをします。

```
    public void StartListening()
    {
        try
        {
```

```
                MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
                ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
                ListenerSocket.Listen(10);
                ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch(Exception ex)
        {
            throw new Exception("listening error" + ex);
        }
    }
```

クライアントがすると、このでけれることができます

ここでは**3**つのをしています

1. ListenerSocket.EndAccept

   `Listener.BeginAccept()`コールバックをしました。コールバックをするがあります。 `The EndAccept()`メソッドはIAsyncResultパラメータをけります。これはメソッドのをします。このから、がしたソケットをできます。

2. `ClientController.AddClient()`

   `EndAccept()`からしたソケットをして、のメソッドサーバーののの*ClientController*コード`EndAccept()`をつクライアントをします 。

3. ListenerSocket.BeginAccept

   しいのでソケットがしたら、びリスニングをするがあります。このコールバックをキャッチするメソッドをします。また、Listenerソケットをintにすので、のでこのソケットをできます。

```
    public void AcceptCallback(IAsyncResult ar)
    {
        try
        {
            Console.WriteLine($"Accept CallBack port:{Port} protocol type:
{ProtocolType.Tcp}");
            Socket acceptedSocket = ListenerSocket.EndAccept(ar);
            ClientController.AddClient(acceptedSocket);

            ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch (Exception ex)
        {
            throw new Exception("Base Accept error"+ ex);
        }
    }
```

はListen Socketがありますが、クライアントがのコードがしているデータをどのようにけるのでしょうか。

クライアントごとにサーバーレシーバーをする

に、Socketをパラメータとしてけるコンストラクタをつクラスをします。

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

のでは、にバッファに4バイトInt32のサイズをえるか、またはパッケージにに{Lenght、のデータ}をめることからめます。の4バイトはデータのさのためにされ、りはのデータのりのです。

に、 BeginReceiveメソッドをします。このメソッドは、されたクライアントからのをするためにされ、データをすると`ReceiveCallback`がされます。

```
    public void StartReceiving()
    {
        try
        {
            _buffer = new byte[4];
            _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
ReceiveCallback, null);
        }
        catch {}
    }

    private void ReceiveCallback(IAsyncResult AR)
    {
        try
        {
            // if bytes are less than 1 takes place when a client disconnect from the server.
            // So we run the Disconnect function on the current client
            if (_receiveSocket.EndReceive(AR) > 1)
            {
                // Convert the first 4 bytes (int 32) that we received and convert it to an
Int32 (this is the size for the coming data).
                _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
                // Next receive this data into the buffer with size that we did receive before
                _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
                // When we received everything its onto you to convert it into the data that
you've send.
                // For example string, int etc... in this example I only use the
implementation for sending and receiving a string.

                // Convert the bytes to string and output it in a message box
                string data = Encoding.Default.GetString(_buffer);
                MessageBox.Show(data);
                // Now we have to start all over again with waiting for a data to come from
the socket.
                StartReceiving();
            }
            else
            {
                Disconnect();
```

```
            }
        }
        catch
        {
            // if exeption is throw check if socket is connected because than you can
startreive again else Dissconect
            if (!_receiveSocket.Connected)
            {
                Disconnect();
            }
            else
            {
                StartReceiving();
            }
        }
    }

    private void Disconnect()
    {
        // Close connection
        _receiveSocket.Disconnect(true);
        // Next line only apply for the server side receive
        ClientController.RemoveClient(_clientId);
        // Next line only apply on the Client Side receive
        Here you want to run the method TryToConnect()
    }
```

そこで、をしてできるサ―バ―をセットアップしました。クライアントがすると、クライアントのリストにされ、すべてのクライアントはのクラスをちます。サ―バ―がlistenするようにするには

```
Listener listener = new Listener();
listener.StartListening();
```

このでするクラス

```
    class Client
    {
        public Socket _socket { get; set; }
        public ReceivePacket Receive { get; set; }
        public int Id { get; set; }

        public Client(Socket socket, int id)
        {
            Receive = new ReceivePacket(socket, id);
            Receive.StartReceiving();
            _socket = socket;
            Id = id;
        }
    }

     static class ClientController
     {
          public static List<Client> Clients = new List<Client>();

          public static void AddClient(Socket socket)
          {
              Clients.Add(new Client(socket,Clients.Count));
```

```
        }

        public static void RemoveClient(int id)
        {
            Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
        }
    }
```

クライアントサイドの

サーバーにつなげる

まずに、サーバーにするクラスをしたいとします。をけますConnector

```
class Connector
{
    private Socket _connectingSocket;
}
```

へこのクラスのメソッドはTryToConnectです。

このは、いくつかのいものがあります

1. ソケットをします。
2. に、ソケットがされるまでループします
3. すべてのループそれはちょうどたちがDOSにサーバーXDしたくないスレッドを1している
4. Connectをすると、サーバーにしようとします。したはがスローされますが、プログラムはサーバーにしたままになります。このためにConnect CallBackメソッドをすることはできますが、Socketがされているときにメソッドをびすためにします。
5. クライアントがポート1234のローカルPCにしようとしていることにしてください。

```
 public void TryToConnect()
 {
     _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

      while (!_connectingSocket.Connected)
      {
          Thread.Sleep(1000);

          try
          {
              _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
1234));
          }
          catch { }
      }
      SetupForReceiveing();
     }
 }
```

```
private void SetupForReceiveing()
{
   // View Client Class bottom of Client Example
    Client.SetClient(_connectingSocket);
    Client.StartReceiving();
}
```

サ―バ―へのメッセ―ジの

これでほぼまたはソケットアプリケ―ションがしました。たちがジェットをっていないのは、サ―バにメッセ―ジをするためのクラスだけです。

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what hapends here:
                 1. Create a list of bytes
                 2. Add the length of the string to the list.
                    So if this message arrives at the server we can easily read the length of
the coming message.
                 3. Add the message(string) bytes
            */

            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}*/
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
```

に、とメッセ―ジの2つのボタンをします。

```
    private void ConnectClick(object sender, EventArgs e)
    {
        Connector tpp = new Connector();
        tpp.TryToConnect();
    }

    private void SendClick(object sender, EventArgs e)
```

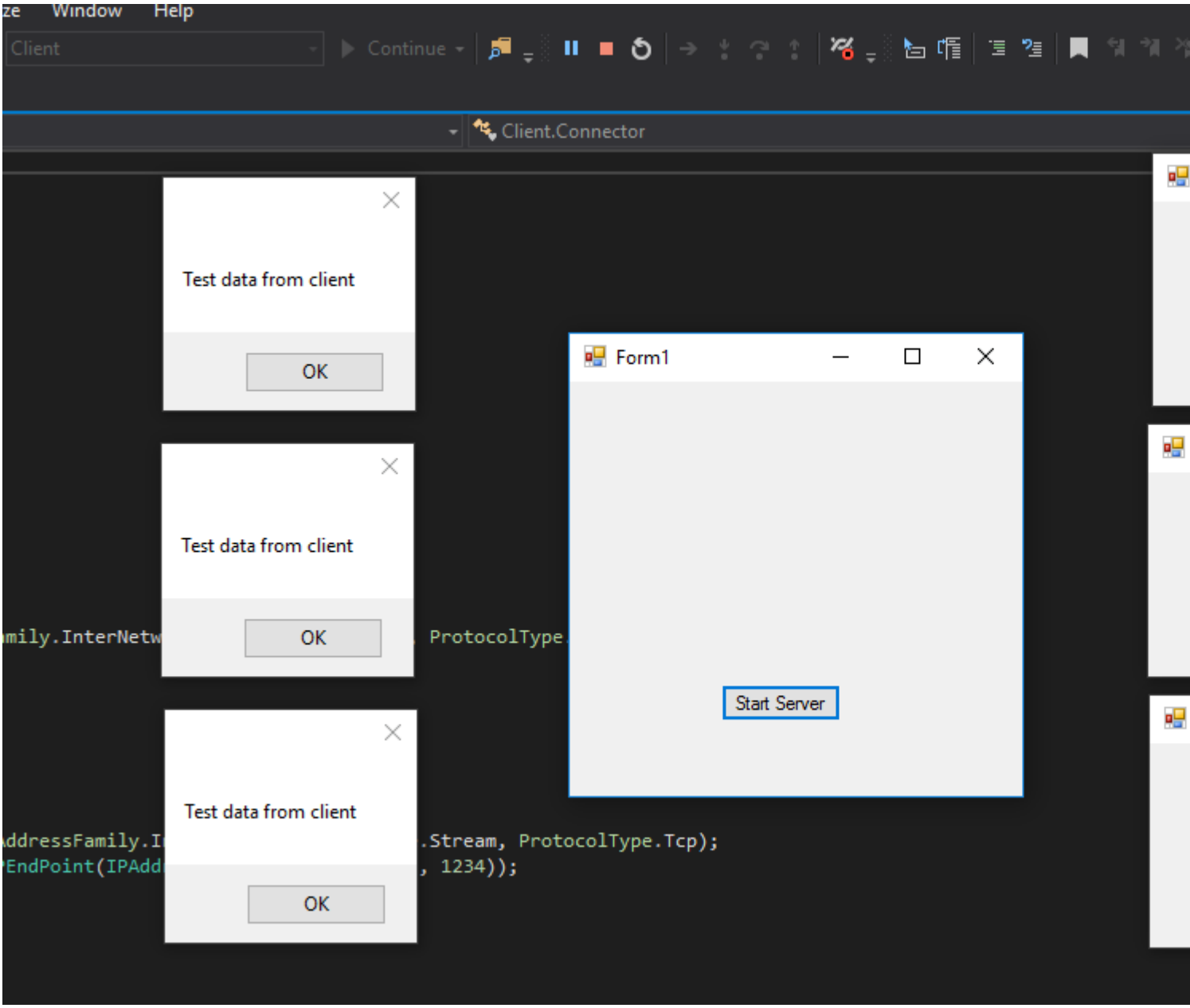
```
    {
        Client.SendString("Test data from client");
    }
```

このでしたクラ イアント クラス

```
    public static void SetClient(Socket socket)
    {
        Id = 1;
        Socket = socket;
        Receive = new ReceivePacket(socket, Id);
        SendPacket = new SendPacket(socket);
    }
```

サ―バ―からのクラスは、クライアントからのクラスとじです。

これで、サ―バ―とクライアントがされまし た。このなをいこなすことができます。えば、サ―バ―がファイルやそののをできるようにします。または、クライアントにメッセ―ジをします。サ―バ―にはクライアントのリストがあるので、かをけったときにクライアントからのをることができます。



オンラインでソケットをむ https://riptutorial.com/ja/csharp/topic/9638/ソケット

クレジット

S. No		Contributors
1	Cをいめる	4444, A. Raza, A_Arnold , aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Configure, crunchy, David G., David Pine, DavidG, DAXaholic , Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch , Frankenstine Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis , Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k , Kevin Montrose, Kritner , Krzyserious, leumas1960, M Monis

		Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput , Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki , Zze
2	.NETコンパイラプラットフォームRoslyn	4444, Lukáš Lánský
3	.NETのガベージコレクタ	Andrei Rînea, da_sann, Eamon Charles, J3soon, Luke Ryan, Squidward, Suren Srapyan
4	.NETのでないコード	Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz
5	.zipファイルのみき	4444, DLeh, Naveen Gogineni, Nisarg Shah
6	ASP.NETアイデンティティ	HappyCoding, Skullomania
7	AssemblyInfo.csの	Adi Lester, Ameya Deshpande,

		AndreyAkinshin, Boggin, Dodzi Dzakuma, dove, Joel Martinez, pinkfloydx33, Ralf Bönning, Theodoros Chatzigiannakis, Wasabi Fan
8	Async-Awaitのコンテキスト	codeape, Mark Shevchenko, RamenChef
9	BackgroundWorker	Bovaz, Draken, ephtee, Jacobr365, Will
10	BigInteger	4444, Ed Marty, James Hughes, Rob, The_Outsider
11	BindingList	Bovaz, Stephen Leppik, yumaikas
12	C3.0の	0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob
13	C4.0の	Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima, Sibeesh Venu, Squidward, Theodoros Chatzigiannakis
14	C5.0の	Abob, alex.b, H. Pauwelyn
15	C6.0の	A_Arnold, Aaron Anodide, Aaron Hudon, Adil Mammadov, Adriano Repetti, AER, AGB, Akshay Anand, Alan McBee, Alex Logan, Amitay Stern, anaximander, andre_ss6, Andrea, AndroidMechanic, Ares, Arthur Rizzo, Ashwin Ramaswami, avishayp, Balagurunathan

		Marimuthu, Bardia, Ben Aaronson, Blubberguy22, Bobson, bpoiss, Bradley Uffner, Bret Copeland, C4u, Callum Watkins, Chad Levy, Charlie H, ChrFin, Community, Conrad.Dean, Cyprien Autexier, Dan, Daniel Minnaar, Daniel Stradowski, DarkV1, dasblinkenlight, David, David G., David Pine, Deepak gupta, DLeh, dotctor, Durgpal Singh, Ehsan Sajjad, el2iot2, Emre Bolat, enrico.bacis, Erik Schierboom, fabriciorissetto, faso, Franck Dernoncourt, FrankerZ, Gabor Kecskemeti, Gary, Gates Wong, Geoff, GingerHead, Gordon Bell, Guillaume Pascal, H. Pauwelyn, hankide, Henrik H, iliketocode, Iordanis , Irfan, Ivan Yurchenko, J. Steen, Jacob Linney, Jamie Rees, Jason Sturges, Jeppe Stig Nielsen, Jim, JNYRanger, Joe, Joel Etherton, John Slegers, Johnbot, Jojodmo, Jonas S, Juan, Kapep, ken2k, Kit, Konamiman, Krikor Ailanjian, Lafexlos, LaoR, Lasse Vågsæther Karlsen, M.kazem Akhgary, Mafii, Magisch, Makyen, MANISH KUMAR CHOUDHARY, Marc, MarcinJuraszek, Mark Shevchenko, Matas Vaitkevicius,

		Mateen Ulhaq, Matt, Matt, Matt, Matt Thomas, Maximillian Laumeister, mbrdev, Mellow, Michael Mairegger, Michael Richardson, Michał Perłakowski, mike z, Minhas Kamal, Mitch Talmadge, Mohammad Mirmostafa, Mr.Mindor, mshsayem, MuiBienCarlota, Nate Barbettini, Nicholas Sizer , nik, nollidge, Nuri Tasdemir, Oliver Mellet, Orlando William, Osama AbuSitta, Panda, Parth Patel, Patrick, Pavel Voronin, PSN, qJake, QoP, Racil Hilan, Radouane ROUFID, Rahul Nikate, Raidri, Rajeev, Rakitić, ravindra, rdans, Reeven, Richa Garg, Richard, Rion Williams, Rob, Robban, Robert Columbia, Ryan Hilbert, ryanyuyu, Sam, Sam Axe, Samuel, Sender, Shekhar, Shoe, Slayther, solidcell, Squidward, Squirrel, stackptr, stark, Stilgar, Sunny R Gupta, Suren Srapyan, Sworgkh, syb0rg, takrl, Tamir Vered, Theodoros Chatzigiannakis, Timothy Shields, Tom Droste, Travis J, Trent, Trikaldarshi, Troyen, Tushar patel, tzachs, Uri Agassi, Uriil, uTeisT, vcsjones, Ven, viggity, Vishal Madhvani, Vlad, Wai Ha Lee, Xiaoy312, Yury Kerbitskov, Zano,

		Ze Rubeus, Zimm1
16	C7.0の	Adil Mammadov, afuna, Amitay Stern, Amr Badawy, Andreas Pähler, Andrew Diamond, Avi Turner, Benjamin Hodgson, Blorgbeard, bluray, Botond Balázs, Bovaz, Cerbrus, Clueless, Conrad.Dean, Dale Chen, David Pine, Degusto, Didgeridoo, Diligent Key Presser, ECC-Dan, Emre Bolat, fallaciousreasoning, ferday, Florian Greinacher, ganchito55, Ginkgo, H. Pauwelyn, Henrik H, Icy Defiance, Igor Ševo, iliketocode, Jatin Sanghvi, Jean-Bernard Pellerin, Jesse Williams, Jon Schoning, Kimmax, Kobi, Kris Vandermotten, Kritner, leppie, Llwyd, Maakep, maf-soft, Marc Gravell, MarcinJuraszek, Mariano Desanze, Matt Rowland, Matt Thomas, MemphiZ, mnoronha, MotKohn, Name, Nate Barbettini, Nico, Niek, nietras, NikolayKondratyev, Nuri Tasdemir, PashaPash, Pavel Mayorov, PeteGO, petrzjunior, Philippe, Pratik, Priyank Gadhiya, Pyritie, qJake, Raidri, Rakitić, RamenChef, Ray Vega, RBT, René Vogt, Rob, samuelesque, Squidward, Stavm, Stefano, Stefano d'Antonio, Stilgar, Tim Pohlmann, Uriil,

		user1304444, user2321864, user3185569, uTeisT, Uwe Keim, Vlad, Vlad, Wai Ha Lee, Wasabi Fan , WerWet, wezten, Wojciech Czerniak, Zze
17	CStructsをしてUnionをするC Unionと	DLeh, Milton Hernandez, Squidward, usr
18	Cコレクションの	Aaron Hudon, Andrew Diamond, Denuath, Jeremy Kato, Jon Schneider, Jorge, Juha Palomäki, Leon Husmann, Michael Mairegger, Michael Richardson, Nikita, rene, Rob, Sebi, TarkaDaal, wertzui, Will Ray
19	Cスクリプト	mehrandvd, Squidward, Stephen Leppik
20	CでSQLiteをう	Carmine, NikolayKondratyev, th1rdey3, Tim Yusupov
21	Cでをする	A. Can Aydemir, Adi Lester, Alexander Mandt , DLeh, J3soon, Rob
22	Cハンドラ	Abbas Galiyakotwala
23	CLSCompliantAttribute	mybirthname, Rob
24	DateTimeメソッド	AbdulRahman Ansari, C4u, Christian Gollhardt, Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz , Oluwafemi, Pavel Yermalovich, Sondre, theinarasu, Thulani Chivandikwa

25	FileSystemWatcher	Sondre
26	Funcデリゲート	Theodoros Chatzigiannakis, Valentin
27	Googleのをインポートする	4444, Supraj v
28	HTTPリクエストの	Gordon Bell, Jon Schneider, Mark Shevchenko
29	ICloneable	ja72, Rob
30	IComparable	alex
31	IDisposableインターフェイス	Aaron Hudon, Adam, BatteryBackupUnit, binki, Bogdan Gavril, Bryan Crosby, ChrisWue, Dmitry Bychenko, Ehsan Sajjad, H. Pauwelyn, Jarrod Dixon, Josh Peterson, Matas Vaitkevicius, Maxime, Nicholas Sizer, OliPro007, Pavel Mayorov, pinkfloydx33, pyrocumulus, RamenChef, Rob, Thennarasan, Will Ray
32	IEnumerable	4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider
33	ILGenerator	Aleks Andreev, thehennyy
34	INotifyPropertyChangedインターフェイス	mbrdev, Stephen Leppik, Vlad
35	IQueryableインターフェイス	lucavgobbi, Michiel van Oosterhout, RamenChef, Rob
36	json.netの	Aleks Andreev, Snipzwolf

37	LINQ to XML	Denis Elkhov, Stephen Leppik, Uali
38	Linqからオブジェクトへ	brijber, Christian Gollhardt, FortyTwo, Kevin Green, Raphael Pantaleão, Simon Halsey, Tanveer Badar
39	LINQクエリ	Adam Clifford, Ade Stringer, Adi Lester, Adil Mammadov, Akshay Anand, Aleksey L., Alexey Koptyaev, AMW, anaximander, Andrew Piliser, Ankit Vijay, Aphelion, bbonch, Benjamin Hodgson, bmadtiger, BOBS, BrunoLM, Budi, bumbeishvili, callisto, cbale, Chad McGrath, Chris, Chris H., coyote, Daniel Argüelles, Daniel Corzo, darcyq, David, David G., David Pine, DavidG, die maus, Diligent Key Presser, Dmitry Bychenko, Dmitry Egorov, dotctor, Ehsan Sajjad, Erick, Erik Schierboom, EvenPrime, fabriciorissetto, faso, Finickyflame, Florin M, forsvarir, fubo, gbellmann, Gene, Gert Arnold, Gilad Green, H. Pauwelyn, Hari Prasad, hellyale, HimBromBeere, hWright, iliketocode, Ioannis Karadimas, Jagadisha B S, James Ellis-Jones, jao, jiaweizhang, Jodrell, Jon Bates, Jon G, Jon Schneider, Jonas S, karaken12, KevinM,

		Koopakiller, leppie, LINQ , Lohitha Palagiri, ltiveron, Mafii, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matt, Maxime, mburleigh, Meloviz, Mikko Viitala, Mohammad Dehghan, mok, Nate Barbettini, Neel, Neha Jain, Néstor Sánchez A., Nico, Noctis , Pavel Mayorov, Pavel Yermalovich, Paweł Hemperek, Pedro, Phuc Nguyen, pinkfloydx33, przno, qJake, Racil Hilan , rdans, Rémi, Rion Williams, rjdevereux, RobPethi, Ryan Abbott, S. Rangeley, S.Akbari, S.L. Barth, Salvador Rubio Martinez, Sanjay Radadiya, Satish Yadav, sebingel, Sergio Domínguez, SilentCoder, Sivanantham Padikkasu, slawekwin, Sondre, Squidward, Stephen Leppik, Steve Trout, Tamir Vered, techspider, teo van kot, th1rdey3, Theodoros Chatzigiannakis, Tim Iles , Tim S. Van Haren, Tobbe, Tom, Travis J, tungns304, Tushar patel, user1304444, user3185569, Valentin, varocarbas, VictorB, Vitaliy Fedorchenko, vivek nuna, void, wali, wertzui, WMios, Xiaoy312, Yaakov Ellis, Zev Spitz
40	Microsoft.Exchange.WebServices	Bassie

41	Nullable	Benjamin Hodgson, Braydie, DmitryG, Gordon Bell, Jasmin Solanki, Jon Schneider, Konstantin Vdovkin, Maximilian Ast, Mikko Viitala, Nicholas Sizer, Patrick Hofman, Pavel Mayorov, pinkfloydx33, Vitaliy Fedorchenko
42	NullReferenceException	4444, Agramer, Ashutosh, krimog, Kyle Trauberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato
43	Onののためのアルゴリズム	AFT
44	ObservableCollection	demonplus, GeralexGR, Jonathan Anctil, MuiBienCarlota
45	Stacktracesをみ、する	S.L. Barth
46	String.Format	Aaron Hudon, Akshay Anand, Alexander Mandt , Andrius, Aseem Gautam, Benjol, BrunoLM, Dmitry Egorov , Don Vince, Dweeberly, ebattulga, ejhn5, gdoron, H. Pauwelyn, Hossein Narimani Rad, Jasmin Solanki, Marek Musielak, Mark Shevchenko, Matas Vaitkevicius, Mendhak, MGB, nikchi, Philip C, Rahul Nikate, Raidri, RamenChef, Richard, Richard, Rion Williams, ryanyuyu, teo van kot, Vincent, void, Wyck
47	StringBuilder	ATechieThought, brijber,

		Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider
48	System.DirectoryServices.Protocols.LdapConnection	Andrew Stollak
49	System.Management.Automation	Mikko Viitala
50	T4コード	lloyd, Pavel Mayorov
51	Windows Communication Foundation	NtFreX
52	Windowsフォームアプリケーションでの MessageBoxをする	Mansel Davies, Vaibhav_Welcomes_You
53	XDocumentとSystem.Xml.Linq	Crowcoder, Jon Schneider
54	XmlDocumentとSystem.Xml	Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr
55	XMLドキュメントのコメント	Alexander Mandt, James , jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn
56	アクションフィルター	Lokesh_Ram
57	アクセス	Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz
58	イテレータ	Botond Balázs, Lijo, Nate Barbettini, Tagc
59	イベント	Aaron Hudon, Adi Lester , Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng , NotEnoughData, OliPro007, paulius_l, PSGuy, Reza Aghaei,

		Roy Dictus, Squidward, Steven, vbnet3d
60	インターフェイス	Avia, Botond Balázs, CyberFox, harriyott, hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511
61	インデクサー	A_Arnold, Ehsan Sajjad, jHilscher
62	オーバーフロー	Akshay Anand, Nuri Tasdemir, tonirush
63	オブジェクト	Andrei, Kroltan, LeopardSkinPillBoxHat, Marco, Nick DeVore, Stephen Leppik
64	オブジェクトプログラミングC	Yashar Aliabasi
65	ガイド	Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void
66	キーワード	4444, A_Arnold, Aaron Hudon, Ade Stringer, Adi Lester, Aditya Korti, Adriano Repetti, AJ., Akshay Anand, Alex Filatov, Alexander Pacha , Amir Pourmand, Andrei Rînea, Andrew Diamond, Angela, Anna, Avia, Bart , Ben, Ben Fogel, Benjamin Hodgson, Bjørn-Roger Kringsjå, Botz3000, Brandon, brijber, BrunoLM, BunkerMentality, BurnsBA, bwegs, Callum Watkins, Chris, Chris

Akridge, Chris H., Chris Skardon, ChrisPatrick, Chuu, Cihan Yakar, cl3m , Craig Brett, Daniel, Daniel J.G., Danny Chen , Darren Davies, Daryl, dasblinkenlight, David, David G., David L, David Pine, DAXaholic, deadManN, DeanoMachino, digitlworld, Dmitry Bychenko, dotctor, DPenner1, Drew Kennedy, DrewJordan, Ehsan Sajjad, EJoshuaS , Elad Lachmi, Eric Lippert, EvenPrime, F_V, Felix, fernacolo, Fernando Matsumoto, forsvarir, Francis Lord, Gavin Greenwalt, gdoron , George Duckett, Gilad Naaman, goric, greatwolf , H. Pauwelyn, Happypig375, Icemanind , Jack, Jacob Linney, Jake, James Hughes, Jcoffman, Jeppe Stig Nielsen, jHilscher, João Lourenço, John Slegers, JohnD, Jon Schneider, Jon Skeet, JoshuaBehrens, Kilazur, Kimmax, Kirk Woll, Kit, Kjartan, kjhf, Konamiman , Kyle Trauberman, kyurthich, levininja, lokusking, Mafii, Mamta D, Mango Wong, MarcE, MarcinJuraszek, Marco Scabbiolo, Martin, Martin Klinke, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matěj Pokorný, Mat's Mug, Matthew Whited, Max,

Maximilian Ast, Medeni Baykal, Michael Mairegger, Michael Richardson, Michel Keijzers, Mihail Shishkov , mike z, Mr.Mindor, Myster, Nicholas Sizer, Nicholaus Lawson, Nick Cox, Nico, nik, niksofteng, NotEnoughData, numaroth, Nuri Tasdemir , pascalhein, Pavel Mayorov, Pavel Pája Halbich, Pavel Yermalovich, Pavieł Kraskoŭski, Paweł Mach, petelids, Peter Gordon, Peter L., PMF, Rakitić, RamenChef, ranieuwe, Razan, RBT, Renan Gemignani, Ringil, Rion Williams, Rob, Robert Columbia, ro binmckenzie, RobSiklos, Romain Vincent, RomCoo, ryanyuyu, Sain Pradeep, Sam, Sándor Mátyás Márton, Sanjay Radadiya, Scott, sebingel, Skipper, Sobieck, sohnryang, somebody, Sondre, Squidward, Stephen Leppik, Sujay Sarma, Suyash Kumar Singh, svick, TarkaDaal, th1rdey3, Thaoden, Theodoros Chatzigiannakis, Thorsten Dittmar, Tim Ebenezer, titol, tonirush, topolm, Tot Zam, user3185569, Valentin, vcsjones, void, Wasabi Fan, Wavum, Woodchipper,

		Xandrmoro, Zaheer Ul Hassan, Zalomon, Zohar Peled
67	キャッシング	Aliaksei Futryn, th1rdey3
68	コード	MegaTron
69	コードとアサーション	Roy Dictus
70	コメントと	Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward
71	コレクション	Aphelion, ASh, Bart Jolling, Chronocide, CodeCaster, CyberFox, DLeh, Jacob Linney, Jeromy Irvine, Jonas S, Matas Vaitkevicius, Rob, robert demartino, rudygt, Squidward, Tamir Vered, TarkaDaal, Thulani Chivandikwa, WMios
72	コンストラクタとファイナライザ	Adam Sills, Adi Lester, Adriano Repetti, Andrei Rînea, Andrew Diamond, Arjan Einbu, Avia, BackDoorNoBaby, BanksySan, Ben Fogel, Benjamin Hodgson, Benjol, Bogdan Gavril, Bovaz, Carlos Muñoz, Dan Hulme, Daryl, DLeh, Dmitry Bychenko, drusellers, Ehsan Sajjad, Fernando Matsumoto, guntbert, hatchet, Ian, Jeremy Kato, Jon Skeet, Julien Roncaglia, kamilk, Konamiman, ltiveron, Michael Richardson, Neel, Oly, Pavel Mayorov, Pavel Sapehin, Pavel Voronin, Peter Hommel, pinkfloydx33,

		Robert Columbia, RomCoo, Roy Dictus, Sam, Saravanan Sachi, Seph, Sklivvz, The_Cthulhu_Kid, Tim Medora, usr, Verena Haunschmid, void, Wouter, ZenLulz
73	ジェネリックス	AGB, andre_ss6, Ben Aaronson, Benjamin Hodgson, Benjol, Bobson, Carsten, darth_phoenixx, dymanoid, Eamon Charles, Ehsan Sajjad, Gajendra, GregC, H. Pauwelyn, ja72, Jim, Kroltan, Matas Vaitkevicius, mehmetgil, meJustAndrew, Mord Zuber, Mujassir Nasir, Oly, Pavel Voronin, Richa Garg, Sam, Sebi, Sjoerd222888, Theodoros Chatzigiannakis, user3185569, VictorB, void, Wallace Zhang
74	シングルトン	Aaron Hudon, Adam, Adi Lester, Andrei Rînea, cbale, Disk Crasher, Ehsan Sajjad, Krzysztof Branicki, lothlarias, Mark Shevchenko, Pavel Mayorov, Sklivvz, snickro, Squidward, Squirrel, Stephen Leppik , Victor Tomaili, Xandrmoro
75	ステートメントの	Adam Houldsworth, Ahmar, Akshay Anand, Alex Wiese, andre_ss6, Aphelion, Benjol, Boris Callens, Bradley Grainger, Bradley Uffner,

		bubbleking, Chris Marisic, ChrisWue, Cristian T, cubrr, Dan Ling, Danny Chen, dav_i, David Stockinger, dazerdude, Denis Elkhov, Dmitry Bychenko, Erik Schierboom, Florian Greinacher, gdoron, H. Pauwelyn, Herbstein, Jon Schneider, Jon Skeet, Jonesopolis, JT., Ken Keenan, Kev, Kobi, Kyle Trauberman, Lasse Vågsæther Karlsen, LegionMammal978, Lorentz Vedeler, Martin, Martin Zikmund, Maxime, Nuri Tasdemir, Peter K, Philip C, pid, René Vogt, Rion Williams, Ryan Abbott, Scott Koland, Sean, Sparrow, styfle, Sunny R Gupta, Sworgkh, Thaoden, The_Cthulhu_Kid, Tom Droste, Tot Zam, Zaheer Ul Hassan
76	ストップウォッチ	Adam, demonplus, dotctor, Gavin Greenwalt, Jeppe Stig Nielsen, Sondre
77	ストリーム	Danny Bogers, jlawcordova, Jon Schneider, Nuri Tasdemir, Pushpendra
78	スレッディング	Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepdelaw, Mamta D, Matthijs Wessels, Mellow, RamenChef, Zoba

79	スレッドセーフなをる	Wyck
80	ダイナミックタイプ	Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob
81	タイプ	Community, connor, Ehsan Sajjad, Lijo
82	タイマー	Adam, Akshay Anand, Benjamin Kozuch, ephtee, RamenChef, Thennarasan
83	タスクライブラリ	Benjamin Hodgson, Brandon, Collin Stevens, i3arnon, Mokhtar Ashour , Murtuza Vohra
84	タスクライブラリTPLのデータフロー	Droritos, Stephen Leppik
85	タプル	Bovaz, Chawin, EFrank, H. Pauwelyn, Mark Benovsky, Muhammad Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011
86	チェックされているとチェックされていない	Botond Balázs, Rahul Nikate, Sam Johnson, ZenLulz
87	データベースへのアクセス	ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864
88	データ	Maxime, Mikko Viitala, The_Outsider, Will Ray
89	デコレータデザインパターンの	Jan Bońkowski
90	ヌルき	Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas

		Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan
91	ヌル	aashishkoirala, Ankit Rana, Aristos, Bradley Uffner, David Arno, David G., David Pine, demonplus, Denis Elkhov, Diligent Key Presser, Eamon Charles, Ehsan Sajjad, eouw0o83hf, Fernando Matsumoto, H. Pauwelyn , Jodrell, Jon Schneider, Jonesopolis, Martin Zikmund, Mike C, Nate Barbettini, Nic Foster, petelids, Prateek, Rahul Nikate, Rion Williams, Rob, smead, tonirush, Wasabi Fan, Will Ray
92	ネットワーキング	Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano
93	バイナリシリアル	David, Maxim, RamenChef, Stephen Leppik
94	はじめにJson with C	Neo Vijay, Rob, VitorCioletti
95	ハッシュ	Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz
96	パラレルLINQPLINQ	Adi Lester
97	ビルトインタイプ	Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee

98	ファイルとストリームのI / O	BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgun DZ, intox, Mikko Viitala, Sender, Squidward, Tolga Evcimen, Wasabi Fan
99	フォントリソースをむ	Bales, Facebamm
100	フライウェイトデザインパターンの	Jan Bońkowski
101	プリプロセッサディレクティブ	Andrei, Gilad Naaman, Matas Vaitkevicius, qJake, RamenChef, theB, volvis
102	プレーンテキストエディタとCコンパイラcsc.exeをしたコンソールアプリケーションの	delete me
103	プロパティ	Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled
104	プロパティの	Blorgbeard, hatchet, jaycer, Michael Sorens, Parth Patel, Stephen Leppik
105	ポインタ	Jeppe Stig Nielsen, Theodoros Chatzigiannakis
106	ポインタとでないコード	Aaron Hudon, Botond Balázs, undefined
107	メソッド	Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky
108	ユーザとパスワードでネットワークフォルダにアク	Mohsin khan

	セスする	
109	ラムダ	Andrei Rînea, Benjamin Hodgson, Benjol, David L, David Pine, Federico Allocati, Feelbad Soussi Wolfgun DZ, Fernando Matsumoto, H. Pauwelyn , haim770, Matas Vaitkevicius, Matt Sherman, Michael Mairegger, Michael Richardson, NotEnoughData, Oly, RubberDuck, S.L. Barth, Sunny R Gupta, Tagc, Thriggle
110	ランタイムコンパイル	Artificial Stupidity, Stephen Leppik, Tommy
111	リアクティブエクステンションRx	stefankmitph
112	リテラル	jaycer, NotEnoughData, Racil Hilan
113	ルーピング	Alisson, Andrei Rînea, B Hawkins, Benjamin Hodgson, Botond Balázs , connor, Dialecticus, DJCubed, Freelex, Jon Schneider, Oluwafemi, Racil Hilan, Squidward, Testing123, Tolga Evcimen
114	ロックステートメント	Aaron Hudon, Alexey Groshev, Andrei Rînea, Benjamin Hodgson, Botond Balázs, Christopher Currens, Cihan Yakar, David Ben Knoble, Denis Elkhov, Diligent Key Presser, George Duckett, George Polevoy, Jargon, Jasmin Solanki, Jivan, Mark Shevchenko, Matas

		Vaitkevicius, Mikko Viitala, Nuri Tasdemir, Oluwafemi, Pavel Mayorov, Richard, Rob, Scott Hannen, Squidward, Vahid Farahmandian
115		Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer
116		Aaron Hudon, Adam, Ben Aaronson, Benjamin Hodgson, Bradley Uffner , CalmBit, Cihan Yakar, CodeWarrior, EyasSH, Huseyin Durmus, Jasmin Solanki, Jeppe Stig Nielsen, Jon G, Jonas S, Matt, NikolayKondratyev, niksofteng, Rajput, Richa Garg, Sam Farajpour Ghamari, Shog9, Stu, Thulani Chivandikwa, trashr0x
117		0x49D1, Abdul Rehman Sayed, Adam Lear, Adil Mammadov, Andrew Diamond, Aseem Gautam, Athafoud, Botond Balázs, Collin Stevens, Danny Chen, Dmitry Bychenko, dove, Eldar Dordzhiev, fabriciorissetto, faso, flq, George Duckett, Gilad Naaman, Gudradain, Jack, James Hughes, Jamie Rees, John Meyer , Jonesopolis, MadddinTribleD, Marimba, Matas Vaitkevicius, Matt, matteeyah, Mendhak, Michael Bisbjerg, Nate Barbettini, Nathaniel

		Ford, nik0lias, niksofteng , Oly, Pavel Pája Halbich , Pavel Voronin, PMF, Racil Hilan, raidensan, Rasa , Robert Columbia, RomCoo, Sam Hanley, Scott Koland, Squidward , Steve Dunn, Thulani Chivandikwa, vesi
118		Buh Buh, iaminvinicble, Kyle Trauberman, Wiktor Dębski
119	のとの	Abdul Rehman Sayed, Adam, Amir Pourmand, Blubberguy22, Chronocide, Craig Brett, docesam, GWigWam, matiaslauriti, meJustAndrew, Michael Mairegger, Michele Ceo, Moe Farag, Nate Barbettini, RamenChef, Rob, scher, Snympi, Tagc, Theodoros Chatzigiannakis
120		Austin T French, Blachshma, bluish, CharithJ, Chief Wiggum, cyberj0g, Daryl, deloreyk , jaycer, Jaydip Jadhav, Jon G, Jon Schneider, juergen d, Konamiman, Maniero, Paul Weiland, Racil Hilan, RoelF, Stefan Steiger, Steven, The_Outsider, tiedied61, un-lucky, WizardOfMenlo
121		Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes,

		Yves Schelpe
122		Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked, Kyle Trauberman, Martin Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, Tanner Swett, TarkaDaal , Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai
123	なデザイ ンパターン	DWright, Jan Bońkowski, Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel
124	の	Fernando Matsumoto, goric, Stephen Leppik
125		Alexander Mandt, Aman Sharma, artemisart, Aseem Gautam, Axarydax, Benjamin Hodgson, Botond Balázs , Carson McManus, Cigano Morrison Mendez , Cihan Yakar, da_sann, DVJex, Ehsan Sajjad, H. Pauwelyn, Haim Bendanan,

		HimBromBeere, James Ellis-Jones, James Hughes, Jamie Rees, Jan Peldřimovský, Johny Skovdal, JSF, Kobi, Konamiman, Kristijan, Lovy, Matas Vaitkevicius , Mourndark, Nuri Tasdemir, pinkfloydx33, Rekshino, René Vogt, Sachin Chavan, Shuffler, Sjoerd222888, Sklivvz, Tamir Vered, Thriggle, Travis J, uygar.raf, Vadim Ovchinnikov, wablab, Wai Ha Lee
126	のキーワード	Aaron Hudon, Andrew Diamond, Ben Aaronson , ChrisPatrick, Damon Smithies, David G., David Pine, Dmitry Bychenko, dotctor, Ehsan Sajjad, erfanrazi, Gajendra, George Duckett, H. Pauwelyn, HimBromBeere, Jeremy Kato, João Lourenço, Joe Amenta, Julien Roncaglia, just.ru, Karthik AMR, Mark Shevchenko, Michael Richardson, MuiBienCarlota, Myster, Nate Barbettini, Noctis, Nuri Tasdemir, Olivier De Meulder, OP313, ravindra, Ricardo Amores, Rion Williams, rocky, Sompom, Tot Zam, un-lucky, Vlad, void, Wasabi Fan, Xiaoy312, ZenLulz
127	き	Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen

		Leppik
128	きとオプション	RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov
129		Ben Aaronson, Callum Watkins, PMF, ZenLulz
130		Ade Stringer, ganchito55 , H. Pauwelyn, Karthik, Maximilian Ast, void
131		Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili
132	メソッド	Aaron Hudon, AbdulRahman Ansari, Adi Lester, Adil Mammadov, AGB, AldoRomo88, anaximander, Aphelion, Ashwin Ramaswami, ATechieThought, Ben Aaronson, Benjol, binki, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, Brett Veenstra, brijber, Callum Watkins, Chad McGrath, Charlie H, Chris Akridge, Chronocide, CorrectorBot, cubrr, Dan-Cook, Daniel Stradowski, David G., David Pine, Deepak gupta, diiN__________, DLeh, Dmitry Bychenko, DoNot, DWright, Đan, Ehsan Sajjad, ekolis, el2iot2, Elton, enrico.bacis, Erik

Schierboom, ethorn10, extremeboredom, Ezra, fahadash, Federico Allocati, Fernando Matsumoto, FrankerZ, gdziadkiewicz, Gilad Naaman, GregC, Gudradain, H. Pauwelyn, HimBromBeere, Hsu Wei Cheng, Icy Defiance, Jamie Rees, Jeppe Stig Nielsen, John Peters, John Slegers, Jon Erickson, Jonas S, Jonesopolis, Kev, Kevin Avignon, Kevin DiTraglia , Kobi, Konamiman, krillgar, Kurtis Beavers, Kyle Trauberman, Lafexlos, LMK, lothlarias, Lukáš Lánský, Magisch, Marc, MarcE, Marek Musielak, Martin Zikmund, Matas Vaitkevicius, Matt, Matt Dillard, Maximilian Ast, mbrdev, MDTech.us_MAN, meJustAndrew, Michael Benford, Michael Freidgeim, Michael Richardson, Michał Perłakowski, Nate Barbettini, Nick Larsen, Nico, Nisarg Shah, Nuri Tasdemir, Parth Patel, pinkfloydx33, PMF, Prashanth Benny, QoP, Raidri, Reddy, Reeven, Ricardo Amores, Richard , Rion Williams, Rob, Robert Columbia, Ryan Hilbert, ryanyuyu, S. Tarık Çetin, Sam Axe, Shoe , Sibeesh Venu, solidcell , Sondre, Squidward, Steven, styfle, SysVoid,

		Tanner Swett, Timothy Rascher, TKharaishvili, T-moty, Tobbe, Tushar patel, unarist, user3185569, user40521, Ven, Victor Tomaili, viggity
133	の	Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit, Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam
134	エスケープシーケンス	Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush
135	の	Arjan Einbu, ATechieThought, avs099, bluray, Brendan L, Dave Zych, DLeh, Ehsan Sajjad, fabriciorissetto, Guilherme de Jesus Santos, H. Pauwelyn, Jon Skeet, Nate Barbettini, RamenChef, Rion Williams, Squidward, Stephen Leppik, Tushar patel, Wasabi Fan
136	をのにするのFormatExceptionの	Rakitić, un-lucky
137		Blachshma, Jon Schneider, sferencik, The_Outsider
138		Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi,

		Rob, RubberDuck, Testing123, The_Outsider
139	System.Security.Cryptography	glaubergft, MikeS159, Ogglas, Pete
140		Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz
141		abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad Lachmi, ganchito55, Jon Schneider, NikolayKondratyev
142	パターン	Timon Post
143	プログラミング	Andrei Epure, Boggin, Botond Balázs, richard
144	の	C4u
145	ラムダクエリビルダ	4444, PedroSouki
146		Adam Houldsworth, Adi Lester, Adil Mammadov, Akshay Anand, Alan McBee, Avi Turner, Ben Fogel, Blorgbeard, Blubberguy22, Chris Jester-Young, David Basarab, DLeh, Dmitry Bychenko, dotctor, Ehsan Sajjad, fabriciorissetto, Fernando Matsumoto, H. Pauwelyn, Henrik H, Jake Farley, Jasmin Solanki, Jephron, Jeppe Stig Nielsen, Jesse Williams, Joe, JohnLBevan, Jon Schneider, Jonas S,

		Kevin Montrose, Kimmax, lokusking, Matas Vaitkevicius, meJustAndrew, Mikko Viitala, mmushtaq, Mohamed Belal, Nate Barbettini, Nico, Oly, pascalhein, Pavel Voronin, petelids, Philip C, Racil Hilan, RhysO, Robert Columbia, Rodolfo Fadino Junior, Sachin Joseph, Sam, slawekwin, slinzerthegod, Squidward, Testing123, TyCobb, Wasabi Fan, Xiaoy312, Zaheer Ul Hassan
147	の	Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan
148		Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward
149	しいとGetHashCode	Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew, Rob, scher, Timitry, viggity
150		Vadim Martynov
151	みみのエイリアス	Racil Hilan, Rahul Nikate, Stephen Leppik

152		Almir Vuk, andre_ss6, Andrew Diamond, Barathon, Ben Aaronson, Ben Fogel, Benjol, David L, deloreyk, Ehsan Sajjad, harriyott, ja72, Jon Ericson, Karthik, Konamiman, MarcE, Matas Vaitkevicius, Pete Uh, Rion Williams, Robert Columbia, Steven, Suren Srapyan, VirusParadox, Yehuda Shapira
153		Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer, Matas Vaitkevicius, matteeyah, meJustAndrew, Nathan Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Рахул Маквана
154	のりをつ	Adam, Alexey Mitev, Durgpal Singh, Tolga Evcimen
155		Jasmin Solanki, Luke Ryan, TylerH
156	な	Alan McBee, Amitay Stern, Andrew Diamond, Aphelion, Arjan Einbu, avb, Bryan Crosby, Charlie H, David G., devuxer, DLeh, Ehsan Sajjad, Freelex, goric, Jared Hooper, Jeremy Kato, Jonas S, Kevin Montrose, Kilazur, Mateen Ulhaq, Ricardo Amores, Rion Williams, Sam Johnson, Sophie Jackson-Lee, Squirrel,

		th1rdey3
157		Dunno, Petr Hudeček, Stephen Leppik, TorbenJ
158	なクラスとメソッド	Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider
159		A_Arnold, Aaron Hudon, Alexey Groshev, Anas Tasadduq, Andrii Abramov, Baddie, Benjamin Hodgson, bluray, coyote, D.J., das_keyboard, Fernando Matsumoto, granmirupa, Jaydip Jadhav, Jeppe Stig Nielsen, Jon Schneider, Ogoun, RamenChef, Robert Columbia, Shyju, The_Outsider, Thomas Weller, tonirush, Tormod Haugene, Wasabi Fan, Wen Qin, Xiobiq, Yotam Salmon
160		Benjamin Hodgson, MSE , RamenChef, StriplingWarrior
161	クラス	MCronin, The_Outsider, Xiaoy312
162	-	Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean, Craig Brett, Donald Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft,

		Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev, Stefano d'Antonio, Stephen Leppik, SynerCoder, trashr0x, Tseng, user2321864, Vincent
163	/、バックグラウンドワーカー、タスク、スレッドの	Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi
164	ソケット	Timon Post