Kostenloses eBook

LERNEN

C# Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c#

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit C # Language

Bemerkungen

C # ist eine aus mehreren Paradigmen stammende, von C nachkommende Programmiersprache. C # ist eine verwaltete Sprache, die nach CIL kompiliert wird, ein Zwischenbytecode, der unter Windows, Mac OS X und Linux ausgeführt werden kann.

Die Versionen 1.0, 2.0 und 5.0 wurden von ECMA (als ECMA-334) standardisiert, und Standardisierungsbemühungen für modernes C # sind im Gange.

Versionen

Ausführung	Veröffentlichungsdatum
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

Erstellen einer neuen Konsolenanwendung (Visual Studio)

1. Öffnen Sie Visual Studio
2. Gehen Sie in der Symbolleiste zu **Datei** → **Neues Projekt**
3. Wählen Sie den Projekttyp der **Konsolenanwendung aus**
4. Öffnen Sie die Datei `Program.cs` im Projektmappen-Explorer
5. Fügen Sie den folgenden Code zu `Main()` :

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

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

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

6. Klicken Sie in der Symbolleiste auf **Debug** -> **Debugging starten** oder **drücken Sie F5** oder **Strg + F5** (ohne Debugger), um das Programm auszuführen.

Live Demo auf ideone

Erläuterung

- `class Program` ist eine Klassendeklaration. Die Klasse `Program` enthält die Daten- und Methodendefinitionen, die Ihr Programm verwendet. Klassen enthalten im Allgemeinen mehrere Methoden. Methoden definieren das Verhalten der Klasse. Die `Program` hat jedoch nur eine Methode: `Main` .

- `static void Main()` definiert die `Main` Methode, die den Einstiegspunkt für alle C # - Programme darstellt. Die `Main` Methode gibt an, was die Klasse bei ihrer Ausführung tut. Nur ein `Main` wird pro Klasse erlaubt.

- `System.Console.WriteLine("Hello, world!");` Diese Methode druckt bestimmte Daten (in diesem Beispiel `Hello, world!`) als Ausgabe im Konsolenfenster.

- `System.Console.ReadKey()` stellt sicher, dass das Programm nicht sofort nach Anzeige der Nachricht geschlossen wird. Dazu wird gewartet, bis der Benutzer eine Taste auf der Tastatur drückt. Jeder Tastendruck des Benutzers beendet das Programm. Das Programm wird beendet, wenn die letzte Codezeile in der Methode `main()` ist.

Verwenden der Befehlszeile

Verwenden Sie zum Kompilieren über die Befehlszeile entweder `MSBuild` oder `csc.exe` *(den C # - Compiler)* , beide Teil des Microsoft Build Tools- Pakets.

Führen Sie zum Kompilieren dieses Beispiels den folgenden Befehl in demselben Verzeichnis aus, in dem sich `HelloWorld.cs` befindet:

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

Es kann auch sein, dass Sie zwei Hauptmethoden in einer Anwendung haben. In diesem Fall müssen Sie dem Compiler mitteilen, welche Hauptmethode ausgeführt werden soll, indem Sie den

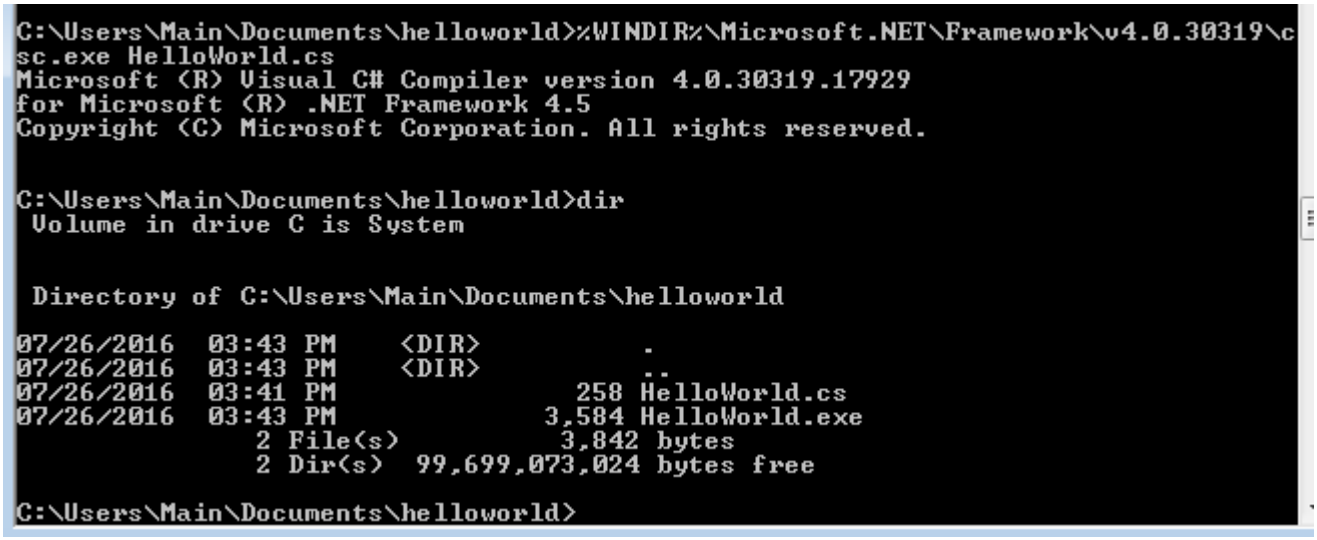
folgenden Befehl in die **Konsole** `ClassA` (Angenommen, Class `ClassA` hat auch eine Hauptmethode in derselben `HelloWorld.cs` Datei im HelloWorld-Namespace)

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

wo HelloWorld der Namespace ist

***Hinweis** : Dies ist der Pfad, in dem sich **.NET Framework v4.0** im Allgemeinen befindet. Ändern Sie den Pfad entsprechend Ihrer .NET-Version. Wenn Sie das 32-Bit-.NET Framework verwenden, ist das Verzeichnis möglicherweise ein **Framework** anstelle von **framework64** . Über die Windows-Eingabeaufforderung können Sie alle Pfade des csc.exe-Frameworks auflisten, indem Sie die folgenden Befehle ausführen (die erste für 32-Bit-Frameworks):*

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



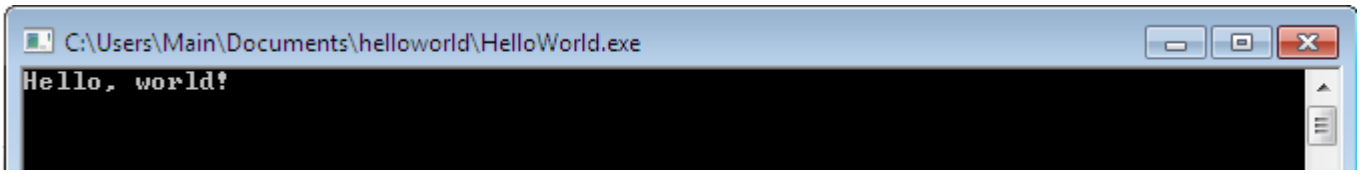
Es sollte sich nun eine ausführbare Datei namens `HelloWorld.exe` im selben Verzeichnis befinden. Um das Programm von der Eingabeaufforderung aus auszuführen, geben Sie einfach den Namen der ausführbaren Datei ein und drücken Sie die `Eingabetaste` wie folgt:

```
HelloWorld.exe
```

Dies wird produzieren:

> Hallo Welt!

Sie können auch auf die ausführbare Datei doppelklicken und ein neues Konsolenfenster mit der Meldung " **Hallo, Welt!** "

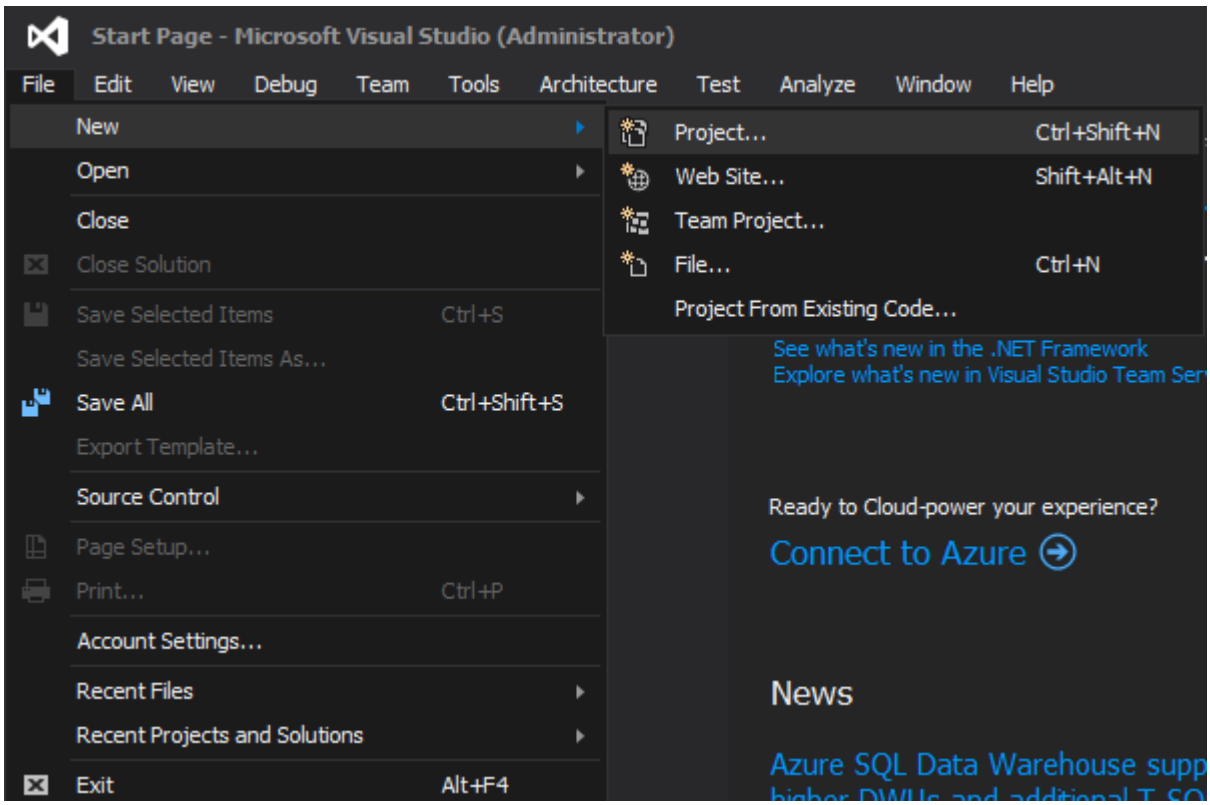


Erstellen eines neuen Projekts in Visual Studio (Konsolenanwendung) und Ausführen im Debugmodus

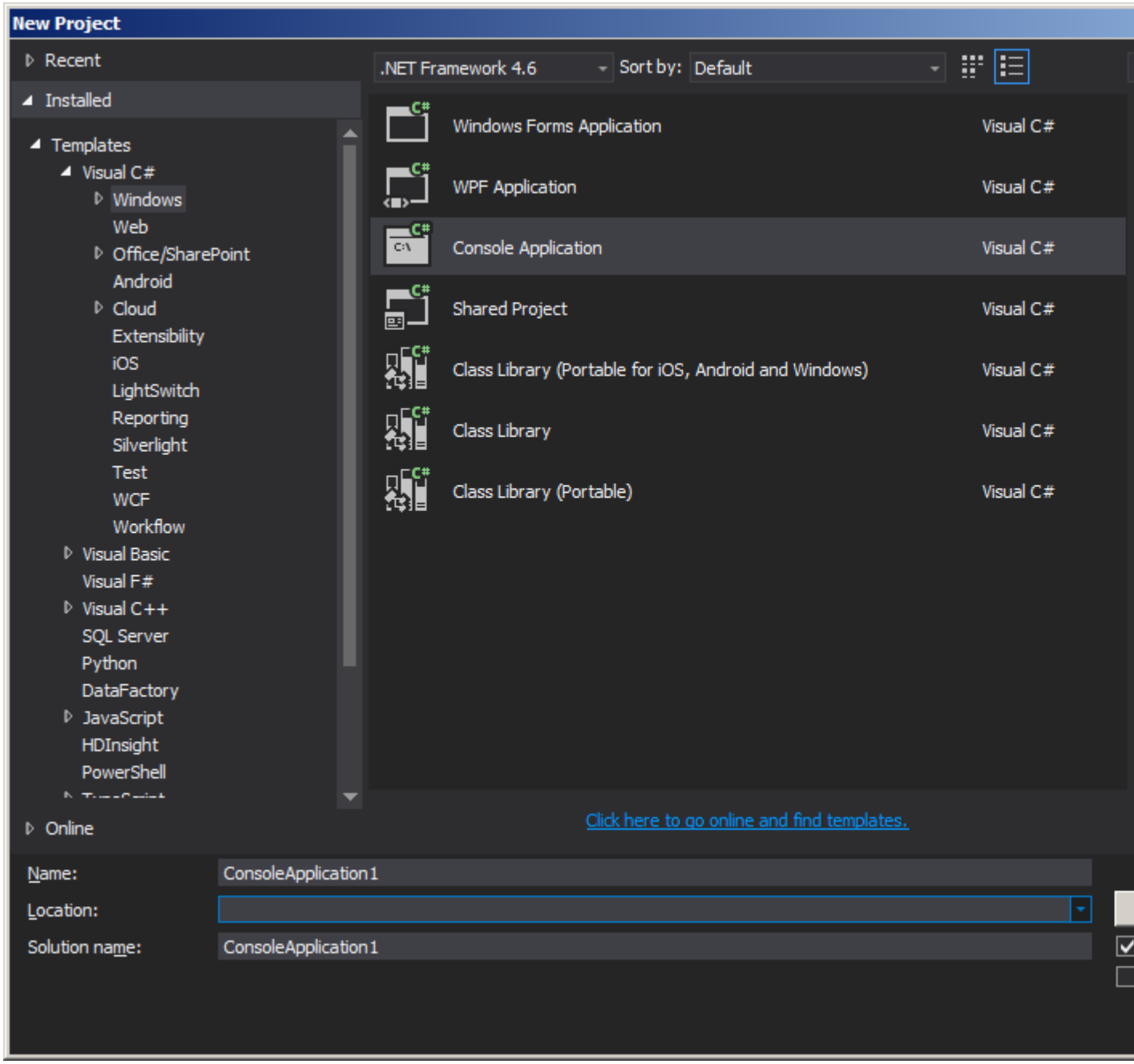
1. **Laden Sie Visual Studio herunter und installieren Sie es** . Visual Studio kann von VisualStudio.com heruntergeladen werden . Die Community Edition wird vorgeschlagen, erstens, weil sie kostenlos ist, und zweitens, weil sie alle allgemeinen Funktionen beinhaltet und weiter erweitert werden kann.

2. **Öffnen Sie Visual Studio.**

3. **Herzlich willkommen.** Gehen Sie zu **Datei → Neu → Projekt** .

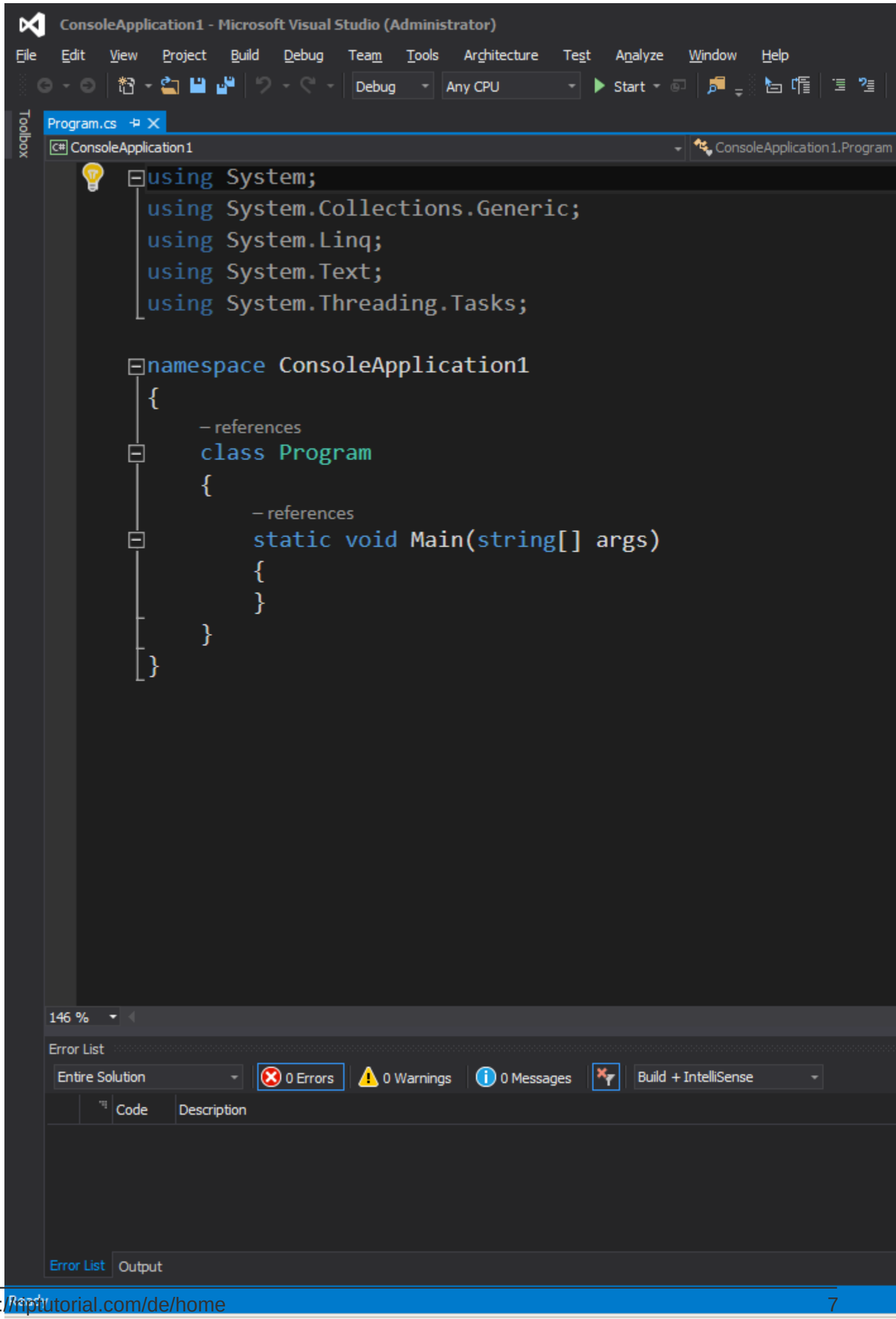


4. Klicken Sie auf **Vorlagen** → **Visual C #** → **Konsolenanwendung**



5. **Nachdem Sie die Konsolenanwendung ausgewählt haben, geben** Sie einen Namen für Ihr Projekt und einen Speicherort ein und drücken Sie `OK` . Machen Sie sich keine Sorgen um den Lösungsnamen.

6. **Projekt erstellt** Das neu erstellte Projekt sieht ähnlich aus:


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

(Verwenden Sie immer beschreibende Namen für Projekte, damit sie leicht von anderen Projekten unterschieden werden können. Es wird empfohlen, keine Leerzeichen in Projekt- oder Klassennamen zu verwenden.)

7. **Code schreiben** Sie können jetzt Ihre `Program.cs` aktualisieren, um "Hallo Welt!" an den Benutzer.

   aktualisieren, um "Hallo Welt!" an den Benutzer.

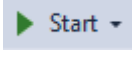
   ```
   using System;

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

   Fügen Sie dem `public static void Main(string[] args)` Objekt `public static void Main(string[] args)` in `Program.cs` die folgenden zwei Zeilen hinzu: (Stellen Sie sicher, dass es in den geschweiften Klammern steht)

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

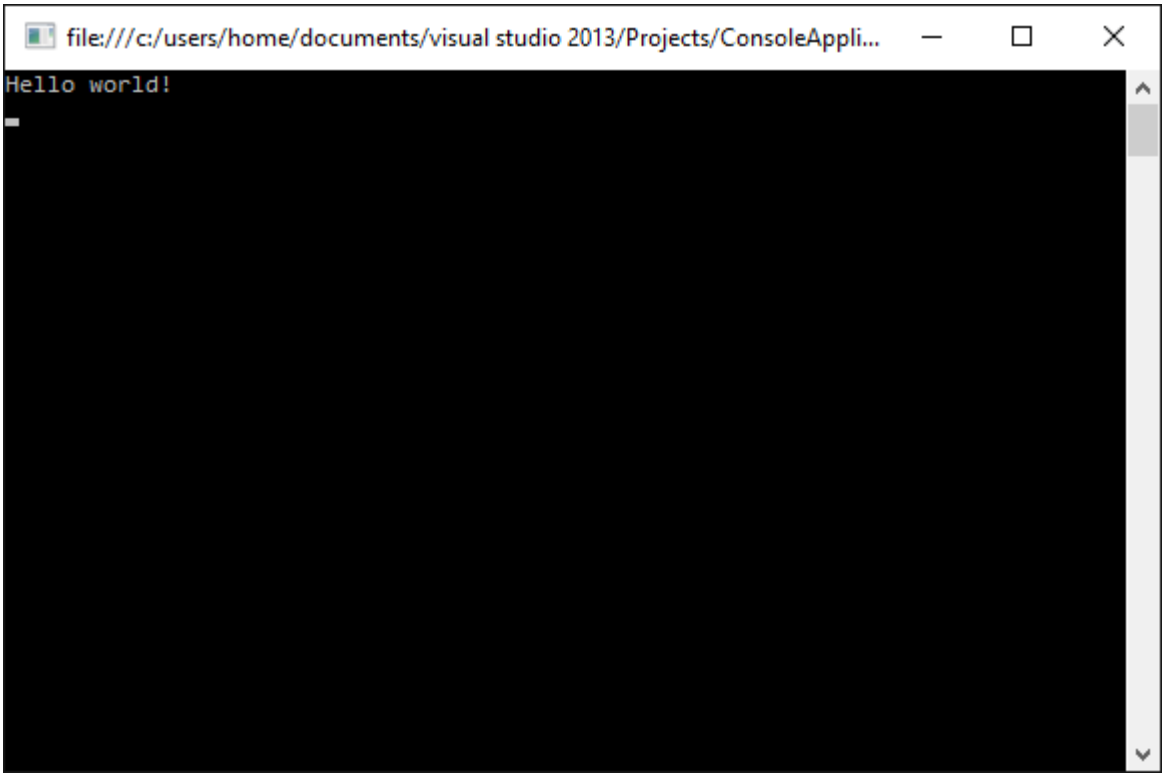
   Warum `Console.Read()` **?** Die erste Zeile druckt den Text "Hallo Welt!" an die Konsole und die zweite Zeile wartet auf die Eingabe eines einzelnen Zeichens; Dadurch wird die Ausführung des Programms angehalten, sodass Sie die Ausgabe während des Debugging sehen können. Ohne `Console.Read();` Wenn Sie mit dem Debuggen der Anwendung beginnen, wird nur "Hallo Welt!" gedruckt. auf die Konsole und dann sofort schließen. Ihr Code-Fenster sollte jetzt wie folgt aussehen:

   ```
   using System;

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

8. **Debuggen Sie Ihr Programm.** Drücken Sie die Start-Schaltfläche in der Symbolleiste am oberen Rand des Fensters 

oder drücken Sie `F5` auf Ihrer Tastatur, um Ihre Anwendung auszuführen. Wenn die Schaltfläche nicht vorhanden ist, können Sie das

Programm über das Hauptmenü ausführen: **Debug → Debugging starten** . Das Programm wird ein Konsolenfenster kompilieren und dann öffnen. Es sollte dem folgenden Screenshot ähnlich aussehen:



9. **Stoppen Sie das Programm.** Um das Programm zu schließen, drücken Sie einfach eine beliebige Taste auf Ihrer Tastatur. Das von uns hinzugefügte `Console.Read()` diente zu demselben Zweck. Sie können das Programm auch beenden, indem Sie in das Menü wechseln, in dem sich die Schaltfläche `Start befindet` , und auf die Schaltfläche `Stopp` klicken.

Ein neues Programm mit Mono erstellen

Installieren Sie zuerst Mono, indem Sie die Installationsanweisungen für die Plattform Ihrer Wahl durchgehen, wie im Installationsabschnitt beschrieben .

Mono ist für Mac OS X, Windows und Linux verfügbar.

Erstellen Sie nach Abschluss der Installation eine Textdatei, nennen Sie sie `HelloWorld.cs` und kopieren Sie den folgenden Inhalt hinein:

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

Wenn Sie Windows verwenden, führen Sie die Mono-Eingabeaufforderung aus, die in der Mono-Installation enthalten ist, und stellt sicher, dass die erforderlichen Umgebungsvariablen festgelegt sind. Öffnen Sie unter Mac oder Linux ein neues Terminal.

Führen Sie den folgenden Befehl in dem Verzeichnis mit `HelloWorld.cs` um die neu erstellte Datei zu kompilieren:

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

Die resultierende `HelloWorld.exe` kann dann ausgeführt werden mit:

```
mono HelloWorld.exe
```

was wird die Ausgabe erzeugen:

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

Erstellen eines neuen Programms mit .NET Core

Installieren Sie zunächst das **.NET Core SDK,** indem Sie die Installationsanweisungen für die Plattform Ihrer Wahl durchgehen:

- Windows
- OSX
- Linux
- Docker

Öffnen Sie nach Abschluss der Installation eine Eingabeaufforderung oder ein Terminalfenster.

1. Erstellen Sie ein neues Verzeichnis mit `mkdir hello_world` und wechseln Sie in das neu erstellte Verzeichnis mit `cd hello_world` .
2. Erstellen Sie eine neue Konsolenanwendung mit der `dotnet new console` .
   Dadurch werden zwei Dateien erzeugt:

   - **hello_world.csproj**

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

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

     </Project>
     ```

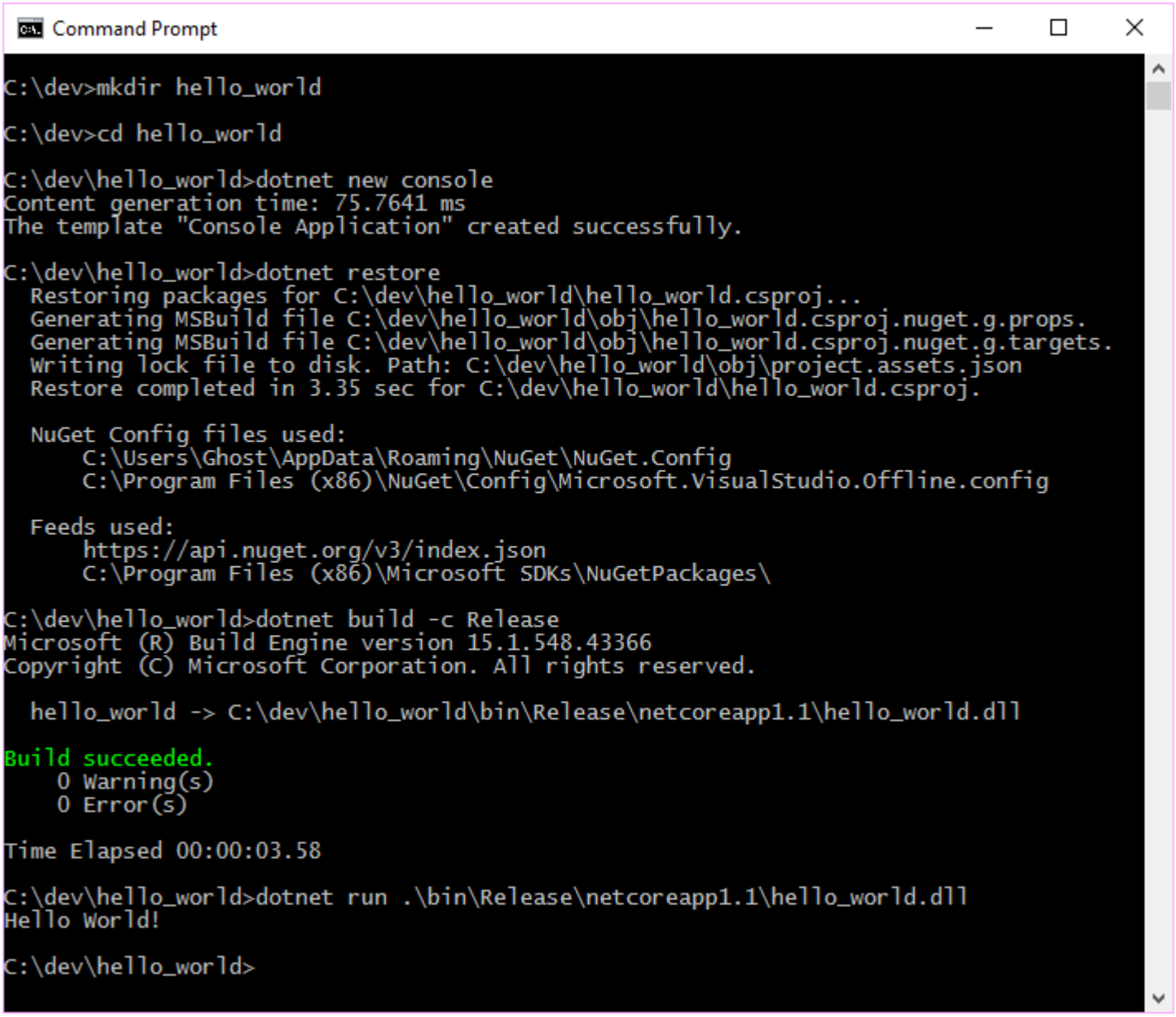
   - **Program.cs**

```
using System;

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

3. `dotnet restore` die benötigten Pakete mit `dotnet restore` .
4. *Optional* Erstellen Sie die Anwendung mit `dotnet build` für Debug oder `dotnet build -c Release` für Release. `dotnet run` führt auch den Compiler aus und wirft Build-Fehler aus, falls welche gefunden werden.
5. Führen Sie die Anwendung mit `dotnet run` für Debug oder `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` für Release

---

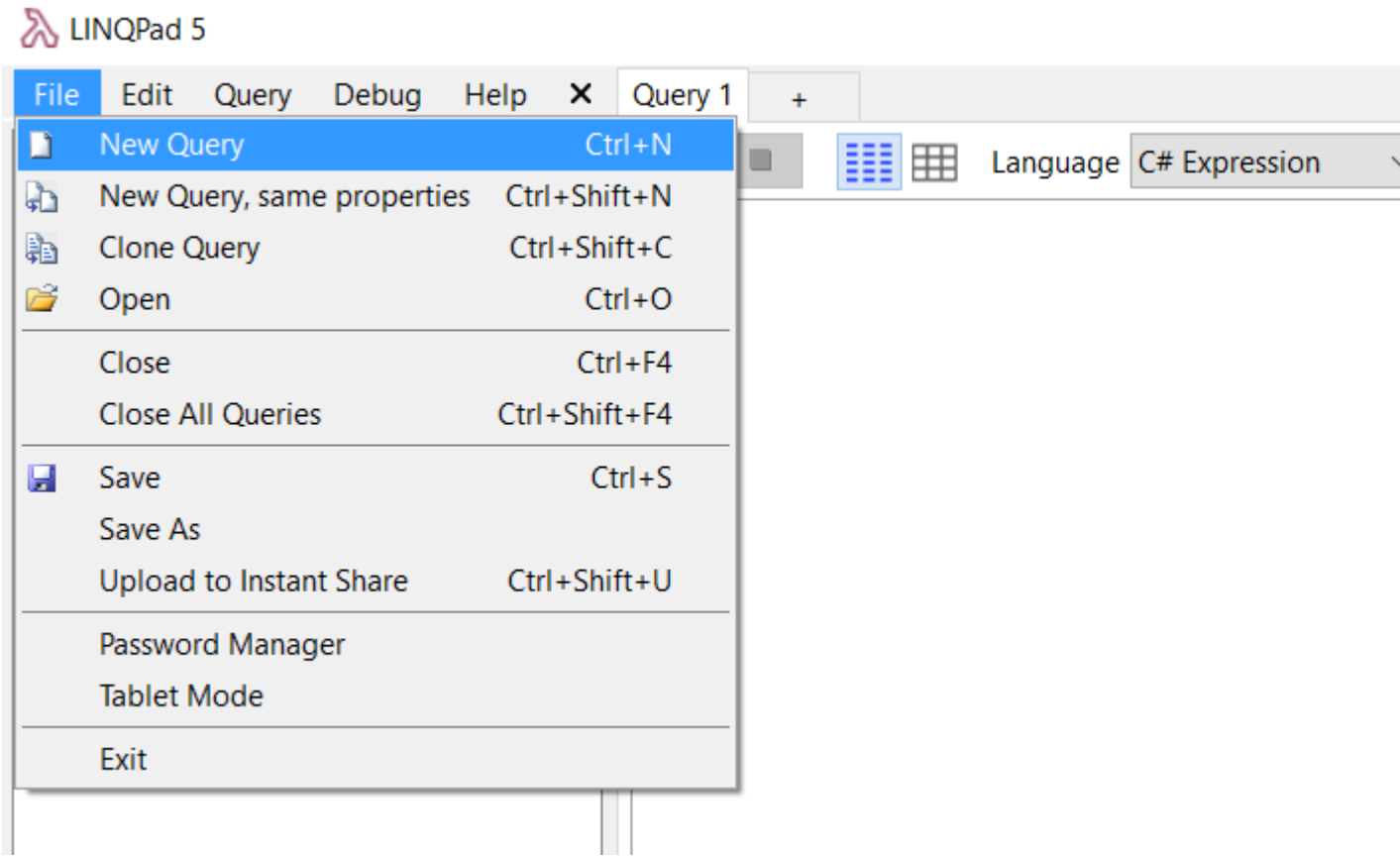
Ausgabe der Eingabeaufforderung



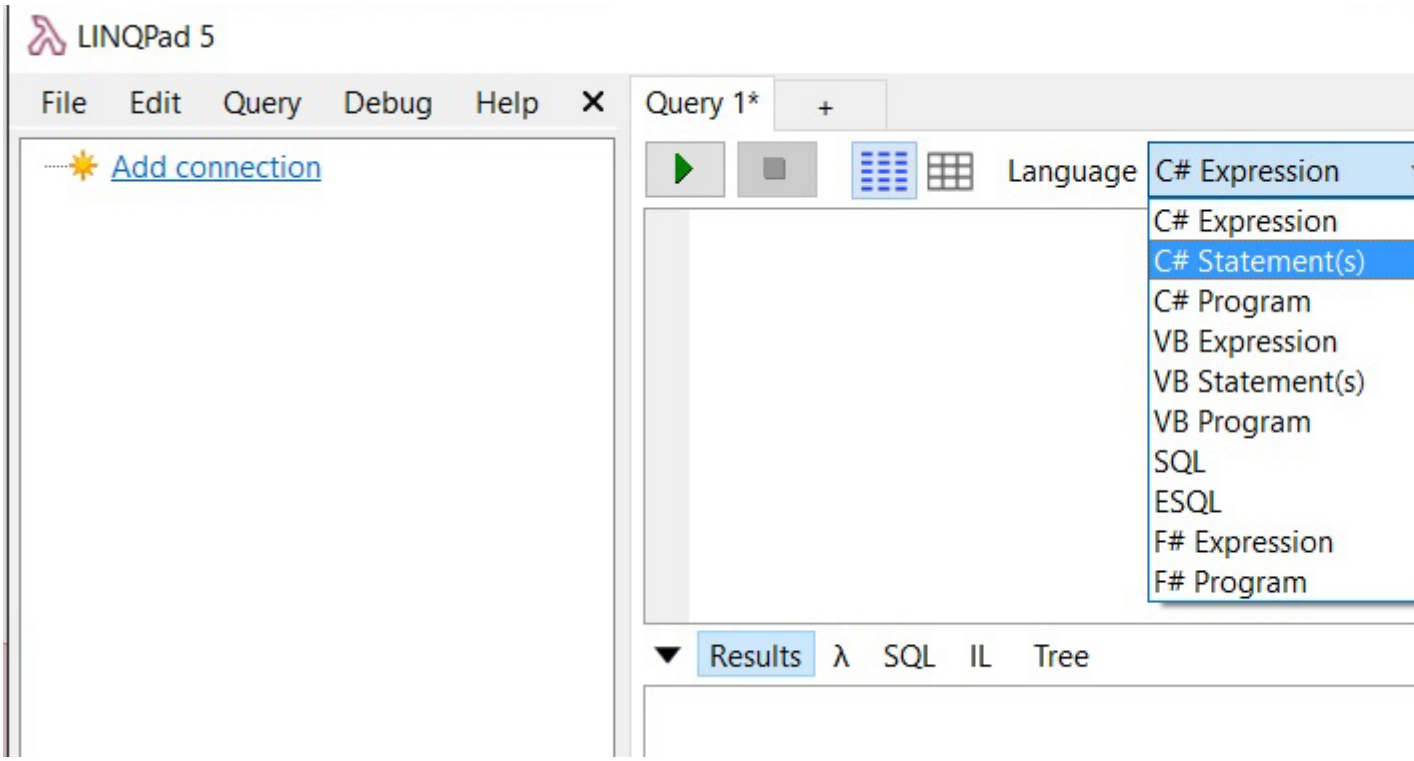
Neue Abfrage mit LinqPad erstellen

LinqPad ist ein großartiges Tool, mit dem Sie die Funktionen von .Net-Sprachen (C #, F # und VB.Net) lernen und testen können.

1. Installieren Sie LinqPad
2. Neue Abfrage erstellen (`Strg` + `N`)

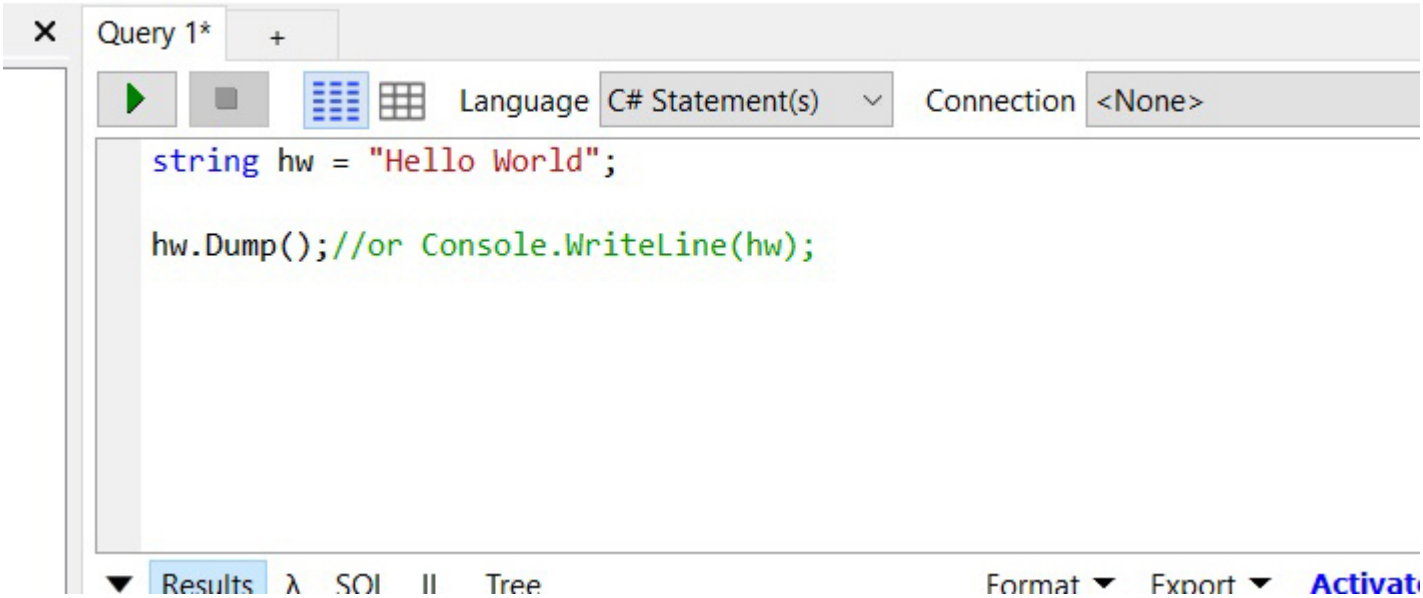


3. Wählen Sie unter Sprache "C # -Anweisungen" aus.

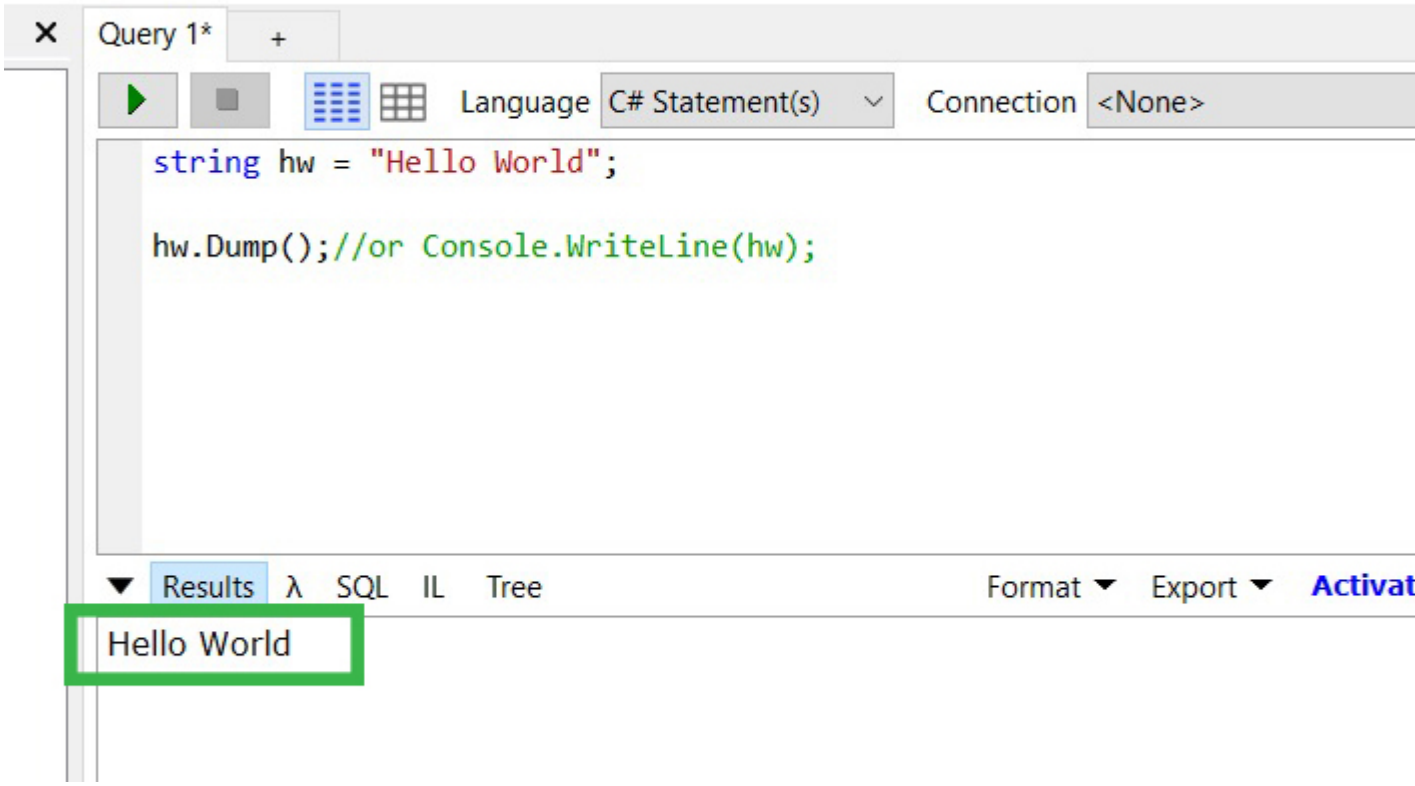


4. Geben Sie den folgenden Code ein und drücken Sie "Ausführen" (`F5`).

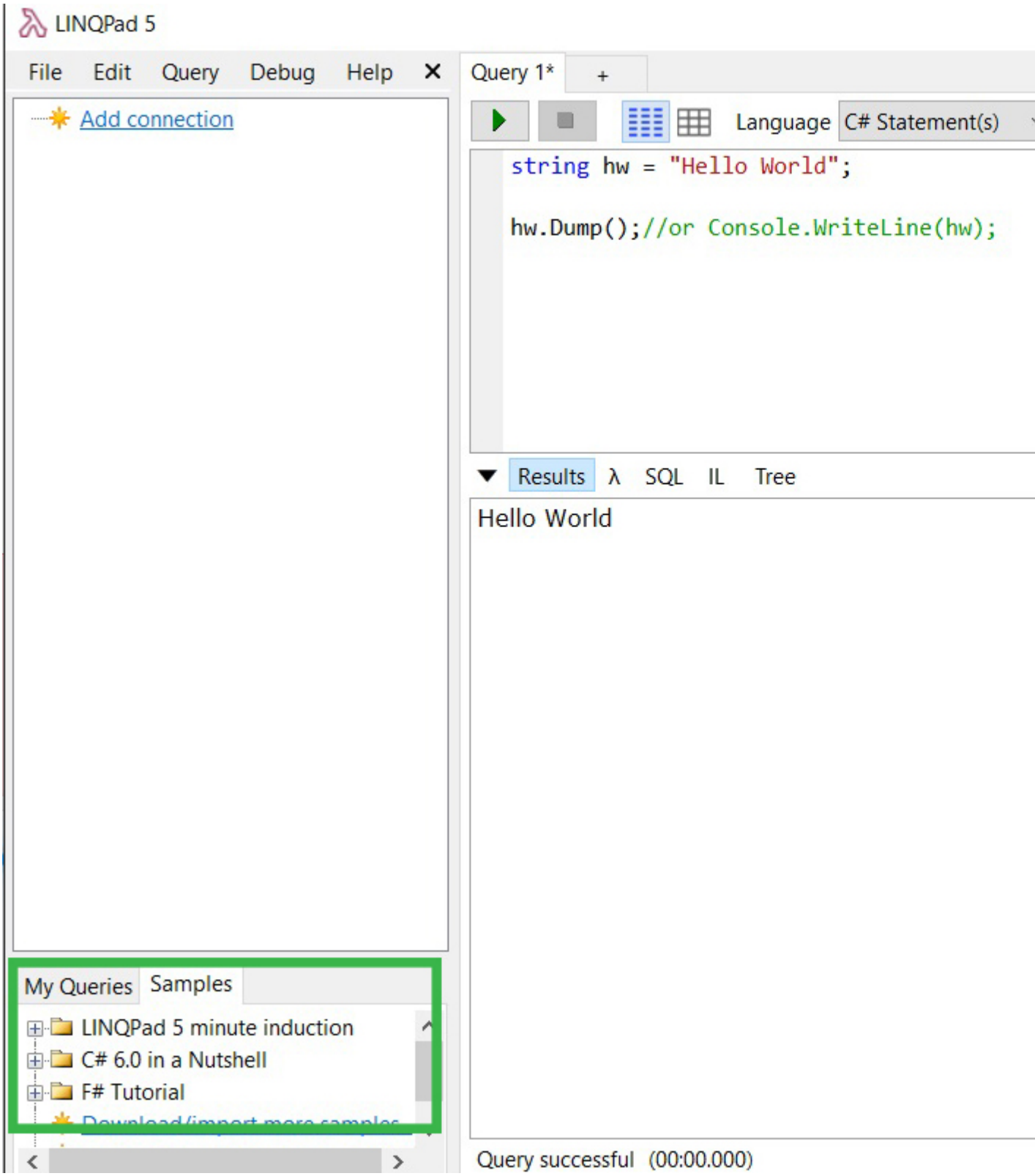
```
string hw = "Hello World";

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



5. Sie sollten "Hello World" auf dem Ergebnisbildschirm anzeigen.

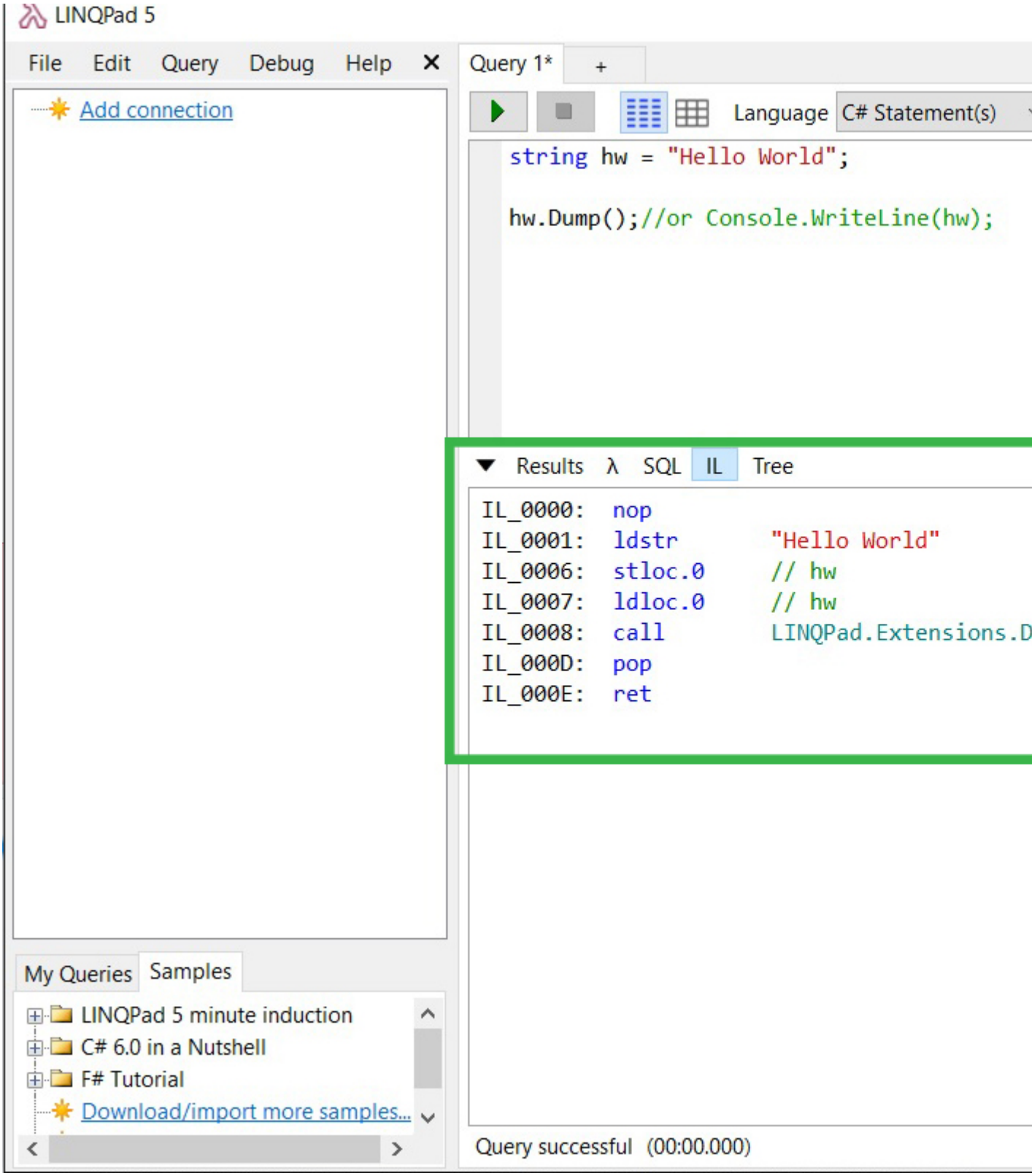


6. Nachdem Sie Ihr erstes .Net-Programm erstellt haben, können Sie die in LinqPad enthaltenen Beispiele über den Browser "Samples" überprüfen. Es gibt viele großartige Beispiele, die Ihnen viele verschiedene Funktionen der .Net-Sprachen zeigen.



Anmerkungen:

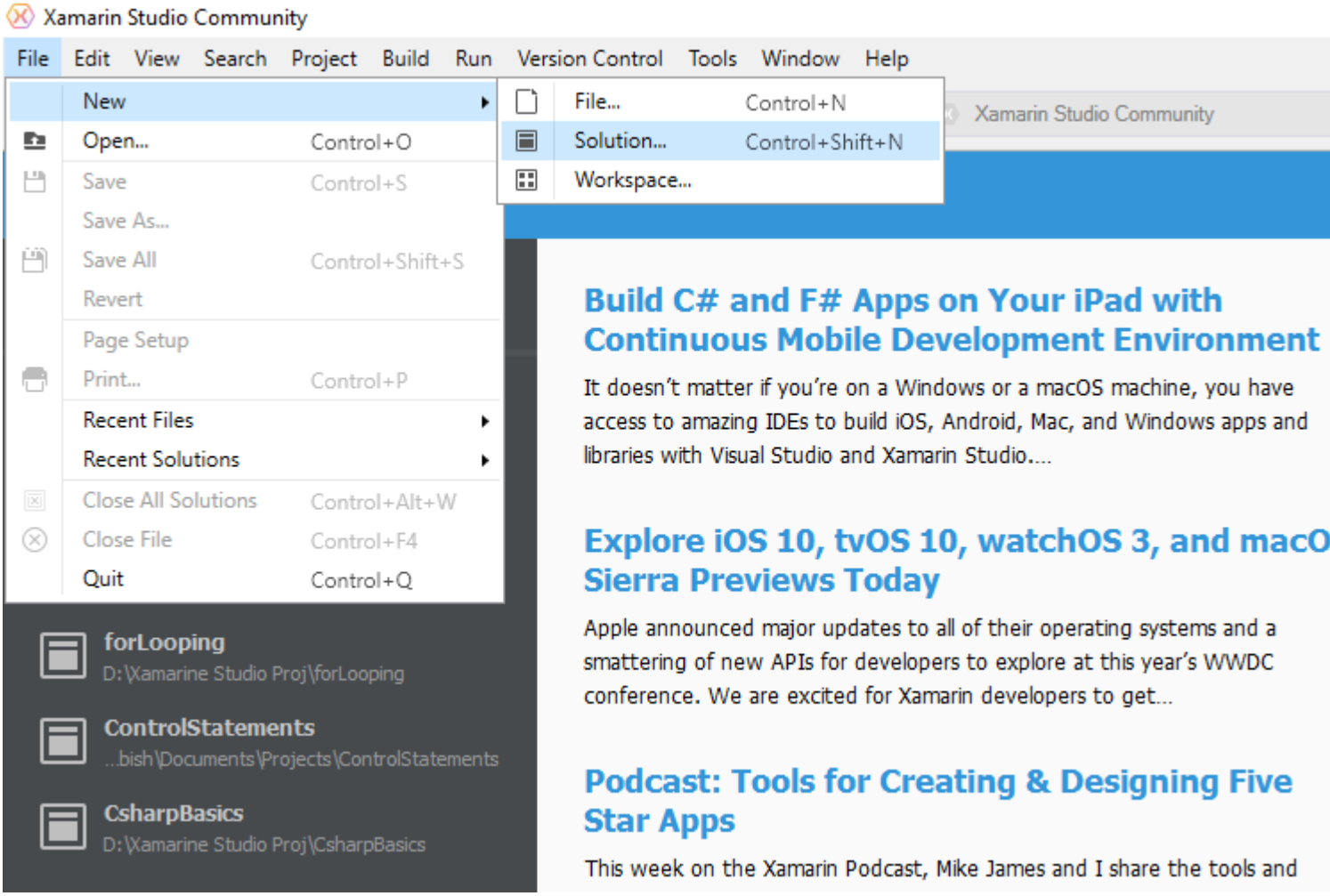
1. Wenn Sie auf "IL" klicken, können Sie den von Ihrem .net-Code generierten IL-Code überprüfen. Dies ist ein großartiges Lernwerkzeug.



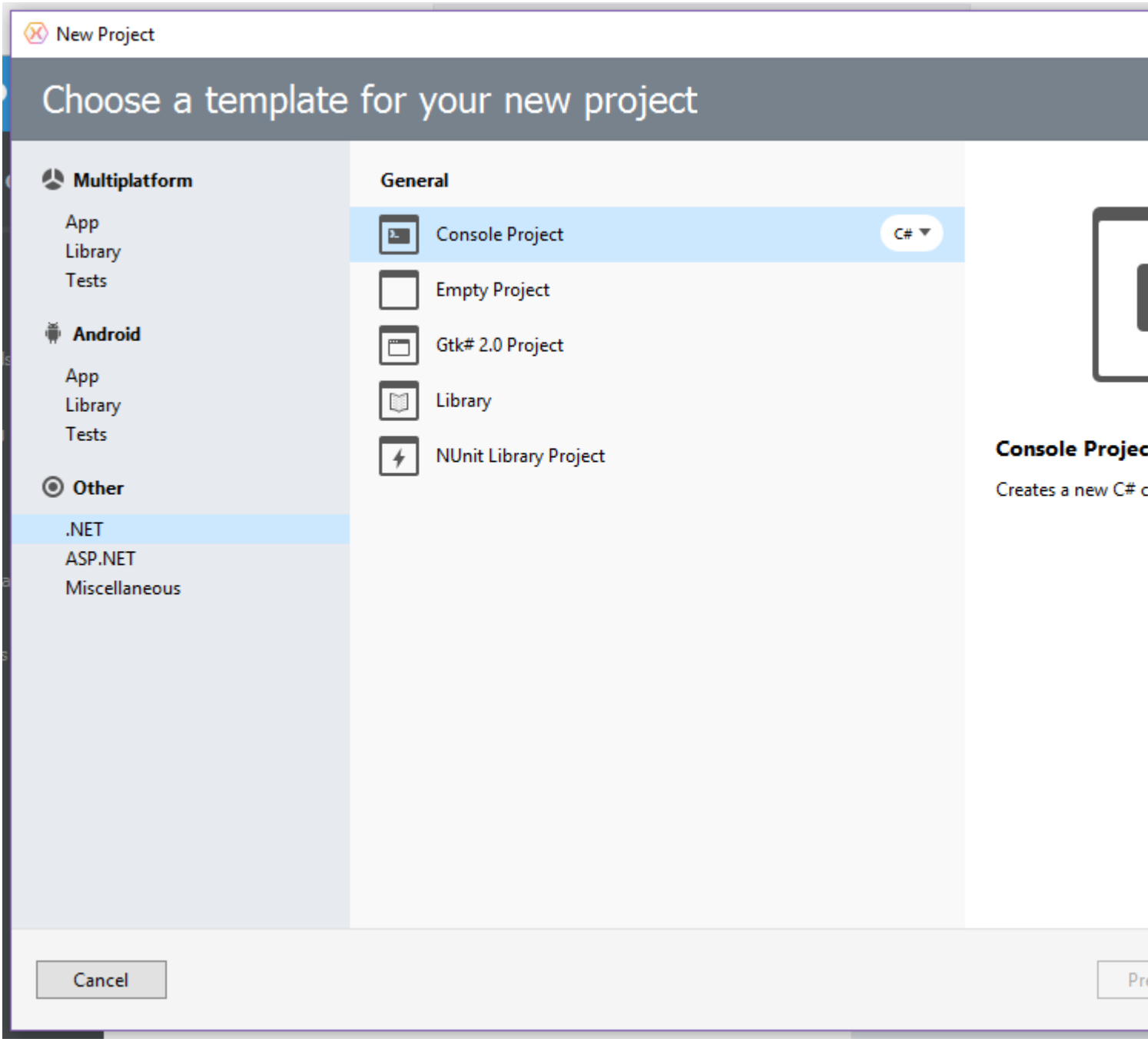
2. Wenn Sie `LINQ to SQL` oder `Linq to Entities` , können Sie die erzeugte SQL überprüfen. `Linq to Entities` ist eine weitere gute Möglichkeit, LINQ kennenzulernen.

Ein neues Projekt mit Xamarin Studio erstellen

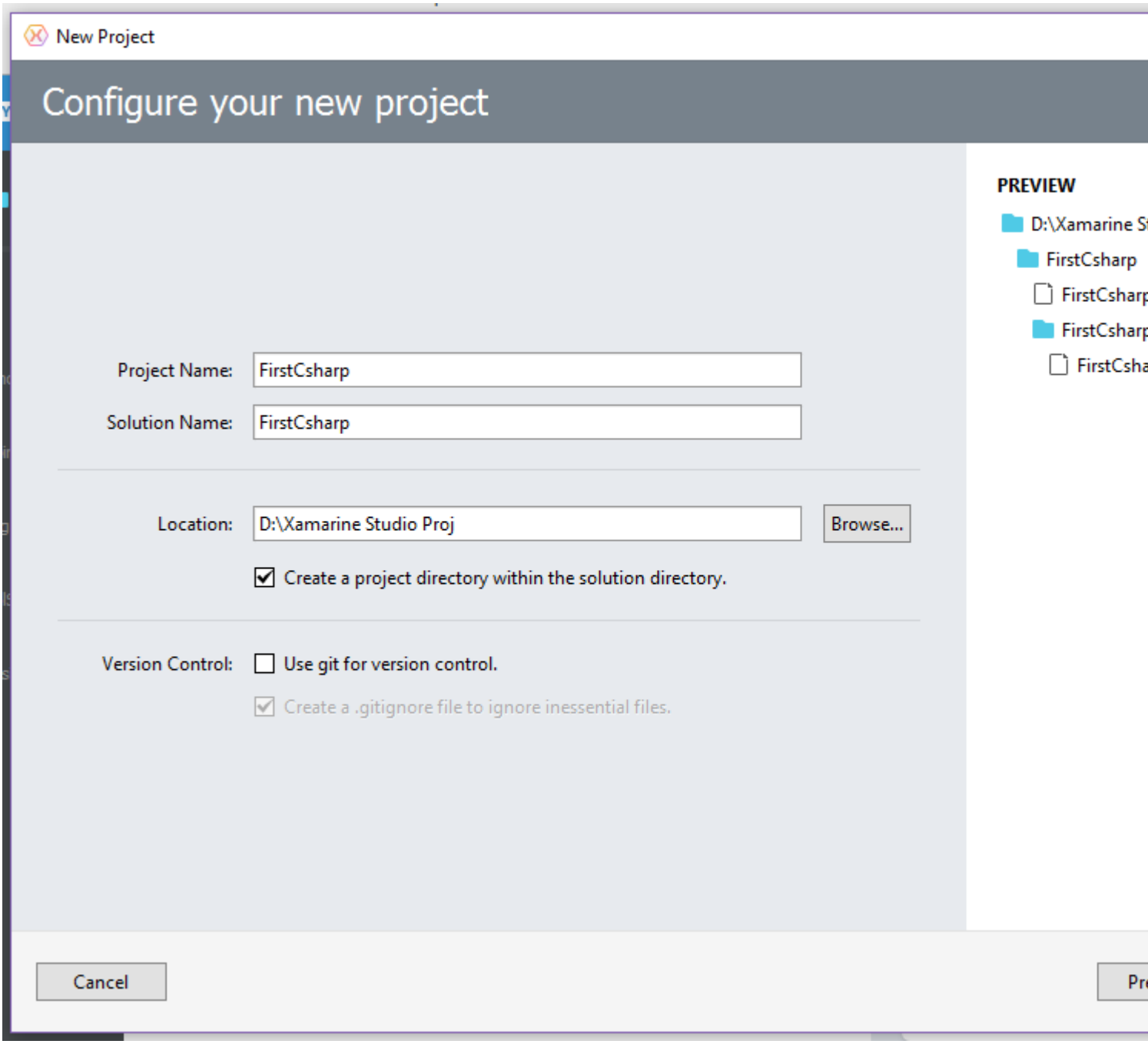
1. Laden Sie die Xamarin Studio Community herunter und installieren Sie sie.
2. Öffnen Sie Xamarin Studio.
3. Klicken Sie auf **Datei** → **Neu** → **Lösung** .



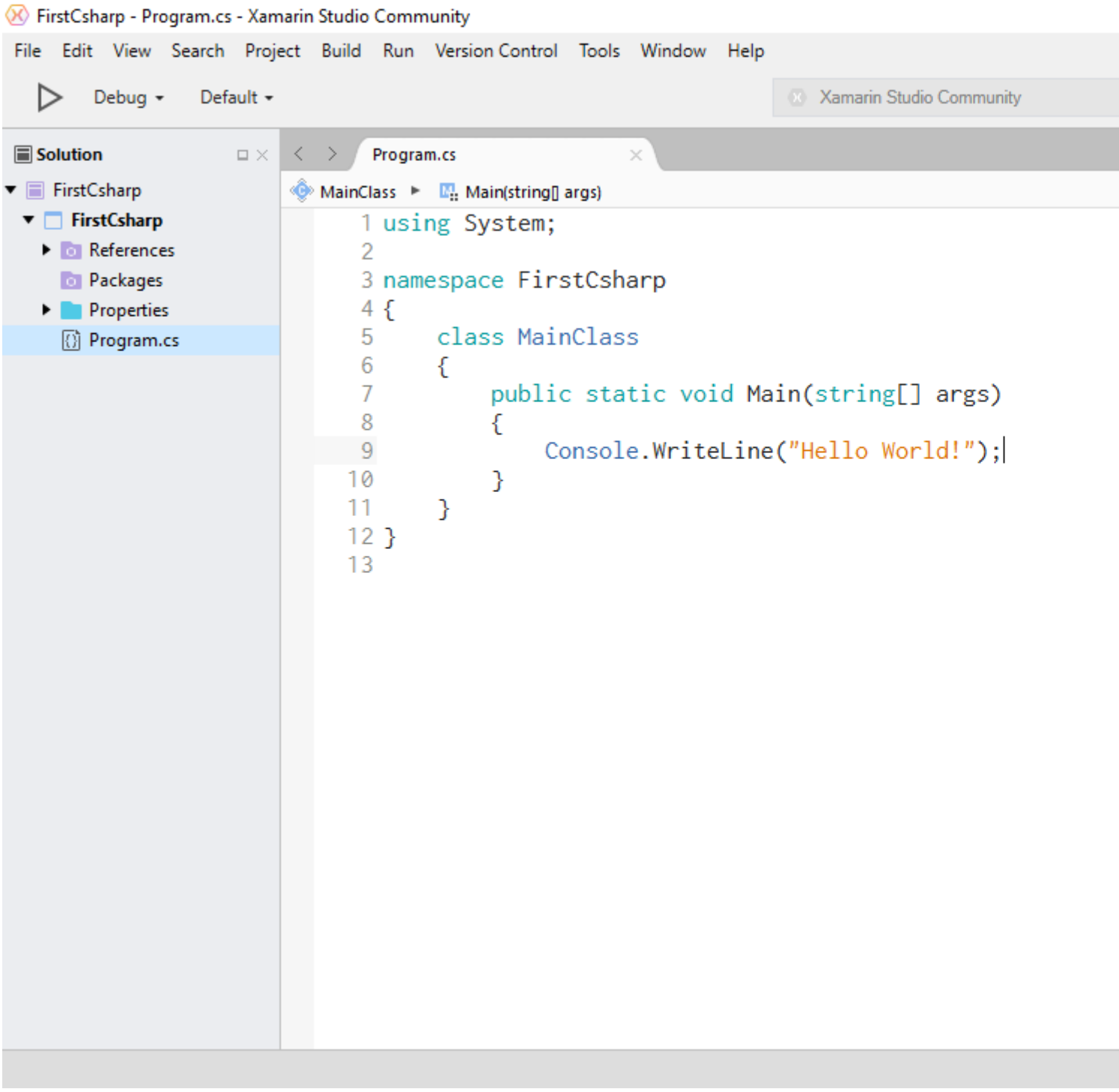
4. Klicken Sie auf **.NET** → **Console Project** und wählen Sie **C #** .
5. Klicken Sie auf `Weiter`, um fortzufahren.



6. Geben Sie die **Projektnamen** und `Durchsuchen ...` für einen **Ort** zu speichern und anschließend auf `Erstellen`.



7. Das neu erstellte Projekt sieht ähnlich aus:

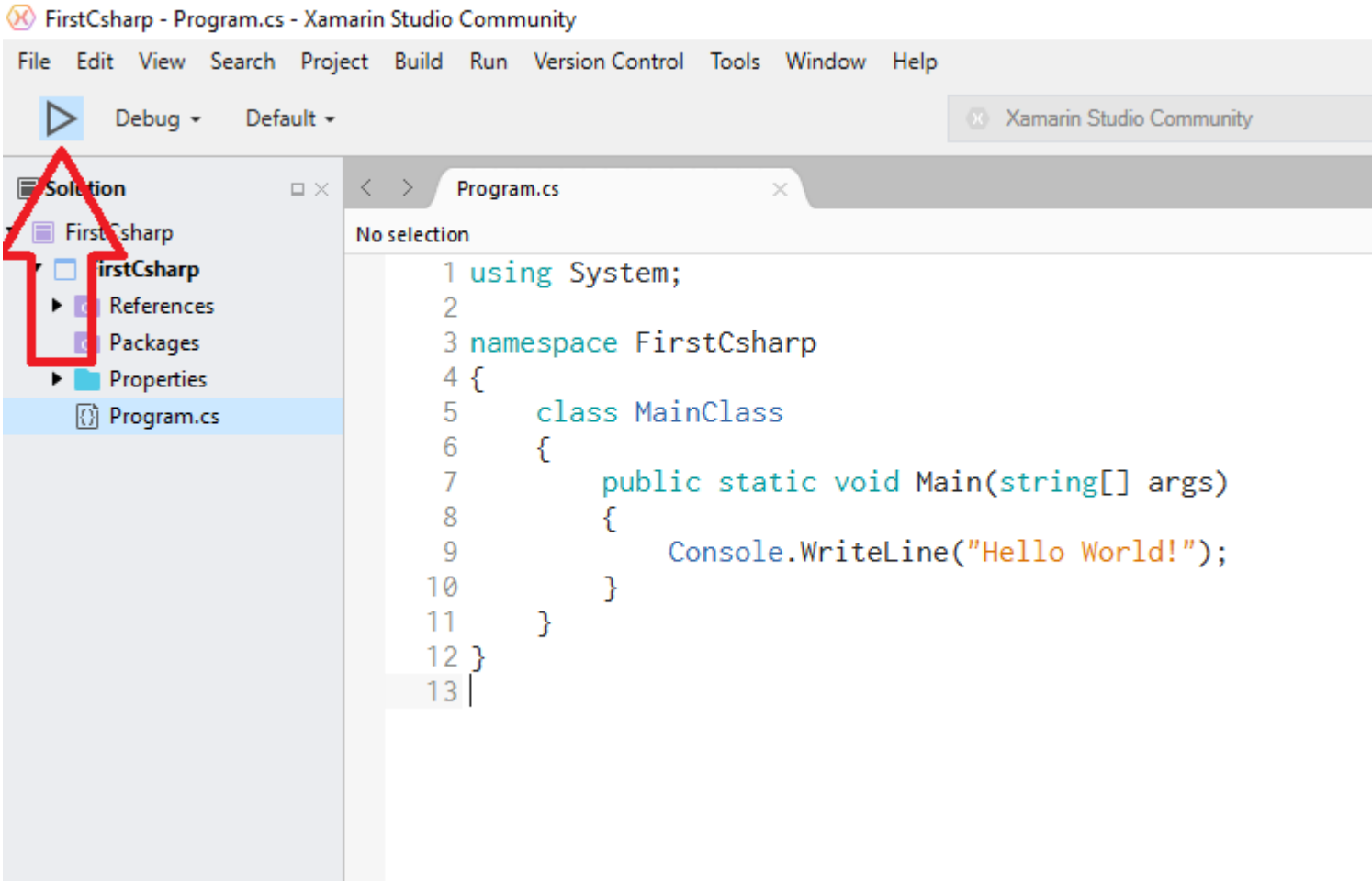


8. Dies ist der Code im Texteditor:

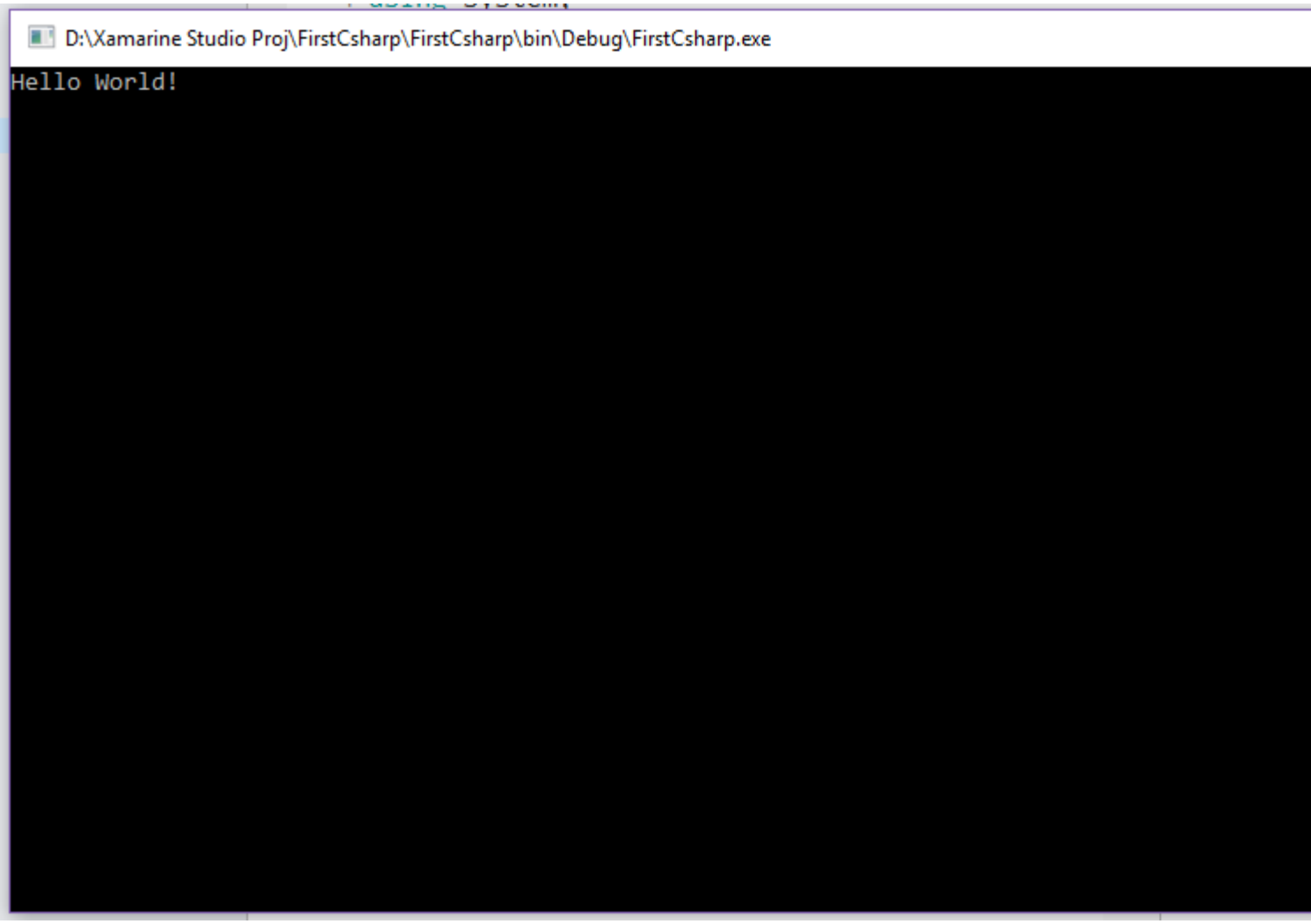
```
using System;

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

```
}
```

9. Um den Code auszuführen, drücken Sie `F5` oder klicken Sie auf die **Wiedergabeschaltfläche** (siehe unten):



10. Folgendes ist die Ausgabe:



Erste Schritte mit C # Language online lesen: https://riptutorial.com/de/csharp/topic/15/erste-schritte-mit-c-sharp-language

Kapitel 2: .NET Compiler-Plattform (Roslyn)

Examples

Erstellen Sie einen Arbeitsbereich aus dem MSBuild-Projekt

Besorgen Sie sich zunächst das `Microsoft.CodeAnalysis.CSharp.Workspaces` Nuget, bevor Sie fortfahren.

```
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);
}
```

Um vorhandenen Code in den Arbeitsbereich zu laden, kompilieren und melden Sie Fehler. Danach wird der Code im Speicher abgelegt. Von hier aus kann sowohl mit der syntaktischen als auch mit der semantischen Seite gearbeitet werden.

Syntaxbaum

Ein **Syntaxbaum** ist eine unveränderliche Datenstruktur, die das Programm als einen Baum mit Namen, Befehlen und Marken darstellt (wie zuvor im Editor konfiguriert).

Angenommen, eine `Microsoft.CodeAnalysis.Compilation` Instanz namens `compilation` wurde konfiguriert. Es gibt mehrere Möglichkeiten, die Namen aller im geladenen Code deklarierten Variablen aufzulisten. Um dies einfach zu tun, nehmen Sie alle Syntaxteile in jedem Dokument (`DescendantNodes` Methode) und verwenden Sie Linq, um Knoten auszuwählen, die die Deklaration von Variablen beschreiben:

```
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);
    }
}
```

Jeder Typ eines C # -Konstrukts mit einem entsprechenden Typ ist im Syntaxbaum vorhanden. Um bestimmte Typen schnell zu finden, verwenden Sie das `Syntax Visualizer` Fenster in Visual

Studio. Dadurch wird das aktuell geöffnete Dokument als Roslyn-Syntaxbaum interpretiert.

Semantisches Modell

Ein **semantisches Modell** bietet eine tiefere Interpretationsebene und Einblick in den Code-Vergleich mit einem Syntaxbaum. Wo Syntaxbäume die Namen von Variablen angeben können, geben Semantikmodelle auch den Typ und alle Referenzen an. Syntaxbäume beachten Methodenaufrufe, aber semantische Modelle geben Hinweise auf den genauen Ort, an dem die Methode deklariert wird (nachdem die Überlastauflösung angewendet wurde)

```
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)"
            }
        }
    }
}
```

Dadurch wird eine Liste lokaler Variablen mit einem Syntaxbaum ausgegeben. Dann konsultiert es das semantische Modell, um den vollständigen Typnamen zu erhalten und alle Referenzen zu jeder Variablen zu finden.

.NET Compiler-Plattform (Roslyn) online lesen: https://riptutorial.com/de/csharp/topic/4886/-net-compiler-plattform--roslyn-

Kapitel 3: Abhängigkeitsspritze

Bemerkungen

Wikipedia-Definition der Abhängigkeitsinjektion ist:

Beim Software-Engineering ist Abhängigkeitsinjektion ein Software-Entwurfsmuster, das eine Inversion der Steuerung zur Auflösung von Abhängigkeiten implementiert. Eine Abhängigkeit ist ein Objekt, das verwendet werden kann (ein Dienst). Eine Injektion ist die Weitergabe einer Abhängigkeit an ein abhängiges Objekt (einen Client), das diese verwenden würde.

** **Diese Website enthält eine Antwort auf die Frage Wie ist Abhängigkeitsinjektion für einen 5-jährigen Patienten zu erklären. Die am besten bewertete Antwort von John Munsch bietet eine überraschend genaue Analogie für den (imaginären) fünfjährigen Inquisitor: Wenn Sie sich selbst etwas aus dem Kühlschrank holen, können Sie Probleme verursachen. Sie könnten die Tür offen lassen, Sie könnten etwas bekommen, das Mama oder Papa nicht haben wollen. Vielleicht suchen Sie sogar nach etwas, das wir nicht haben oder das abgelaufen ist. Was Sie tun sollten, ist ein Bedürfnis: "Ich brauche etwas zu Mittag zu trinken" und dann sorgen wir dafür, dass Sie etwas haben, wenn Sie sich zum Essen setzen. Für die objektorientierte Softwareentwicklung bedeutet dies Folgendes: Zusammenarbeitende Klassen (die Fünfjährigen) sollten sich auf die Infrastruktur (die Eltern) verlassen, die sie zur Verfügung stellen**

** Dieser Code verwendet MEF, um die DLL dynamisch zu laden und die Abhängigkeiten aufzulösen. Die ILogger-Abhängigkeit wird von MEF aufgelöst und in die Benutzerklasse eingefügt. Die Benutzerklasse erhält nie eine konkrete Implementierung von ILogger und hat keine Ahnung, welche Art von Logger sie verwendet. **

Examples

Abhängigkeitsinjektion mit MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
```

```
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);
```

```
        //Fill the imports of this object
        try
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
}
```

Abhängigkeit Injection C # und ASP.NET mit Unity

Warum sollten wir in unserem Code die Depedenzinjektion verwenden? Wir möchten andere Komponenten von anderen Klassen unseres Programms entkoppeln. Zum Beispiel haben wir die Klasse AnimalController, die folgenden Code hat:

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

Wir betrachten diesen Code und denken, dass alles in Ordnung ist, aber jetzt ist unser AnimalController auf das Objekt _SantaAndHisReindeer angewiesen. Automatisch ist mein Controller schlecht zum Testen und die Wiederverwendbarkeit meines Codes wird sehr schwierig sein.

Sehr gute Erklärung , warum sollten wir Depedency Injection und Schnittstellen verwenden hier .

Wenn wir möchten, dass Unity mit DI umgeht, ist der Weg dafür sehr einfach :) Mit NuGet (Paketmanager) können wir Unity problemlos in unseren Code importieren.

> in Visual Studio Tools -> NuGet Package Manager -> Verwalten von Paketen für Lösungen -> in der Eingabe-Eingabeeinheit schreiben -> Wählen Sie unser Projekt -> Klicken Sie auf Installieren

Nun werden zwei Dateien mit schönen Kommentaren erstellt.

> im App-Data-Ordner UnityConfig.cs und UnityMvcActivator.cs
>
> UnityConfig - In der RegisterTypes-Methode sehen wir den Typ, der in unsere Konstruktoren injiziert wird.

```
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 -> auch mit schönen Kommentaren, die besagen, dass diese Klasse Unity mit ASP.NET MVC integriert

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
```

```
        var container = UnityConfig.GetConfiguredContainer();


FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().Firs

        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRec

    }

    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

Jetzt können wir unseren Controller von der Klasse SantAndHisReindeer entkoppeln :)

```
 public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

Vor dem Ausführen unserer Anwendung müssen Sie noch eine letzte Maßnahme ausführen.

In Global.asax.cs müssen wir eine neue Zeile hinzufügen: UnityWebActivator.Start (), die startet, Unity konfiguriert und unsere Typen registriert.

```
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();
        }
    }
}
```

Abhängigkeitsspritze online lesen:
https://riptutorial.com/de/csharp/topic/5766/abhangigkeitsspritze

Kapitel 4: Aktionsfilter

Examples

Benutzerdefinierte Aktionsfilter

Wir schreiben benutzerdefinierte Aktionsfilter aus verschiedenen Gründen. Möglicherweise verfügen wir über einen benutzerdefinierten Aktionsfilter zum Protokollieren oder zum Speichern von Daten in der Datenbank vor einer Aktionsausführung. Wir könnten auch eine haben, um Daten aus der Datenbank abzurufen und sie als globale Werte der Anwendung festzulegen.

Um einen benutzerdefinierten Aktionsfilter zu erstellen, müssen wir die folgenden Aufgaben ausführen:

1. Erstellen Sie eine Klasse
2. Vererben Sie es von der ActionFilterAttribute-Klasse

Überschreiben Sie mindestens eine der folgenden Methoden:

OnActionExecuting - Diese Methode wird aufgerufen, bevor eine Controller-Aktion ausgeführt wird.

OnActionExecuted - Diese Methode wird aufgerufen, nachdem eine Controller-Aktion ausgeführt wurde.

OnResultExecuting - Diese Methode wird aufgerufen, bevor ein Controller-Aktionsergebnis ausgeführt wird.

OnResultExecuted - Diese Methode wird aufgerufen, nachdem ein Controller-Aktionsergebnis ausgeführt wurde.

Der Filter kann wie in der folgenden Liste gezeigt erstellt werden:

```
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);
            }
        }
    }
```

Aktionsfilter online lesen: https://riptutorial.com/de/csharp/topic/1505/aktionsfilter

Kapitel 5: Aliase von eingebauten Typen

Examples

Tabelle mit eingebauten Typen

Die folgende Tabelle zeigt die Schlüsselwörter für integrierte `C#` -Typen, bei denen es sich um Aliase vordefinierter Typen in den System-Namespaces handelt.

C # -Typ	.NET Framework-Typ
bool	System.Boolean
Byte	System.Byte
sbyte	System.SByte
verkohlen	System.Char
Dezimal	System.Decimal
doppelt	System.Doppel
schweben	System.Single
int	System.Int32
uint	System.UInt32
lange	System.Int64
ulong	System.UInt64
Objekt	System.Object
kurz	System.Int16
ushort	System.UInt16
Schnur	System.String

Die Schlüsselwörter vom Typ `C#` und ihre Aliasnamen sind austauschbar. Sie können beispielsweise eine Integer-Variable mit einer der folgenden Deklarationen deklarieren:

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

Aliase von eingebauten Typen online lesen: https://riptutorial.com/de/csharp/topic/1862/aliase-von-eingebauten-typen

Kapitel 6: Allgemeine Zeichenkettenoperationen

Examples

Einen String nach bestimmten Zeichen aufteilen

```
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?"]
```

Teilstrings einer gegebenen Zeichenfolge abrufen

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

`Substring` gibt den String aus einem bestimmten Index oder zwischen zwei Indizes (beide inklusive) zurück.

Bestimmen Sie, ob eine Zeichenfolge mit einer angegebenen Sequenz beginnt

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Einen String innerhalb eines Strings finden

Mit dem `System.String.Contains` Sie herausfinden, ob eine bestimmte Zeichenfolge in einer Zeichenfolge vorhanden ist. Die Methode gibt einen booleschen Wert zurück, true, wenn die Zeichenfolge vorhanden ist, andernfalls false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Abschneiden unerwünschter Zeichen am Anfang und / oder Ende von Zeichenfolgen.

`String.Trim()`

```
string x = "   Hello World!    ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

`String.TrimStart()` und `String.TrimEnd()`

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

Eine Zeichenfolge formatieren

Verwenden Sie die `String.Format()` Methode, um ein oder mehrere Elemente in der Zeichenfolge durch die Zeichenfolgendarstellung eines angegebenen Objekts zu ersetzen:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Ein String-Array zu einem neuen zusammenfügen

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Auffüllen einer Zeichenfolge auf eine feste Länge

```
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)
```

Konstruieren Sie einen String aus Array

Die `String.Join` Methode hilft uns beim `String.Join` einer Zeichenfolge aus einem Array / einer Liste von Zeichen oder einer Zeichenfolge. Diese Methode akzeptiert zwei Parameter. Der erste ist das Trennzeichen oder das Trennzeichen, das Ihnen hilft, jedes Element im Array zu trennen. Der zweite Parameter ist das Array selbst.

Zeichenfolge aus dem `char array` :

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Ausgabe : `a,b,c` Wenn wir das `delimiter` als `""` ändern, wird die Ausgabe zu `abc` .

Zeichenfolge aus der `List of char` :

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Ausgabe : `a|b|c`

String aus der `List of Strings` :

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

Ausgabe : `Ram is a boy`

`array of strings` aus einem String- `array of strings` :

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

Ausgabe : `Ram_is_a_boy`

Formatierung mit ToString

Normalerweise verwenden wir die `String.Format` Methode zum Formatieren. Der `.ToString` wird normalerweise zum Konvertieren anderer Typen in eine Zeichenfolge verwendet. Wir können das Format zusammen mit der ToString-Methode angeben, während die Konvertierung stattfindet. So können wir eine zusätzliche Formatierung vermeiden. Lassen Sie mich erklären, wie es mit verschiedenen Typen funktioniert.

Ganzzahl für formatierte Zeichenfolge:

```
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"
```

doppelt zu formatierter String:

```
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
```

DateTime mit ToString formatieren

```
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"
```

X-Zeichen von der rechten Seite einer Zeichenfolge abrufen

Visual Basic verfügt über Left-, Right- und Mid-Funktionen, die Zeichen von Left, Right und Middle einer Zeichenfolge zurückgeben. Diese Methoden sind in C # nicht vorhanden, können jedoch mit `Substring()` implementiert werden. Sie können als Erweiterungsmethoden wie die folgenden implementiert werden:

```
   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 );

     }
   }
```

Diese Erweiterungsmethode kann wie folgt verwendet werden:

```
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"
```

Suchen nach leerem String mithilfe von String.IsNullOrEmpty () und String.IsNullOrWhiteSpace ()

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;

result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

Ein Zeichen an einem bestimmten Index abrufen und die Zeichenfolge auflisten

Sie können die `Substring` Methode verwenden, um an einer beliebigen Stelle eine beliebige Anzahl von Zeichen aus einer Zeichenfolge `Substring` . Wenn Sie jedoch nur ein einzelnes Zeichen wünschen, können Sie den String-Indexer verwenden, um ein einzelnes Zeichen an einem bestimmten Index zu erhalten, wie Sie es bei einem Array tun:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Beachten Sie, dass der Rückgabetyp `char` , im Gegensatz zur `Substring` Methode, die einen `string` Typ zurückgibt.

Sie können den Indexer auch verwenden, um die Zeichen der Zeichenfolge zu durchlaufen:

```
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
**********/
```

Konvertieren Sie die Dezimalzahl in das Binär-, Oktal- und Hexadezimal-Format

1. Um die Dezimalzahl in ein Binärformat umzuwandeln, verwenden Sie **Basis 2**

```
Int32 Number = 15;
Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
```

2. Um die Dezimalzahl in das Oktalformat umzuwandeln, verwenden Sie die **Basis 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
```

3. Um eine Dezimalzahl in ein Hexadezimalformat zu konvertieren, verwenden Sie die **Basis 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

Einen String durch einen anderen String teilen

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Ergebnis:

> ["dies", "ist", "ein", "vollständig", "Satz"]

Zeichenkette richtig umkehren

Meistens, wenn Leute eine Zeichenkette umkehren müssen, tun sie es mehr oder weniger so:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

Was diese Leute jedoch nicht erkennen, ist, dass dies tatsächlich falsch ist.
Und ich meine nicht wegen des fehlenden NULL-Checks.

Es ist eigentlich falsch, weil ein Glyph / GraphemeCluster aus mehreren Codepunkten (auch als Zeichen) bestehen kann.

Um zu verstehen, warum dies so ist, müssen wir uns zunächst der Tatsache bewusst sein, was der Begriff "Charakter" eigentlich bedeutet.

Referenz:

> Charakter ist ein überladener Begriff, der viele Dinge bedeuten kann.
>
> Ein Codepunkt ist die atomare Informationseinheit. Text ist eine Folge von Codepunkten. Jeder Codepunkt ist eine Zahl, die vom Unicode-Standard eine Bedeutung erhält.

> Ein Graphem ist eine Folge von einem oder mehreren Codepunkten, die als eine einzige grafische Einheit angezeigt werden, die ein Lesegerät als ein einzelnes Element des Schreibsystems erkennt. Zum Beispiel sind sowohl a als auch ä Grapheme, sie können jedoch aus mehreren Codepunkten bestehen (z. B. kann ä zwei Codepunkte sein, einer für das Basiszeichen a, gefolgt von einem für die Diaresis), es gibt aber auch einen alternativen, älteren Code Punkt, der dieses Diagramm darstellt). Einige Codepunkte sind niemals Teil eines Graphems (z. B. Nicht-Joiner mit Nullbreite oder Richtungsüberschreibungen).
>
> Eine Glyphe ist ein Bild, das normalerweise in einer Schrift (die eine Sammlung von Glyphen ist) gespeichert wird und zur Darstellung von Graphemen oder Teilen davon dient. Schriften können mehrere Glyphen zu einer einzigen Darstellung zusammensetzen. Wenn der obige ä ein einzelner Codepunkt ist, kann ein Zeichensatz diese als zwei separate, räumlich überlagerte Glyphen darstellen. Bei OTF enthalten die GSUB- und GPOS-Tabellen der Schriftart Ersetzungs- und Positionierungsinformationen, damit dies funktioniert. Ein Zeichensatz kann auch mehrere alternative Glyphen für dasselbe Graphem enthalten.

In C # ist ein Zeichen also eigentlich ein CodePoint.

Das bedeutet, wenn Sie nur eine gültige Zeichenfolge wie `Les Misé rables` umkehren, die so aussehen kann

```
string s = "Les Mise\u0301rables";
```

Als Folge von Zeichen erhalten Sie:

> selbaŕesiM seL

Wie Sie sehen, liegt der Akzent auf dem R-Zeichen anstelle des E-Zeichens.
Obwohl string.reverse.reverse die ursprüngliche Zeichenfolge ergibt, wenn Sie beide Male das Zeichen-Array umkehren, ist diese Art der Umkehrung definitiv NICHT die Umkehrung der ursprünglichen Zeichenfolge.

Sie müssen nur jedes GraphemeCluster umkehren.
Wenn Sie dies richtig machen, kehren Sie eine Zeichenfolge folgendermaßen um:

```
    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);
    }
```

Und - oh Freude - Sie werden feststellen, wenn Sie es richtig machen, funktioniert es auch für asiatische / südasiatische / ostasiatische Sprachen (und Französisch / Schwedisch / Norwegisch usw.) ...

Ersetzen eines Strings innerhalb eines Strings

Mit der Methode `System.String.Replace` können Sie einen Teil einer Zeichenfolge durch eine andere Zeichenfolge ersetzen.

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Alle Vorkommen der Suchzeichenfolge werden ersetzt.

Diese Methode kann auch zum Entfernen eines Teils einer Zeichenfolge verwendet werden, indem das Feld `String.Empty` wird:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Ändern der Groß- / Kleinschreibung von Zeichen in einem String

Die `System.String` Klasse unterstützt eine Reihe von Methoden zum Konvertieren zwischen Groß- und Kleinbuchstaben in einer Zeichenfolge.

- `System.String.ToLowerInvariant` wird verwendet, um ein in Kleinbuchstaben konvertiertes

String-Objekt zurückzugeben.

- `System.String.ToUpperInvariant` wird verwendet, um ein in Großbuchstaben konvertiertes String-Objekt zurückzugeben.

Anmerkung: Der Grund für die Verwendung der *invarianten* Versionen dieser Methoden besteht darin, die Erzeugung unerwarteter kulturspezifischer Buchstaben zu verhindern. Dies wird hier ausführlich erklärt .

Beispiel:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Beachten Sie, dass Sie beim Konvertieren in Klein- und Großbuchstaben eine bestimmte **Kultur** angeben *können* , indem Sie die Methoden String.ToLower (CultureInfo) und String.ToUpper (CultureInfo) verwenden.

Verketten Sie ein String-Array zu einem String

Die `System.String.Join` Methode ermöglicht die Verkettung aller Elemente in einem String-Array unter Verwendung eines angegebenen Trennzeichens zwischen jedem Element:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

String-Verkettung

Zeichenfolgenverkettung kann mithilfe der Methode `System.String.Concat` oder (viel einfacher) mit dem Operator `+` werden:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

In C # 6 kann dies wie folgt durchgeführt werden:

```
string concat = $"{first},{second}";
```

Allgemeine Zeichenkettenoperationen online lesen:
https://riptutorial.com/de/csharp/topic/73/allgemeine-zeichenkettenoperationen

Kapitel 7: Anonyme Typen

Examples

Anonymen Typ erstellen

Da anonyme Typen nicht benannt werden, müssen die Variablen dieser Typen implizit typisiert werden (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

Wenn die Mitgliedsnamen nicht angegeben sind, werden sie auf den Namen der Eigenschaft / Variablen gesetzt, mit der das Objekt initialisiert wird.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Beachten Sie, dass Namen nur ausgelassen werden können, wenn der Ausdruck in der anonymen Typdeklaration ein einfacher Eigenschaftszugriff ist. Für Methodenaufrufe oder komplexere Ausdrücke muss ein Eigenschaftsname angegeben werden.

```
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
```

Anonym vs. Dynamik

Mit anonymen Typen können Objekte erstellt werden, ohne dass sie vorab explizit definiert werden müssen, während die statische Typprüfung beibehalten wird.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Im Gegensatz dazu hat `dynamic` eine dynamische Typprüfung und entscheidet sich für Laufzeitfehler anstelle von Kompilierzeitfehlern.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Generische Methoden mit anonymen Typen

Generische Methoden ermöglichen die Verwendung anonymer Typen durch Typinferenz.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

Dies bedeutet, dass LINQ-Ausdrücke mit anonymen Typen verwendet werden können:

```
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
```

Instanziieren generischer Typen mit anonymen Typen

Die Verwendung von generischen Konstruktoren erfordert die Benennung der anonymen Typen, was nicht möglich ist. Alternativ können generische Verfahren verwendet werden, um das Auftreten von Typinferenz zu ermöglichen.

```
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);
```

Im Falle von `List<T>` können implizit typisierte Arrays durch die `ToList` LINQ-Methode in eine `List<T> ToList` werden:

```
var list2 = new[] {anon, anon2}.ToList();
```

Anonyme Typgleichheit

Die Gleichheit eines anonymen Typs wird durch die Instanzmethode `Equals` . Zwei Objekte sind gleich, wenn sie den gleichen Typ und gleiche Werte haben (durch `a.Prop.Equals(b.Prop)`) für jede Eigenschaft.

```
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)
```

Zwei anonyme Typen werden nur dann als identisch angesehen, wenn ihre Eigenschaften denselben Namen und Typ haben und in derselben Reihenfolge angezeigt werden.

```
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)
```

Implizit typisierte Arrays

Arrays von anonymen Typen können mit impliziter Typisierung erstellt werden.

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

Anonyme Typen online lesen: https://riptutorial.com/de/csharp/topic/765/anonyme-typen

Kapitel 8: Anweisung verwenden

Einführung

Stellt eine praktische Syntax bereit , die die korrekte Verwendung von IDisposable- Objekten gewährleistet .

Syntax

- mit (Einweg) {}
- using (IDisposable disposable = new MyDisposable ()) {}

Bemerkungen

Das Objekt in der `using` Anweisung muss die `IDisposable` Schnittstelle implementieren.

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

Ausführlichere Beispiele für die `IDisposable` Implementierung finden Sie in den MSDN-Dokumenten .

Examples

Statement-Grundlagen verwenden

`using` syntaktischem Zucker ermöglicht es Ihnen sicherzustellen, dass eine Ressource bereinigt wird, ohne dass eine explizite `try-finally` Blockierung erforderlich ist. Dies bedeutet, dass Ihr Code viel sauberer ist und Sie keine nicht verwalteten Ressourcen verlieren.

Standard `Dispose` Bereinigungsmuster für Objekte, die die `IDisposable` Schnittstelle implementieren (die der `FileStream` -Basisklasse `Stream` in .NET verwendet):

```
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` vereinfacht Ihre Syntax, indem Sie das explizite `try-finally` :

```
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
}
```

Ebenso wie `finally` Blöcke werden immer Fehler und Rückgaben immer ausgeführt, `using` immer `Dispose()` , auch im Fehlerfall:

```
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
}
```

Hinweis: Da garantiert ist, dass `Dispose` unabhängig vom Code-Fluss `Dispose` wird, sollten Sie sicherstellen, dass `Dispose` niemals eine Ausnahme `IDisposable` wenn Sie `IDisposable` implementieren. Andernfalls würde eine tatsächliche Ausnahme von der neuen Ausnahme überschrieben, was zu einem Alptraum für das Debugging führt.

Rückkehr aus dem Block

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

Wegen der Semantik von `try..finally` , auf die die `using` von Block übersetzt, die `return` arbeitet Aussage als erwartete - der Rückgabewert ausgewertet wird , bevor `finally` Block ausgeführt wird , und der Wert angeordnet ist . Die Reihenfolge der Bewertung lautet wie folgt:

1. Bewerten Sie den `try`
2. Werten Sie den zurückgegebenen Wert aus und zwischenspeichern Sie ihn
3. Führen Sie zum Schluss Block aus
4. Gibt den zwischengespeicherten Rückgabewert zurück

Sie können jedoch nicht die Variable zurückgeben `disposable` selbst, da sie ungültig enthalten würden, entsorgt reference - siehe verwandtes Beispiel .

Mehrere using-Anweisungen mit einem Block

Es ist möglich, mehrere verschachtelte `using` Anweisungen zu verwenden, ohne mehrere geschachtelte geschweifte Klammern hinzuzufügen. Zum Beispiel:

```
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
```

Eine Alternative ist zu schreiben:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Was genau dem ersten Beispiel entspricht.

Hinweis: Verschachtelte `using` Anweisungen können die Microsoft-Code-Analysis-Regel CS2002 auslösen (zur Erläuterung siehe diese Antwort) und eine Warnung generieren. Wie in der verknüpften Antwort erläutert, ist das Schachteln `using` Anweisungen im Allgemeinen sicher.

Wenn die Typen in der `using` Anweisung vom gleichen Typ sind, können Sie sie durch Kommas voneinander trennen und den Typ nur einmal angeben (dies ist jedoch nicht üblich):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

Dies kann auch verwendet werden, wenn die Typen eine gemeinsame Hierarchie haben:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

Das Schlüsselwort `var` *kann* im obigen Beispiel *nicht* verwendet werden. Ein Kompilierungsfehler würde auftreten. Sogar die durch Kommas getrennte Deklaration funktioniert nicht, wenn die deklarierten Variablen Typen aus unterschiedlichen Hierarchien haben.

Gotcha: Rückgabe der Ressource, die Sie zur Verfügung haben

Das Folgende ist eine schlechte Idee, da die `db` Variable vor der Rückgabe beseitigt würde.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

Dies kann auch subtilere Fehler verursachen:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

Das sieht in Ordnung aus, aber der Haken ist, dass die LINQ-Ausdrucksauswertung faul ist und möglicherweise erst dann ausgeführt wird, wenn der zugrunde liegende `DBContext` bereits `DBContext` wurde.

Kurz gesagt, der Ausdruck wird nicht ausgewertet, bevor die `using` . Eine mögliche Lösung für dieses Problem, die immer noch verwendet `using` , besteht darin, den Ausdruck sofort auszuwerten, indem eine Methode aufgerufen wird, die das Ergebnis auflistet. Zum Beispiel `ToList()` , `ToArray()` , etc. Wenn Sie die neueste Version von Entity Framework verwenden , können Sie die Verwendung `async` Pendants wie `ToListAsync()` oder `ToArrayAsync()` .

Nachfolgend finden Sie das Beispiel in Aktion:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

Es ist jedoch wichtig anzumerken, dass durch den Aufruf von `ToList()` oder `ToArray()` der Ausdruck eifrig ausgewertet wird, was bedeutet, dass alle Personen mit dem angegebenen Alter in den Speicher geladen werden, auch wenn Sie sie nicht wiederholen.

Die Verwendung von Anweisungen ist nullsicher

Sie müssen das `IDisposable` Objekt nicht auf `null` überprüfen. `using` löst keine Ausnahme aus und `Dispose()` wird nicht aufgerufen:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Gotcha: Exception in Dispose-Methode, die andere Fehler bei der Verwendung von Blöcken maskiert

Betrachten Sie den folgenden Code-Block.

```
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.");
    }
}
```

Sie können erwarten, dass auf der Konsole die Meldung "Vorgang konnte nicht ausgeführt werden" angezeigt werden, tatsächlich würden Sie jedoch sehen, dass "Nicht erfolgreich entsorgt werden konnte". Die Dispose-Methode wird auch dann aufgerufen, wenn die erste Ausnahme ausgelöst wird.

Es ist empfehlenswert, sich dieser Subtilität bewusst zu sein, da sie möglicherweise den tatsächlichen Fehler maskiert, der die Beseitigung des Objekts verhindert und das Debuggen erschwert.

Verwenden von Anweisungen und Datenbankverbindungen

Das Schlüsselwort `using` stellt sicher, dass die in der Anweisung definierte Ressource nur im Geltungsbereich der Anweisung selbst vorhanden ist. Alle in der Anweisung definierten Ressourcen müssen die `IDisposable` Schnittstelle implementieren.

Diese sind beim Umgang mit Verbindungen, die die `IDisposable` Schnittstelle implementieren, `IDisposable` , da sie sicherstellen können, dass die Verbindungen nicht nur ordnungsgemäß geschlossen werden, sondern auch, dass ihre Ressourcen freigegeben werden, nachdem die `using` Anweisung außerhalb des Gültigkeitsbereichs liegt.

Häufige `IDisposable`

Viele der folgenden Klassen sind `IDisposable` Klassen, die die `IDisposable` Schnittstelle implementieren und `IDisposable` Kandidaten für eine `using` Anweisung sind:

- `SqlConnection` , `SqlCommand` , `SqlDataReader` usw.
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` usw.
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` usw.
- `DbContext`

Alle diese Optionen werden im Allgemeinen für den Zugriff auf Daten über C # verwendet. Sie werden häufig in gebäudedatenzentrierten Anwendungen angetroffen. Von vielen anderen Klassen, die nicht erwähnt werden und die dieselben Klassen `FooConnection` , `FooCommand` und `FooDataReader` `FooCommand` , kann erwartet werden, dass sie sich genauso verhalten.

Allgemeines Zugriffsmuster für ADO.NET-Verbindungen

Ein allgemeines Muster, das beim Zugriff auf Ihre Daten über eine ADO.NET-Verbindung verwendet werden kann, könnte wie folgt aussehen:

```
// 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
         }
    }
}
```

Oder wenn Sie nur ein einfaches Update durchführen und keinen Leser benötigen, würde dasselbe Grundkonzept gelten:

```
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();
     }
}
```

Anweisungen mit DataContexts verwenden

Viele ORMs wie Entity Framework machen Abstraktionsklassen verfügbar, die zur Interaktion mit darunterliegenden Datenbanken in Form von Klassen wie `DbContext` . Diese Kontexte implementieren im Allgemeinen auch die `IDisposable` Schnittstelle und sollten dies durch die `using` Anweisungen nutzen, wenn möglich:

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

Verwenden von Dispose Syntax zum Definieren des benutzerdefinierten Bereichs

In einigen Anwendungsfällen können Sie mithilfe der Syntax `using` einen benutzerdefinierten Bereich definieren. Sie können beispielsweise die folgende Klasse definieren, um Code in einer bestimmten Kultur auszuführen.

```
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;
    }
}
```

Mit dieser Klasse können Sie dann Codeblöcke definieren, die in einer bestimmten Kultur ausgeführt werden.

```
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
```

Hinweis: `CultureContext` wir die von uns erstellte `CultureContext` Instanz nicht verwenden, weisen wir ihr keine Variable zu.

Diese Technik wird vom `BeginForm` Helfer in ASP.NET-MVC verwendet.

Ausführen von Code im Kontext der Einschränkung

Wenn Sie Code (eine *Routine*) haben, die Sie in einem bestimmten Kontext (Constraint) ausführen möchten, können Sie Abhängigkeitseinspritzung verwenden.

Das folgende Beispiel zeigt die Einschränkung der Ausführung unter einer offenen SSL-Verbindung. Dieser erste Teil würde sich in Ihrer Bibliothek oder Ihrem Framework befinden, die Sie dem Client-Code nicht zugänglich machen.

```
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);
            }
        }
    }
}
```

Nun der Client-Code, der etwas unter SSL tun möchte, aber nicht alle SSL-Details verarbeiten möchte. Sie können jetzt innerhalb des SSL-Tunnels tun, was Sie möchten, beispielsweise einen symmetrischen Schlüssel austauschen:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

Sie führen diese Routine wie folgt aus:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

Dazu benötigen Sie die `using()` Klausel, da dies die einzige Möglichkeit ist (abgesehen von einem `try..finally` Block), dass der Client-Code (`ExchangeSymmetricKey`) niemals beendet wird, ohne die verfügbaren Ressourcen ordnungsgemäß zu entsorgen. Ohne die `using()` Klausel `using()` würden Sie niemals wissen, ob eine Routine die Einschränkung des Kontextes für die Entsorgung dieser Ressourcen durchbrechen könnte.

Anweisung verwenden online lesen: https://riptutorial.com/de/csharp/topic/38/anweisung-verwenden

Kapitel 9: Arrays

Syntax

- **Ein Array deklarieren:**

  <Typ> [] <Name>;

- **Zweidimensionales Array deklarieren:**

  <Typ> [,] <Name> = neuer <Typ> [<Wert>, <Wert>];

- **Deklarieren eines gezackten Arrays:**

  <Typ> [] <Name> = neuer <Typ> [<Wert>];

- **Unterfeld für ein gezacktes Array deklarieren:**

  <Name> [<Wert>] = neuer <Typ> [<Wert>];

- **Ein Array ohne Werte initialisieren:**

  <Name> = neuer <Typ> [<Länge>];

- **Ein Array mit Werten initialisieren:**

  <Name> = neuer <Typ> [] {<Wert>, <Wert>, <Wert>, ...};

- **Initialisieren eines zweidimensionalen Arrays mit Werten:**

  <Name> = neuer <Typ> [,] {{<Wert>, <Wert>}, {<Wert>, <Wert>}, ...};

- **Zugriff auf ein Element am Index i:**

  <name> [i]

- **Länge des Arrays erhalten:**

  <name> .Länge

Bemerkungen

In C # ist ein Array ein Referenztyp, *dh* es ist *nullfähig* .

Ein Array hat eine feste Länge, was bedeutet , man kann nicht `.Add()` , um es oder `.Remove()` von ihm. Um diese verwenden zu können, benötigen Sie ein dynamisches Array - `List` oder `ArrayList` .

Examples

Array-Kovarianz

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

Diese Konvertierung ist nicht typsicher. Der folgende Code löst eine Laufzeitausnahme aus:

```
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
```

Array-Werte abrufen und einstellen

```
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
```

Array deklarieren

Ein Array kann mithilfe der Initialisierungssyntax für eckige Klammern (`[]`) deklariert und mit dem Standardwert gefüllt werden. Beispiel: Erstellen eines Arrays von 10 Ganzzahlen:

```
int[] arr = new int[10];
```

Indizes in C # sind nullbasiert. Die Indizes des obigen Arrays sind 0-9. Zum Beispiel:

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Das heißt, das System beginnt den Elementindex ab 0 zu zählen. Zugriffe auf Elemente von Arrays erfolgen in **konstanter Zeit** . Das bedeutet, dass der Zugriff auf das erste Element des Arrays (zeitlich) die gleichen Kosten für den Zugriff auf das zweite Element, das dritte Element usw. verursacht.

Sie können auch einen bloßen Verweis auf ein Array angeben, ohne ein Array zu instantiieren.

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

Ein Array kann auch mit benutzerdefinierten Werten unter Verwendung der

Erfassungsinitialisierungssyntax erstellt und initialisiert werden:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

Der `new int[]` -Abschnitt kann bei der Deklaration einer Array-Variablen weggelassen werden. Dies ist kein in sich geschlossener *Ausdruck* , daher kann es nicht als Teil eines anderen Aufrufs verwendet werden (verwenden Sie dazu die Version mit `new`):

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

Implizit typisierte Arrays

Alternativ kann in Kombination mit dem Schlüsselwort `var` der bestimmte Typ weggelassen werden, so dass auf den Typ des Arrays geschlossen wird:

```
// 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 };
```

Über ein Array iterieren

```
int[] arr = new int[] {1, 6, 3, 3, 9};

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

foreach verwenden:

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

mit unsicherem Zugang mit Zeigern 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
```

```
        }
    }
}
```

Ausgabe:

1
6
3
3
9

Mehrdimensionale Arrays

Arrays können mehr als eine Dimension haben. Im folgenden Beispiel wird ein zweidimensionales Array mit zehn Zeilen und zehn Spalten erstellt:

```
int[,] arr = new int[10, 10];
```

Ein Array von drei Dimensionen:

```
int[,,] arr = new int[10, 10, 10];
```

Sie können das Array auch bei der Deklaration initialisieren:

```
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
```

Gezackte Arrays

Gezackte Arrays sind Arrays, die anstelle von primitiven Typen Arrays (oder andere Sammlungen) enthalten. Es ist wie ein Array von Arrays - jedes Array-Element enthält ein anderes Array.

Sie ähneln mehrdimensionalen Arrays, unterscheiden sich jedoch geringfügig - da multidimensionale Arrays auf eine feste Anzahl von Zeilen und Spalten beschränkt sind und bei gezackten Arrays kann jede Zeile eine andere Anzahl von Spalten haben.

Deklarieren eines gezackten Arrays

Beispiel: Deklarieren eines gezackten Arrays mit 8 Spalten:

```
int[][] a = new int[8][];
```

Das zweite `[]` wird ohne Nummer initialisiert. Um die Sub-Arrays zu initialisieren, müssen Sie dies separat tun:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Werte einstellen / einstellen

Nun ist es ganz einfach, eines der Subarrays zu bekommen. Lassen Sie uns alle Zahlen der 3. Spalte von `a` ausdrucken:

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Einen bestimmten Wert erhalten:

```
a[<row_number>][<column_number>]
```

Einen bestimmten Wert einstellen:

```
a[<row_number>][<column_number>] = <value>
```

Denken Sie **daran** : Es wird immer empfohlen, gezackte Arrays (Arrays von Arrays) anstelle von mehrdimensionalen Arrays (Matrizen) zu verwenden. Die Verwendung ist schneller und sicherer.

Beachten Sie die Reihenfolge der Klammern

Betrachten Sie ein dreidimensionales Array von fünfdimensionalen Arrays von eindimensionalen Arrays von `int` . Dies ist in C # geschrieben als:

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

Im CLR-Typensystem ist die Konvention für die Anordnung der Klammern umgekehrt, so dass wir mit der obigen `arr` Instanz Folgendes haben:

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

und ebenfalls:

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

Prüfen, ob ein Array ein anderes Array enthält

```
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;
    }
}
```

/// Probe

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

Initialisieren eines Arrays, das mit einem wiederholten, nicht standardmäßigen Wert gefüllt ist

Wie wir wissen, können wir ein Array mit Standardwerten deklarieren:

```
int[] arr = new int[10];
```

Dadurch wird ein Array mit 10 Ganzzahlen erstellt, wobei jedes Element des Arrays den Wert `0` (Standardwert des Typs `int`) hat.

Um ein Array zu erstellen, das mit einem nicht standardmäßigen Wert initialisiert wurde, können Sie `Enumerable.Repeat` aus dem `System.Linq` Namespace verwenden:

1. So erstellen Sie ein `bool` Array der Größe 10, das mit **"true"** gefüllt ist

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. So erstellen Sie ein `int` Array der Größe 5 mit **"100"**

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. So erstellen Sie ein `string` Array der Größe 5, das mit **"C #"** gefüllt ist

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Arrays kopieren

Kopieren eines `Array.Copy()` mit der statischen `Array.Copy()` Methode, beginnend bei Index 0 in Quelle und Ziel:

```
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
```

Kopieren des gesamten Arrays mit der `CopyTo()` Instanzmethode, beginnend mit Index 0 der Quelle und dem angegebenen Index im Ziel:

```
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` wird verwendet, um eine Kopie eines Array-Objekts zu erstellen.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

Sowohl `CopyTo` als auch `Clone` führen eine flache Kopie aus. `CopyTo` bedeutet, dass der Inhalt Verweise auf dasselbe Objekt wie die Elemente im ursprünglichen Array enthält.

Erstellen eines Arrays von fortlaufenden Nummern

LINQ stellt eine Methode bereit, mit der sich eine Sammlung mit fortlaufenden Nummern auf einfache Weise erstellen lässt. Sie können beispielsweise ein Array deklarieren, das die ganzen Zahlen zwischen 1 und 100 enthält.

Mit der `Enumerable.Range` Methode können Sie eine Sequenz von Ganzzahlen aus einer

angegebenen Startposition und einer Anzahl von Elementen erstellen.

Die Methode akzeptiert zwei Argumente: den Startwert und die Anzahl der zu generierenden Elemente.

```
Enumerable.Range(int start, int count)
```

Beachten Sie, dass die `count` nicht negativ sein kann.

Verwendungszweck:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

Dadurch wird ein Array mit den Zahlen 1 bis 100 (`[1, 2, 3, ..., 98, 99, 100]`) erstellt.

Da die `Range` Methode ein `IEnumerable<int>` , können wir andere LINQ-Methoden verwenden:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

Dadurch wird ein Array mit 10 ganzzahligen Quadraten beginnend mit `4` generiert: `[4, 9, 16, ..., 100, 121]` .

Vergleich von Arrays auf Gleichheit

LINQ bietet eine integrierte Funktion zum Überprüfen der Gleichheit von zwei `IEnumerable` Funktion kann in Arrays verwendet werden.

Die `SequenceEqual` Funktion gibt `true` wenn die Arrays dieselbe Länge haben und die Werte in den entsprechenden Indizes gleich sind, andernfalls `false` .

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

Dies wird drucken:

```
Arrays equal? True
```

Arrays als IEnumerable <> -Instanzen

Alle Arrays implementieren die nicht generische `IList` Schnittstelle (und damit auch nicht generische `ICollection` und `IEnumerable` Basisschnittstellen).

Wichtiger ist, dass eindimensionale Arrays die generischen Schnittstellen `IList<>` und `IReadOnlyList<>` (und ihre Basisschnittstellen) für den darin enthaltenen Datentyp implementieren. Dies bedeutet, dass sie als generische aufzählbare Typen behandelt und an eine Vielzahl von

Methoden übergeben werden können, ohne dass sie zuerst in eine Nicht-Array-Form konvertiert werden müssen.

```
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.
```

Nach dem Ausführen dieses Codes enthält die Liste `listOfIntegers` eine `List<int>` mit den Werten 3, 5 und 7.

Die Unterstützung für `IEnumerable<>` bedeutet, dass Arrays mit LINQ abgefragt werden können, zum Beispiel `arr1.Select(i => 10 * i)` .

Arrays online lesen: https://riptutorial.com/de/csharp/topic/1429/arrays

Kapitel 10: ASP.NET-Identität

Einführung

Tutorials zu asp.net Identität wie Benutzerverwaltung, Rollenverwaltung, Erstellen von Token und mehr.

Examples

Implementieren eines Kennwort-Reset-Tokens in asp.net identity mit dem Benutzermanager

1. Erstellen Sie einen neuen Ordner mit dem Namen MyClasses, erstellen Sie die folgende Klasse und fügen Sie sie hinzu

```
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. Konfigurieren Sie Ihre Identitätsklasse

```
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. Fügen Sie Ihre Anmeldeinformationen zur web.config hinzu. Ich habe gmail nicht in diesem Abschnitt verwendet, da die Verwendung von gmail an meinem Arbeitsplatz blockiert ist und

immer noch einwandfrei funktioniert.

```
<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. Nehmen Sie die erforderlichen Änderungen an Ihrem Account Controller vor. Fügen Sie den folgenden hervorgehobenen Code hinzu.

```
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);
}
```

Kompilieren, dann ausführen. Prost!

ASP.NET-Identität online lesen: https://riptutorial.com/de/csharp/topic/9577/asp-net-identitat

Kapitel 11: AssemblyInfo.cs Beispiele

Bemerkungen

Der Dateiname `AssemblyInfo.cs` wird von der Konvention als Quelldatei verwendet, in der Entwickler Metadatenattribute platzieren, die die gesamte Assembly beschreiben, die sie erstellen.

Examples

[AssemblyTitle]

Dieses Attribut wird verwendet, um dieser bestimmten Assembly einen Namen zu geben.

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

[Montageprodukt]

Dieses Attribut wird verwendet, um das Produkt zu beschreiben, für das diese bestimmte Baugruppe bestimmt ist. Mehrere Baugruppen können Komponenten desselben Produkts sein. In diesem Fall können sie alle denselben Wert für dieses Attribut haben.

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

Globale und lokale AssemblyInfo

Da ein globaler Code eine bessere DRYness ermöglicht, müssen Sie für Projekte, die Abweichungen aufweisen, nur andere Werte in AssemblyInfo.cs eingeben. Diese Verwendung setzt voraus, dass Ihr Produkt mehr als ein visuelles Studio-Projekt hat.

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 - eine für jedes Projekt

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

Sie können GlobalAssemblyInfo.cs mit dem folgenden Verfahren zum lokalen Projekt hinzufügen:

1. Wählen Sie im Kontextmenü des Projekts Hinzufügen / Vorhandenes Element
2. Wählen Sie GlobalAssemblyInfo.cs aus
3. Erweitern Sie den Add-Button, indem Sie auf den kleinen Abwärtspfeil auf der rechten Seite klicken
4. Wählen Sie "Als Link hinzufügen" in der Dropdown-Liste der Schaltflächen aus

[AssemblyVersion]

Dieses Attribut wendet eine Version auf die Assembly an.

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

Das Zeichen `*` wird verwendet, um einen Teil der Version bei jedem Kompilieren automatisch zu inkrementieren (häufig für die Build-Nummer)

Baugruppenattribute lesen

Mit den umfangreichen Reflection-APIs von .NET können Sie auf die Metadaten einer Assembly zugreifen. Sie können beispielsweise das Titelattribut `this` Assembly mit dem folgenden Code abrufen

```
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}");
```

Automatisierte Versionierung

Ihr Code in der Quellcodeverwaltung enthält Versionsnummern entweder standardmäßig (SVN-

IDs oder Git-SHA1-Hashes) oder explizit (Git-Tags). Anstatt Versionen in AssemblyInfo.cs manuell zu aktualisieren, können Sie einen Build-Time-Prozess verwenden, um die Version von Ihrem Quellcodeverwaltungssystem in Ihre AssemblyInfo.cs-Dateien und somit in Ihre Assemblys zu schreiben.

Die GitVersionTask- oder SemVer.Git.Fody- NuGet-Pakete sind Beispiele dafür. Um beispielsweise GitVersionTask zu verwenden, entfernen Sie nach der Installation des Pakets in Ihrem Projekt die Attribute `Assembly*Version` aus den Dateien AssemblyInfo.cs. Dadurch wird GitVersionTask für die Versionierung Ihrer Baugruppen verantwortlich.

Beachten Sie, dass Semantic Versioning zunehmend der *De-facto*- Standard ist. Daher empfehlen diese Methoden die Verwendung von Quellcode-Tags, die SemVer folgen.

Gemeinsame Felder

Es ist empfehlenswert, die Standardfelder Ihrer AssemblyInfo auszufüllen. Die Informationen können von den Installationsprogrammen abgeholt werden und erscheinen dann, wenn Sie Programme und Funktionen (Windows 10) zum Deinstallieren oder Ändern eines Programms verwenden.

Das Minimum sollte sein:

- AssemblyTitle - normalerweise der Namespace, z. *B.* MyCompany.MySolution.MyProject
- AssemblyCompany - der vollständige Name der juristischen Personen
- AssemblyProduct - Marketing kann hier einen Blick haben
- AssemblyCopyright - halten Sie es auf dem neuesten Stand, da es sonst schlecht aussieht

"AssemblyTitle" wird zur "Dateibeschreibung", wenn Sie die Registerkarte "Eigenschaften" der DLL untersuchen.

[AssemblyConfiguration]

AssemblyConfiguration: Das AssemblyConfiguration-Attribut muss über die Konfiguration verfügen, mit der die Assembly erstellt wurde. Verwenden Sie die bedingte Kompilierung, um verschiedene Assembly-Konfigurationen ordnungsgemäß aufzunehmen. Verwenden Sie den Block ähnlich dem folgenden Beispiel. Fügen Sie beliebig viele verschiedene Konfigurationen hinzu.

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

[InternalsVisibleTo]

Wenn Sie `internal` Klassen oder Funktionen einer Assembly von einer anderen Assembly aus zugänglich machen möchten, geben Sie dies mit `InternalsVisibleTo` und dem Assemblynamen an, auf den zugegriffen werden darf.

In diesem Beispielcode in der Assembly darf `MyAssembly.UnitTests` `internal` Elemente von `MyAssembly` .

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

Dies ist besonders nützlich für Komponententests, um unnötige `public` Deklarationen zu verhindern.

[AssemblyKeyFile]

Wann immer wir möchten, dass unsere Baugruppe in GAC installiert wird, muss sie einen starken Namen haben. Für eine starke Namenserstellung müssen wir einen öffentlichen Schlüssel erstellen. So generieren Sie die `.snk` Datei

So erstellen Sie eine Schlüsseldatei mit starkem Namen

1. Entwickler-Eingabeaufforderung für VS2015 (mit Administratorzugriff)
2. Geben Sie an der Eingabeaufforderung cd C: \ Directory_Name ein, und drücken Sie die EINGABETASTE.
3. Geben Sie an der Eingabeaufforderung sn -k KeyFileName.snk ein, und drücken Sie die EINGABETASTE.

Sobald die keyFileName.snk im angegebenen Verzeichnis erstellt wurde, geben Sie in Ihrem Projekt eine Referenz an. Geben Sie dem Attribut `AssemblyKeyFileAttribute` den Pfad zur `snk` Datei an, um den Schlüssel zu generieren, wenn Sie unsere Klassenbibliothek erstellen.

> Eigenschaften -> AssemblyInfo.cs

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

Nach dem Build wird eine starke Namensassemblierung erstellt. Nachdem Sie Ihre starke Namensassembly erstellt haben, können Sie sie in GAC installieren

Viel Spaß beim Codieren :)

AssemblyInfo.cs Beispiele online lesen: https://riptutorial.com/de/csharp/topic/4264/assemblyinfo-cs-beispiele

Kapitel 12: Async / await, Backgroundworker, Task- und Thread-Beispiele

Bemerkungen

Um eines dieser Beispiele auszuführen, nennen Sie es einfach so:

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

Examples

ASP.NET Konfigurieren Sie Await

Wenn ASP.NET eine Anforderung verarbeitet, wird ein Thread aus dem Threadpool zugewiesen und ein **Anforderungskontext** erstellt. Der Anforderungskontext enthält Informationen zur aktuellen Anforderung, auf die über die statische Eigenschaft `HttpContext.Current` zugegriffen werden kann. Der Anforderungskontext für die Anforderung wird dann dem Thread zugewiesen, der die Anforderung bearbeitet.

Ein gegebener Anforderungskontext **kann jeweils nur für einen Thread aktiv sein** .

Wenn die Ausführung `await` erreicht, wird der Thread, der eine Anforderung verarbeitet, an den Threadpool zurückgegeben, während die asynchrone Methode ausgeführt wird und der Anforderungskontext für einen anderen Thread frei ist.

```
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);
}
```

Wenn die Aufgabe abgeschlossen ist, weist der Thread-Pool einen anderen Thread zu, um die Ausführung der Anforderung fortzusetzen. Der Anforderungskontext wird dann diesem Thread zugewiesen. Dies kann der ursprüngliche Thread sein oder nicht.

Blockierung

Wenn das Ergebnis eines `async` Methodenaufrufs auf **synchron** gewartet wird, können Deadlocks

entstehen. Der folgende Code führt beispielsweise zu einem Deadlock, wenn `IndexSync()` aufgerufen wird:

```
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;
}
```

Dies liegt daran, dass standardmäßig die erwartete Task in diesem Fall `db.Products.ToListAsync()` den Kontext erfasst (im Fall von ASP.NET den Anforderungskontext) und versucht, ihn zu verwenden, sobald er abgeschlossen ist.

Wenn der gesamte Call - Stack ist asynchron ist es kein Problem , denn sobald `await` den ursprünglichen Thread erreicht Mitteilung ist, den Anforderungskontext zu befreien.

Wenn wir synchron mit `Task.Result` oder `Task.Wait()` (oder anderen Blockierungsmethoden) blockieren, ist der ursprüngliche Thread noch aktiv und behält den Anforderungskontext bei. Die erwartete Methode arbeitet weiterhin asynchron und sobald der Callback versucht, auszuführen, dh nachdem die erwartete Task zurückgegeben wurde, versucht sie, den Anforderungskontext abzurufen.

Daher kommt es zu einem Deadlock, da der blockierende Thread mit dem Anforderungskontext darauf wartet, dass die asynchrone Operation abgeschlossen ist, während die asynchrone Operation versucht, den Anforderungskontext abzurufen, um abzuschließen.

ConfigureAwait

Aufrufe einer erwarteten Aufgabe erfassen standardmäßig den aktuellen Kontext und versuchen, die Ausführung im Kontext fortzusetzen, sobald sie abgeschlossen sind.

Durch die Verwendung von `ConfigureAwait(false)` kann dies verhindert und Deadlocks vermieden werden.

```
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;
}
```

Dadurch können Deadlocks vermieden werden, wenn asynchroner Code gesperrt werden muss. Dies geht jedoch mit dem Verlust des Kontexts in der Fortsetzung (Code nach dem Aufruf von wait) einher.

In ASP.NET bedeutet dies, dass, wenn Ihr Code einem Aufruf folgt, `await someTask.ConfigureAwait(false);` zu `await someTask.ConfigureAwait(false);` versucht, auf Informationen aus dem Kontext zuzugreifen, z. B. `HttpContext.Current.User` dann sind die Informationen verloren gegangen. In diesem Fall ist `HttpContext.Current` . Zum Beispiel:

```
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();
}
```

Wenn `ConfigureAwait(true)` verwendet wird (äquivalent dazu, überhaupt kein ConfigureAwait zu haben), werden sowohl `user` als auch `user` mit denselben Daten `user2` .

Aus diesem Grund wird häufig empfohlen, `ConfigureAwait(false)` in Bibliothekscode zu verwenden, in dem der Kontext nicht mehr verwendet wird.

Asynchron / warten

Im Folgenden finden Sie ein einfaches Beispiel für die Verwendung von async / await, um einige zeitintensive Aufgaben in einem Hintergrundprozess auszuführen, während die Option beibehalten wird, andere Aufgaben auszuführen, die nicht auf die zeitintensiven Aufgaben warten müssen.

Wenn Sie jedoch später mit dem Ergebnis der zeitintensiven Methode arbeiten müssen, können Sie dies durch Abwarten der Ausführung tun.

```
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

Nachfolgend finden Sie ein einfaches Beispiel für die Verwendung eines `BackgroundWorker` Objekts zum Ausführen zeitintensiver Vorgänge in einem Hintergrundthread.

Du musst:

1. Definieren Sie eine Arbeitermethode, die die zeitintensive Arbeit erledigt, und rufen Sie sie von einem Ereignishandler für das `DoWork` Ereignis eines `BackgroundWorker` .
2. Starten Sie die Ausführung mit `RunWorkerAsync` . Jedes Argument , die der Arbeitnehmer Verfahren erforderlich an `DoWork` kann über die übergeben werden `DoWorkEventArgs` Parameter `RunWorkerAsync` .

Zusätzlich zum `DoWork` Ereignis definiert die `BackgroundWorker` Klasse zwei Ereignisse, die für die Interaktion mit der Benutzeroberfläche verwendet werden sollen. Diese sind optional.

- Das Ereignis `RunWorkerCompleted` wird ausgelöst, wenn die `DoWork` Handler abgeschlossen sind.
- Das `ProgressChanged` Ereignis wird ausgelöst, wenn die `ReportProgress` Methode aufgerufen wird.

```
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);
}
```

Aufgabe

Nachfolgend finden Sie ein einfaches Beispiel für die Verwendung einer `Task` , um einige Zeit in einem Hintergrundprozess zu erledigen.

Sie müssen nur Ihre zeitintensive Methode in einen `Task.Run()` Aufruf `Task.Run()` .

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

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

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

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

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

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

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

Faden

Im Folgenden finden Sie ein einfaches Beispiel für die Verwendung eines `Thread` , um einige Zeit in einem Hintergrundprozess zu erledigen.

```
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.");
}
```

Wie Sie sehen, können wir keinen Wert aus unserer `TimeIntensiveMethod` da `Thread` eine void-Methode als Parameter erwartet.

Um einen Rückgabewert von einem `Thread` verwenden Sie entweder ein Ereignis oder das folgende:

```
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);
```

Task "run and forget" Erweiterung

In bestimmten Fällen (z. B. Protokollierung) kann es nützlich sein, die Task auszuführen und nicht auf das Ergebnis zu warten. Mit der folgenden Erweiterung können Sie task ausführen und die Ausführung des Restcodes fortsetzen:

```
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);
        }
    }
}
```

Das Ergebnis wird nur innerhalb der Erweiterungsmethode erwartet. Da `async` / `await` verwendet wird, ist es möglich, eine Ausnahme `async` und eine optionale Methode für die Behandlung aufzurufen.

Ein Beispiel zur Verwendung der Erweiterung:

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

Async / await, Backgroundworker, Task- und Thread-Beispiele online lesen: https://riptutorial.com/de/csharp/topic/3824/async---await--backgroundworker--task--und-thread-beispiele

Kapitel 13: Async-Await

Einführung

In C # blockiert eine als `async` deklarierte Methode nicht innerhalb eines synchronen Prozesses, wenn Sie E / A-basierte Vorgänge verwenden (z. B. Webzugriff, Arbeiten mit Dateien, ...). Das Ergebnis solcher async-markierter Methoden kann über die Verwendung des `await` Schlüsselworts `await` .

Bemerkungen

Eine `async` Methode kann `void` `Task` oder `Task<T>` .

Der Rückgabetyp `Task` wartet, bis die Methode abgeschlossen ist, und das Ergebnis wird `void` . `Task<T>` gibt einen Wert vom Typ `T` nachdem die Methode abgeschlossen ist.

`async` Methoden sollten `Task` oder `Task<T>` im Gegensatz zu `void` in fast allen Fällen zurückgeben. `async void` methoden können nicht `await` , was zu einer vielzahl von problemen führt. Das einzige Szenario, in dem ein `async void` sollte `void` ist im Fall eines Event-Handlers.

`async` / `await` funktioniert, indem Sie Ihre `async` Methode in eine Zustandsmaschine umwandeln. Dazu erstellt es hinter den Kulissen eine Struktur, in der der aktuelle Status und beliebige Kontexte (wie lokale Variablen) `MoveNext()` , und eine `MoveNext()` Methode `MoveNext()` wird, um den `MoveNext()` (und den zugehörigen Code auszuführen), wenn ein erwartetes Erwarten beendet wird.

Examples

Einfache aufeinanderfolgende Anrufe

```
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);
}
```

Hierbei ist zu beachten, dass jede `await` Methode asynchron aufgerufen wird und die Steuerung für den Zeitpunkt des Aufrufs an das System zurückgegeben wird. Der Fluss in der Methode ist jedoch linear und erfordert keine besondere Behandlung Asynchronität. Wenn eine der aufgerufenen Methoden fehlschlägt, wird die Ausnahme wie erwartet verarbeitet. Dies bedeutet, dass die Ausführung der Methode abgebrochen wird und die Ausnahme in den Stack aufgenommen wird.

Versuchen Sie / Catch / Schließlich

6,0

Ab C # 6.0 kann das `await` Schlüsselwort jetzt innerhalb eines `catch` und `finally` Blocks verwendet werden.

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

5,0 6,0

Vor C # 6.0 müssten Sie etwas wie folgt tun. Beachten Sie, dass 6.0 die Nullprüfungen auch mit dem Operator für die Nullpropagierung bereinigt hat.

```
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;
}
```

Wenn Sie auf eine Aufgabe warten, die nicht von `async` erstellt wurde (z. B. eine von `Task.Run` erstellte `Task.Run`), können einige Debugger an Ausnahmen brechen, die von der Task ausgelöst werden, selbst wenn sie scheinbar vom umgebenden try / catch behandelt werden. Dies geschieht, weil der Debugger es in Bezug auf Benutzercode als nicht behandelt betrachtet. In Visual Studio gibt es eine Option namens "Just My Code" , die deaktiviert werden kann, damit der Debugger in solchen Situationen nicht beschädigt wird.

Web.config-Setup für das korrekte async-Verhalten auf 4,5 setzen.

Die web.config system.web.httpRuntime muss das Ziel 4.5 sein, um sicherzustellen, dass der Thread den Anforderungskontext durchläuft, bevor die async-Methode fortgesetzt wird.

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

Async und wait haben in ASP.NET vor 4.5 ein undefiniertes Verhalten. Async / await wird in einem beliebigen Thread fortgesetzt, der möglicherweise keinen Anforderungskontext hat. Anwendungen unter Last schlagen nach dem Zufallsprinzip fehl, mit Ausnahme von NULL-Referenzzugriffen, die

nach dem Erwarten auf HttpContext zugreifen. Die Verwendung von HttpContext.Current in WebApi ist aufgrund von Async gefährlich

Gleichzeitige Anrufe

Es ist möglich, auf mehrere Anrufe gleichzeitig zu warten, indem zuerst die wartbaren Aufgaben aufgerufen und *dann* auf sie gewartet werden.

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

    await firstTask;
    await secondTask;
}
```

Alternativ kann `Task.WhenAll` verwendet werden, um mehrere Tasks in einem einzigen `Task` zu gruppieren, der abgeschlossen ist, wenn alle übergebenen Tasks abgeschlossen sind.

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

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

Sie können dies auch innerhalb einer Schleife tun, zum Beispiel:

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

await Task.WhenAll(tasks);
```

Um Ergebnisse von einer Aufgabe zu erhalten, nachdem mehrere Aufgaben mit Task.WhenAll abgewartet wurden, warten Sie einfach erneut auf die Aufgabe. Da die Aufgabe bereits abgeschlossen ist, wird das Ergebnis einfach zurückgegeben

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

await Task.WhenAll(task1, task2);

var result = await task2;
```

Mit `Task.WhenAny` können auch mehrere Tasks parallel ausgeführt werden, z. B. `Task.WhenAll` , mit dem Unterschied, dass diese Methode abgeschlossen wird, wenn *eine* der angegebenen Tasks ausgeführt wird.

```
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);
}
```

Die `Task` zurückgegeben durch `RunConcurrentTasksWhenAny` wird abgeschlossen , wenn ein `firstTask` , `secondTask` oder `thirdTask` abgeschlossen ist .

Warten Sie auf den Operator und das async-Schlüsselwort

`await` Operator und das `async` Schlüsselwort zusammenkommen:

> Die asynchrone Methode, in der **await verwendet** wird, muss durch das Schlüsselwort **async** geändert werden.

Das Gegenteil trifft nicht immer zu: Sie können eine Methode als `async` kennzeichnen, ohne in ihrem Körper zu `await` .

Was `await` wird, ist die Ausführung des Codes bis zum Abschluss der erwarteten Task auszusetzen. auf jede aufgabe kann gewartet werden.

Hinweis: Sie können nicht auf eine asynchrone Methode warten, die nichts zurückgibt (nichtig).

Tatsächlich ist das Wort 'suspends' etwas irreführend, da nicht nur die Ausführung angehalten wird, sondern der Thread möglicherweise für die Ausführung anderer Operationen frei wird. Das `await` wird unter der Haube durch ein wenig Compiler-Zauber umgesetzt: Es teilt eine Methode in zwei Teile - vor und nach dem `await` . Der letzte Teil wird ausgeführt, wenn die erwartete Aufgabe abgeschlossen ist.

Wenn wir einige wichtige Details ignorieren, erledigt der Compiler dies grob für Sie:

```
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;
}
```

wird:

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

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

Jede übliche Methode kann auf folgende Weise in asynchron umgewandelt werden:

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

Dies kann vorteilhaft sein, wenn Sie eine lange ausgeführte Methode im UI-Thread ausführen müssen, ohne die UI einzufrieren.

Es gibt jedoch eine sehr wichtige Bemerkung: **Asynchron bedeutet nicht immer gleichzeitig (parallel oder sogar multithreadig).** `async` - `await` erlaubt selbst in einem einzigen Thread immer noch asynchronen Code. Sehen Sie sich beispielsweise diesen benutzerdefinierten Taskplaner an . Ein solcher "verrückter" Task-Scheduler kann einfach Aufgaben in Funktionen umwandeln, die innerhalb der Message-Loop-Verarbeitung aufgerufen werden.

Wir müssen uns fragen: Welcher Thread führt die Fortsetzung unserer Methode `DoIt_Continuation` ?

Der `await` Operator plant standardmäßig die Ausführung der Fortsetzung mit dem aktuellen Synchronisierungskontext . Dies bedeutet, dass standardmäßig für WinForms und WPF die Fortsetzung im UI-Thread ausgeführt wird. Wenn Sie dieses Verhalten aus irgendeinem Grund ändern müssen, verwenden Sie die Methode `Task.ConfigureAwait()` :

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

Eine Aufgabe zurückgeben, ohne abzuwarten

Methoden, die asynchrone Operationen ausführen, brauchen nicht zu `await` wenn:

- Es gibt nur einen asynchronen Aufruf innerhalb der Methode
- Der asynchrone Aufruf ist am Ende der Methode
- Es ist nicht notwendig, Ausnahmen zu erfassen / zu behandeln, die innerhalb der Task auftreten können

Betrachten Sie diese Methode, die eine `Task` zurückgibt:

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

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

Wenn `GetByKeyAsync` dieselbe Signatur wie `GetUserAsync` (Rückgabe einer `Task<User>`), kann die Methode vereinfacht werden:

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

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

In diesem Fall ist das Verfahren nicht markiert werden müssen , `async` , obwohl es einen asynchronen Vorgang ist Vorformen. Die Aufgabe von zurück `GetByKeyAsync` wird direkt an die rufenden Methode übergeben, wo es wird `await` hrsg.

Wichtig : Wenn Sie die `Task` anstatt auf sie zu warten, wird das Ausnahmeverhalten der Methode geändert, da die Ausnahme nicht innerhalb der Methode ausgelöst wird, die die Task startet, sondern in der Methode, die sie erwartet.

```
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();
```

Dies verbessert die Leistung, da der Compiler die Erzeugung einer zusätzlichen **asynchronen** Zustandsmaschine **erspart** .

Das Blockieren von asynchronem Code kann zu Deadlocks führen

Es ist eine schlechte Praxis, asynchrone Aufrufe zu blockieren, da dies in Umgebungen mit Synchronisationskontext Deadlocks verursachen kann. Die beste Vorgehensweise ist, async / await "ganz nach unten" zu verwenden. Beispielsweise verursacht der folgende Windows Forms-Code einen Deadlock:

```
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");
}
```

Nach Abschluss des asynchronen Aufrufs wartet er im Wesentlichen darauf, dass der Synchronisationskontext verfügbar wird. Der Event-Handler "behält" jedoch den Synchronisationskontext bei, während er auf den `TryThis()` der `TryThis()` Methode wartet, was zu einem `TryThis()` Warten führt.

Um dies zu beheben, sollte der Code in geändert werden

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

Hinweis: Event-Handler sind der einzige Ort, an dem `async void` verwendet werden sollte (da Sie keine `async void` Methode erwarten können).

Async / await verbessert die Leistung nur, wenn die Maschine zusätzliche Arbeit ausführen kann

Betrachten Sie den folgenden Code:

```
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);
}
```

Dies wird nicht besser sein als

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

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

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

Der Hauptzweck von async / await besteht darin, der Maschine zusätzliche Arbeit zu ermöglichen, z. B. um dem aufrufenden Thread andere Arbeit zu ermöglichen, während er auf ein Ergebnis einer E / A-Operation wartet. In diesem Fall darf der aufrufende Thread niemals mehr Arbeit erledigen, als es sonst möglich gewesen wäre. Es gibt also keinen Leistungsgewinn, `MethodA()` Sie einfach `MethodA()` , `MethodB()` und `MethodC()` synchron `MethodC()` .

Async-Await online lesen: https://riptutorial.com/de/csharp/topic/48/async-await

Kapitel 14: Asynchroner Sockel

Einführung

Durch die Verwendung von asynchronen Sockets kann ein Server auf eingehende Verbindungen warten und in der Zwischenzeit eine andere Logik ausführen, im Gegensatz zu synchronem Socket, wenn er abhört. Er blockiert den Haupt-Thread und die Anwendung reagiert nicht mehr und das Einfrieren wird erst nach dem Verbindungsaufbau eines Clients eingestellt.

Bemerkungen

Socket und Netzwerk

Zugriff auf einen Server außerhalb meines eigenen Netzwerks Dies ist eine häufige Frage, und wenn sie gefragt wird, wird sie meistens als Thema gekennzeichnet.

Serverseite

Im Netzwerk Ihres Servers müssen Sie den Router portieren, um ihn an Ihren Server weiterzuleiten.

Zum Beispiel PC, auf dem der Server läuft:

lokale IP = `192.168.1.115`

Der Server hört auf Port 1234.

Leiten Sie eingehende Verbindungen am `Port 1234` Router an `192.168.1.115`

Client-Seite

Das einzige, was Sie ändern müssen, ist die IP. Sie möchten sich nicht mit Ihrer Loopback-Adresse verbinden, sondern mit der öffentlichen IP-Adresse des Netzwerks, auf dem Ihr Server läuft. Diese IP erhalten Sie hier .

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

Nun erstellen Sie eine Anforderung für diesen Endpunkt: `10.10.10.10:1234` Wenn Sie den Router durch den Eigenschafts-Port- `10.10.10.10:1234` wird die Verbindung zwischen Server und Client problemlos hergestellt.

Wenn Sie eine Verbindung zu einer lokalen IP-Adresse herstellen möchten, müssen Sie die Portnummer nicht auf `192.168.1.178` oder ähnliches ändern.

Daten senden:

Daten werden in Byte-Array gesendet. Sie müssen Ihre Daten in ein Byte-Array packen und auf

der anderen Seite entpacken.

Wenn Sie mit Socket vertraut sind, können Sie auch versuchen, Ihr Byte-Array vor dem Senden zu verschlüsseln. Dadurch wird verhindert, dass jemand Ihr Paket stiehlt.

Examples

Beispiel für asynchrones Socket (Client / Server).

Server Side Beispiel

Listener für Server erstellen

Beginnen Sie mit dem Erstellen eines Servers, der Clients behandelt, die eine Verbindung herstellen, und Anforderungen, die gesendet werden. Erstellen Sie also eine Listener-Klasse, die dies erledigt.

```
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);
    }
 }
```

Zuerst müssen wir den Listener-Socket initialisieren, um Verbindungen zu überwachen. Wir werden einen TCP-Sockel verwenden, deshalb verwenden wir SocketType.Stream. Außerdem geben wir an, welchen Port der Server abhören soll

Dann hören wir auf eingehende Verbindungen.

Die Baummethoden, die wir hier verwenden, sind:

1. ListenerSocket.Bind ();

   Diese Methode bindet den Socket an einen IPEndPoint . Diese Klasse enthält die Host- und lokalen oder Remote-Port-Informationen, die eine Anwendung benötigt, um eine Verbindung zu einem Dienst auf einem Host herzustellen.

2. ListenerSocket.Listen (10);

   Der Parameter backlog gibt die Anzahl der eingehenden Verbindungen an, die zur Annahme in die Warteschlange gestellt werden können.

3. ListenerSocket.BeginAccept ();

Der Server wartet auf eingehende Verbindungen und arbeitet mit einer anderen Logik. Wenn eine Verbindung besteht, wechselt der Server wieder zu dieser Methode und führt die AcceptCallBack-Methode aus

```
    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);
        }
    }
```

Wenn sich ein Client verbindet, können wir sie mit dieser Methode akzeptieren:

Drei Methoden, die wir hier verwenden, sind:

1. ListenerSocket.EndAccept ()

   Wir haben den Rückruf mit `Listener.BeginAccept()` beendet. Jetzt müssen wir den Rückruf beenden. `The EndAccept()` -Methode akzeptiert einen IAsyncResult-Parameter. Dies speichert den Status der asynchronen Methode. Aus diesem Status können wir den Socket extrahieren, von dem die eingehende Verbindung stammt.

2. `ClientController.AddClient()`

   Mit dem Socket, den wir von `EndAccept()` haben, erstellen wir einen Client mit einer eigenen Methode *(Code ClientController unter dem Server-Beispiel)* .

3. ListenerSocket.BeginAccept ()

   Wir müssen wieder zuhören, wenn der Socket mit der neuen Verbindung fertig ist. Übergeben Sie die Methode, die diesen Rückruf abfangen soll. Und übergeben Sie auch den Listener-Socket, damit wir diesen Socket für anstehende Verbindungen wiederverwenden können.

```
    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);
        }
    }
```

Jetzt haben wir ein Listening Socket, aber wie erhalten wir vom Client gesendete Daten. Dies ist, was der nächste Code anzeigt.

Erstellen Sie einen Serverempfänger für jeden Client

Erstellen Sie zunächst eine Empfangsklasse mit einem Konstruktor, der einen Socket als Parameter verwendet:

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

In der nächsten Methode beginnen wir zunächst damit, dem Puffer eine Größe von 4 Bytes (Int32) zu geben, oder das Paket enthält die Teile {Länge, tatsächliche Daten}. Die ersten 4 Bytes reservieren wir also für die Länge der Daten, den Rest für die eigentlichen Daten.

Als nächstes verwenden wir die BeginReceive () -Methode. Diese Methode wird verwendet, um den Empfang von verbundenen Clients zu starten. Wenn Daten empfangen werden, wird die `ReceiveCallback` Funktion ausgeführt.

```
    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()
    }
```

Wir haben also einen Server eingerichtet, der eingehende Verbindungen empfangen und überwachen kann. Wenn ein Client eine Verbindung herstellt, wird er zu einer Liste von Clients hinzugefügt, und jeder Client verfügt über eine eigene Empfangsklasse. Um den Server anzuhören:

```
Listener listener = new Listener();
listener.StartListening();
```

Einige Klassen, die ich in diesem Beispiel verwende

```
    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));
          }
      }
```

Client Side Beispiel

Verbindung zum Server herstellen

Zunächst möchten wir eine Klasse erstellen, die eine Verbindung zu dem Server herstellt, den wir als Namen angeben: Connector:

```
class Connector
{
    private Socket _connectingSocket;
}
```

Nächste Methode für diese Klasse ist TryToConnect ()

Diese Methode hat einige interessante Dinge:

1. Erstellen Sie den Sockel.
2. Als nächstes schleife ich, bis die Steckdose angeschlossen ist
3. Jede Schleife hält den Thread nur für 1 Sekunde. Wir wollen den Server XD nicht DOS
4. Mit Connect () wird versucht, eine Verbindung zum Server herzustellen. Wenn dies fehlschlägt, wird eine Ausnahme ausgelöst, das Programm wird jedoch weiterhin mit dem Server verbunden. Sie können dafür eine Connect CallBack- Methode verwenden, aber ich werde einfach eine Methode aufrufen, wenn der Socket angeschlossen ist.
5. Beachten Sie, dass der Client jetzt versucht, sich an Port 1234 mit Ihrem lokalen PC zu verbinden.

```
 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();
 }
```

Senden einer Nachricht an den Server

Jetzt haben wir eine fast fertiggestellte oder Socket-Anwendung. Das einzige, was wir nicht haben, ist eine Klasse für das Senden einer Nachricht an den Server.

```
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();
        }
    }
```

Zum Schluss können Sie zwei Schaltflächen zum Verbinden und zum Senden einer Nachricht eingeben:

```
    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");
    }
```

Die Client-Klasse, die ich in diesem Beispiel verwendet habe

```
    public static void SetClient(Socket socket)
    {
        Id = 1;
        Socket = socket;
        Receive = new ReceivePacket(socket, Id);
        SendPacket = new SendPacket(socket);
    }
```

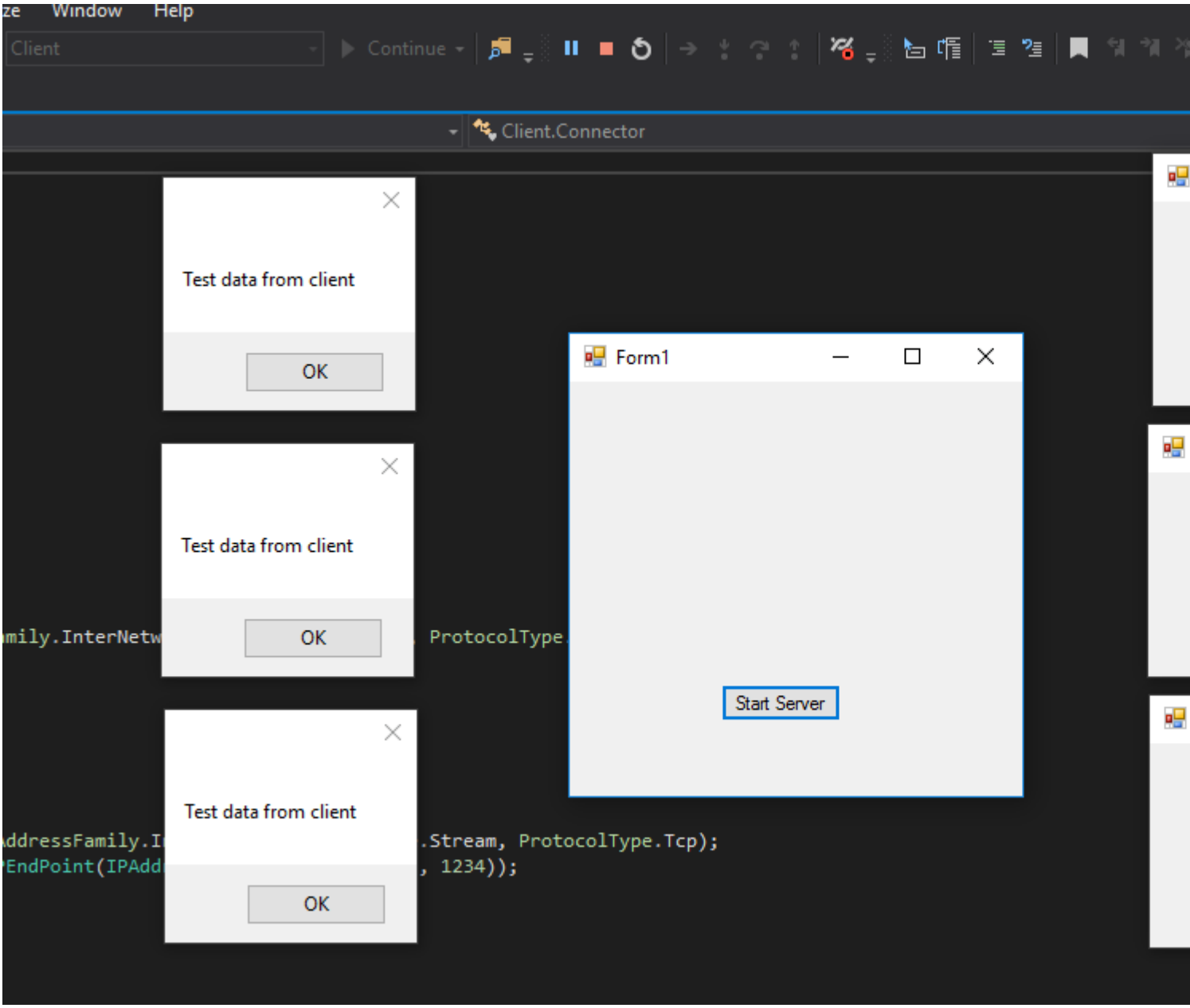
Beachten

Die Empfangsklasse vom Server entspricht der Empfangsklasse vom Client.

Fazit

Sie haben jetzt einen Server und einen Client. Sie können dieses grundlegende Beispiel ausarbeiten. Stellen Sie beispielsweise sicher, dass der Server auch Dateien oder andere Tings empfangen kann. Oder senden Sie eine Nachricht an den Client. Auf dem Server haben Sie eine Liste von Clients, so dass Sie, wenn Sie etwas erhalten, von Client wissen, von dem es gekommen ist.

Endergebnis:



Asynchroner Sockel online lesen: https://riptutorial.com/de/csharp/topic/9638/asynchroner-sockel

Kapitel 15: Attribute

Examples

Benutzerdefiniertes Attribut erstellen

```
//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;
    }
}
```

Verwenden eines Attributs

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Ein Attribut lesen

Die Methode `GetCustomAttributes` gibt ein Array von benutzerdefinierten Attributen zurück, die auf das Member angewendet werden. Nach dem Abrufen dieses Arrays können Sie nach einem oder mehreren bestimmten Attributen suchen.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Oder durchlaufen sie

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`GetCustomAttribute` Erweiterungsmethode `GetCustomAttribute` von `System.Reflection.CustomAttributeExtensions` ruft ein benutzerdefiniertes Attribut eines angegebenen Typs ab. Es kann auf jede `MemberInfo` angewendet werden.

```
var attribute = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute` auch eine generische Signatur, um den Attributtyp anzugeben, nach dem gesucht werden soll.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

Das boolesche Argument " `inherit` kann an beide Methoden übergeben werden. Wenn dieser Wert auf `true` gesetzt ist, sind die Vorfahren des Elements ebenfalls zu prüfen.

DebuggerDisplay-Attribut

Durch das Hinzufügen des `DebuggerDisplay` Attributs wird die Art und Weise geändert, in der der Debugger die Klasse anzeigt, wenn der `DebuggerDisplay` bewegt wird.

Ausdrücke, die in `{}` sind, werden vom Debugger ausgewertet. Dies kann eine einfache Eigenschaft wie im folgenden Beispiel oder eine komplexere Logik sein.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```

Durch das Hinzufügen von `,nq` vor der schließenden Klammer werden die Anführungszeichen entfernt, wenn eine Zeichenfolge `,nq` .

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

Obwohl allgemeine Ausdrücke in `{}` zulässig sind, werden sie nicht empfohlen. Das `DebuggerDisplay` Attribut wird als String in die Assembly-Metadaten geschrieben. Ausdrücke in `{}` werden nicht auf Gültigkeit geprüft. Ein `DebuggerDisplay` Attribut, das eine komplexere Logik enthält als eine einfache Arithmetik, könnte in C # gut funktionieren, aber derselbe Ausdruck, der in VB.NET ausgewertet wird, ist wahrscheinlich nicht syntaktisch gültig und erzeugt beim Debuggen einen Fehler.

Eine Möglichkeit, `DebuggerDisplay` zu machen, besteht darin, den Ausdruck in eine Methode oder Eigenschaft zu schreiben und ihn stattdessen aufzurufen.

```
[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}"";
    }
}
```

Man möchte, dass `DebuggerDisplay` alle oder nur einige der Eigenschaften ausgibt und beim Debuggen auch den Typ des Objekts überprüft.
Das folgende Beispiel umgibt auch die `#if DEBUG` mit `#if DEBUG` da `DebuggerDisplay` in Debugging-Umgebungen verwendet wird.

```
[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
}
```

Attribute für Anruferinformationen

Anrufer-Info-Attribute können verwendet werden, um Informationen über den Aufrufer an die aufgerufene Methode weiterzuleiten. Die Deklaration sieht folgendermaßen aus:

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
```

```
                          [CallerMemberName]string callerMemberName = "",
                          [CallerLineNumber]int callerLineNumber = 0,
                          [CallerFilePath]string callerFilePath = "")
{
    //perform logging
}
```

Und der Aufruf sieht so aus:

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

Beachten Sie, dass nur der erste Parameter explizit an die `LogException` Methode übergeben wird, während die übrigen Parameter zur Kompilierzeit mit den entsprechenden Werten versehen werden.

Der Parameter `callerMemberName` erhält den Wert `"Save"` - den Namen der aufrufenden Methode.

Der Parameter `callerLineNumber` erhält die Nummer der Zeile, in die der Aufruf der `LogException` Methode geschrieben wird.

Der Parameter 'callerFilePath' erhält den vollständigen Pfad der Datei, in der die `Save` Methode deklariert ist.

Ein Attribut von der Schnittstelle lesen

Es gibt keine einfache Möglichkeit, Attribute von einer Schnittstelle zu erhalten, da Klassen keine Attribute von einer Schnittstelle erben. Wenn Sie ein Interface implementieren oder Member in einer abgeleiteten Klasse überschreiben, müssen Sie die Attribute erneut deklarieren. In diesem Beispiel wäre die Ausgabe in allen drei Fällen also " `True` .

```
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
        }
    }
}
```

Eine Möglichkeit, Schnittstellenattribute abzurufen, besteht darin, über alle von einer Klasse implementierten Schnittstellen nach ihnen zu suchen.

```
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
```

Veraltetes Attribut

System.Obsolete ist ein Attribut, mit dem ein Typ oder ein Member mit einer besseren Version markiert wird und daher nicht verwendet werden sollte.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

Wenn die Klasse oben verwendet wird, gibt der Compiler die Warnung "Diese Klasse ist veraltet. Verwenden Sie stattdessen SomeOtherClass."

Attribute online lesen: https://riptutorial.com/de/csharp/topic/1062/attribute

Kapitel 16: Ausdrucksbäume

Einführung

Ausdrucksbäume sind in einer baumähnlichen Datenstruktur angeordnete Ausdrücke. Jeder Knoten im Baum ist eine Darstellung eines Ausdrucks, wobei ein Ausdruck Code ist. Eine In-Memory-Darstellung eines Lambda-Ausdrucks wäre ein Ausdrucksbaum, der die tatsächlichen Elemente (dh Code) der Abfrage enthält, jedoch nicht das Ergebnis. Ausdrucksbäume machen die Struktur eines Lambda-Ausdrucks transparent und explizit.

Syntax

- Ausdruck <TDelegate> name = lambdaExpression;

Parameter

Parameter	Einzelheiten
TDelegate	Der Delegattyp, der für den Ausdruck verwendet werden soll
LambdaExpression	Der Lambda-Ausdruck (zB `num => num < 5`)

Bemerkungen

Intro zu Ausdrucksbäumen

Woher wir kamen

In Ausdrucksbäumen wird zur Laufzeit "Quellcode" verwendet. `decimal CalculateTotalTaxDue(SalesOrder order)` Sie sich eine Methode vor, die die auf eine `decimal CalculateTotalTaxDue(SalesOrder order)` fällige Umsatzsteuer `decimal CalculateTotalTaxDue(SalesOrder order)` . Die Verwendung dieser Methode in einem .NET-Programm ist einfach - Sie nennen es einfach `decimal taxDue = CalculateTotalTaxDue(order);` . Was ist, wenn Sie es auf alle Ergebnisse einer Remote-Abfrage anwenden möchten (SQL, XML, Remote-Server usw.)? Diese entfernten Abfragequellen können die Methode nicht aufrufen! In all diesen Fällen müssten Sie normalerweise den Fluss umkehren. Machen Sie die gesamte Abfrage, speichern Sie sie im Arbeitsspeicher, durchlaufen Sie die Ergebnisse und berechnen Sie die Steuer für jedes Ergebnis.

So vermeiden Sie die Speicher- und Latenzprobleme der

Flussumkehrung

Ausdrucksbäume sind Datenstrukturen in einem Format eines Baums, wobei jeder Knoten einen Ausdruck enthält. Sie werden verwendet, um die kompilierten Anweisungen (wie zum Filtern von Daten verwendete Methoden) in Ausdrücke zu übersetzen, die außerhalb der Programmumgebung verwendet werden könnten, beispielsweise innerhalb einer Datenbankabfrage.

Das Problem hierbei ist, dass eine Remote-Abfrage *nicht auf unsere Methode zugreifen kann* . Wir könnten dieses Problem vermeiden, wenn wir stattdessen die *Anweisungen* für die Methode an die Fernabfrage senden. In unserem `CalculateTotalTaxDue` Beispiel bedeutet das, dass wir diese Informationen senden:

1. Erstellen Sie eine Variable, um die Gesamtsteuer zu speichern
2. Durchlaufen Sie alle Zeilen der Bestellung
3. Prüfen Sie für jede Zeile, ob das Produkt steuerpflichtig ist
4. Ist dies der Fall, multiplizieren Sie die Zeilensumme mit dem anwendbaren Steuersatz und addieren Sie diesen Betrag zur Gesamtsumme
5. Sonst nichts tun

Mit diesen Anweisungen kann die Remote-Abfrage die Arbeit ausführen, während sie die Daten erstellt.

Es gibt zwei Herausforderungen, um dies umzusetzen. Wie wandeln Sie eine kompilierte .NET-Methode in eine Liste von Anweisungen um und wie formatieren Sie die Anweisungen so, dass sie vom Remote-System verwendet werden können?

Ohne Ausdrucksbäume konnten Sie nur das erste Problem mit MSIL lösen. (MSIL ist der Assembler-artige Code, der vom .NET-Compiler erstellt wird.) Das Analysieren von MSIL ist *möglich* , aber nicht einfach. Selbst wenn Sie es richtig analysieren, kann es schwierig sein, die Absicht des ursprünglichen Programmierers mit einer bestimmten Routine zu bestimmen.

Ausdrucksbäume retten den Tag

Ausdrucksbäume behandeln genau diese Probleme. Sie stellen Programmanweisungen in einer Baumdatenstruktur dar, wobei jeder Knoten *eine Anweisung darstellt* und Verweise auf alle Informationen enthält, die Sie zur Ausführung dieser Anweisung benötigen. Eine `MethodCallExpression` hat beispielsweise einen Verweis auf 1) die `MethodInfo` , die aufgerufen werden soll, 2) eine Liste von `Expression` , die an diese Methode übergeben wird, 3) für Instanzmethoden und den `Expression` , für den Sie die Methode aufrufen. Sie können "durch den Baum gehen" und die Anweisungen auf Ihre Remote-Abfrage anwenden.

Ausdrucksbäume erstellen

Die einfachste Möglichkeit zum Erstellen eines Ausdrucksbaums besteht in einem Lambda-Ausdruck. Diese Ausdrücke sehen fast genauso aus wie normale C # -Methoden. Es ist wichtig zu wissen, dass dies *Compiler-Magie ist* . Wenn Sie zum ersten Mal einen Lambda-Ausdruck

erstellen, prüft der Compiler, wozu Sie ihn zuweisen. Wenn es ein ist `Delegate` - Typ (einschließlich `Action` oder `Func`), wandelt der Compiler den Lambda - Ausdruck in einen Delegierten. Wenn es sich um eine `LambdaExpression` (oder um einen `Expression<Action<T>>` oder einen `Expression<Func<T>>` dem es sich stark um `LambdaExpression` - `LambdaExpression`), wandelt der Compiler ihn in eine `LambdaExpression` . Hier setzt die Magie an. Hinter den Kulissen verwendet der Compiler *die Ausdrucksstruktur-API,* um Ihren Lambda-Ausdruck in eine `LambdaExpression` .

Lambda-Ausdrücke können nicht jede Art von Ausdrucksbaum erstellen. In diesen Fällen können Sie die Expressions-API manuell verwenden, um die Baumstruktur zu erstellen, die Sie benötigen. Im Beispiel für das Verständnis der Ausdrücke-API erstellen wir den `CalculateTotalSalesTax` Ausdruck mithilfe der API.

HINWEIS: Die Namen werden hier etwas verwirrend. Ein *Lambda-Ausdruck* (zwei Wörter, Kleinbuchstaben) bezieht sich auf den Codeblock mit einem Indikator `=>` . Es stellt eine anonyme Methode in C # dar und wird entweder in einen `Delegate` oder einen `Expression` konvertiert. Eine *`LambdaExpression`* (ein Wort, PascalCase) bezieht sich auf den Knotentyp innerhalb der Ausdruck-API, der eine Methode darstellt, die Sie ausführen können.

Ausdrucksbäume und LINQ

Eine der häufigsten Anwendungen von Ausdrucksbäumen ist bei LINQ- und Datenbankabfragen. LINQ paart eine Ausdrucksbaumstruktur mit einem Abfrageanbieter, um Ihre Anweisungen auf die Remote-Zielabfrage anzuwenden. Beispielsweise wandelt der Abfrage-Provider LINQ to Entity Framework einen Ausdrucksbaum in SQL um, der direkt für die Datenbank ausgeführt wird.

Wenn Sie alle Teile zusammenfügen, können Sie die wahre Stärke von LINQ erkennen.

1. Schreiben Sie eine Abfrage mit einem Lambda-Ausdruck: `products.Where(x => x.Cost > 5)`
2. Der Compiler wandelt diesen Ausdruck in einen Ausdrucksbaum mit den Anweisungen "prüfe, ob die Cost-Eigenschaft des Parameters größer als fünf ist".
3. Der Abfrageanbieter analysiert die Ausdrucksbaumstruktur und erstellt eine gültige SQL-Abfrage `SELECT * FROM products WHERE Cost > 5`
4. Der ORM projiziert alle Ergebnisse in POCOs und Sie erhalten eine Liste der Objekte zurück

Anmerkungen

- Ausdrucksbäume sind unveränderlich. Wenn Sie einen Ausdrucksbaum ändern möchten, müssen Sie einen neuen erstellen, den vorhandenen in den neuen kopieren (um einen Ausdrucksbaum zu durchsuchen, können Sie den `ExpressionVisitor`) und die gewünschten Änderungen vornehmen.

Examples

Erstellen von Ausdrucksbäumen mithilfe der API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

Ausdrucksbäume kompilieren

```
// 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));
```

Analysieren von Ausdrucksbäumen

```
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
```

Erstellen Sie Ausdrucksbäume mit einem Lambda-Ausdruck

Es folgt der grundlegendste Ausdrucksbaum, der von Lambda erstellt wird.

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

Um Ausdrucksbäume "von Hand" zu erstellen, sollte man die `Expression` Klasse verwenden.

Der obige Ausdruck wäre äquivalent zu:

```
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);
```

Grundlegendes zur API für Ausdrücke

Wir werden die Ausdrucksstruktur-API verwenden, um eine `CalculateSalesTax` Struktur zu erstellen. Im Klartext finden Sie hier eine Zusammenfassung der Schritte, die zum Erstellen des Baums erforderlich sind.

1. Prüfen Sie, ob das Produkt steuerpflichtig ist
2. Ist dies der Fall, multiplizieren Sie die Zeilensumme mit dem anwendbaren Steuersatz und geben Sie diesen Betrag zurück
3. Ansonsten 0 zurückgeben

```
//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);
```

Ausdrucksbaum Basic

Ausdrucksbäume repräsentieren Code in einer baumartigen Datenstruktur, wobei jeder Knoten ein Ausdruck ist

Expression Trees ermöglicht die dynamische Änderung von ausführbarem Code, die Ausführung von LINQ-Abfragen in verschiedenen Datenbanken und die Erstellung dynamischer Abfragen. Sie können Code, der durch Ausdrucksbäume dargestellt wird, kompilieren und ausführen.

Diese werden auch in der Dynamic Language Run-Time (DLR) verwendet, um Interoperabilität zwischen dynamischen Sprachen und .NET Framework bereitzustellen und Compiler-Autoren zu ermöglichen, Ausdrucksbäume anstelle von Microsoft Intermediate Language (MSIL) auszugeben.

Ausdrucksbäume können über erstellt werden

1. Anonymer Lambda-Ausdruck,
2. Manuell unter Verwendung des System.Linq.Expressions-Namespace.

Ausdrucksbäume aus Lambda-Ausdrücken

Wenn der Ausdruckstypvariable ein Lambda-Ausdruck zugewiesen wird, gibt der Compiler Code aus, um einen Ausdrucksbaum zu erstellen, der den Lambda-Ausdruck darstellt.

Die folgenden Codebeispiele zeigen, wie der C # -Compiler eine Ausdrucksbaumstruktur erstellt, die den Lambda-Ausdruck num => num <5 darstellt.

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

Ausdrucksbäume mithilfe der API

Ausdrucksbäume werden auch mit der **Ausdrucksklasse erstellt** . Diese Klasse enthält statische Factory-Methoden, die Ausdrucksstrukturknoten bestimmter Typen erstellen.

Nachfolgend finden Sie einige Baumknoten.

1. ParameterExpression
2. MethodCallExpression

Das folgende Codebeispiel zeigt, wie Sie mithilfe der API eine Ausdrucksbaumstruktur erstellen, die den Lambda-Ausdruck num => num <5 darstellt.

```
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 });
```

Untersuchung der Struktur eines Ausdrucks mithilfe von Besucher

Definieren Sie eine neue Besucherklasse, indem Sie einige der Methoden von ExpressionVisitor überschreiben:

```
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);
    }
}
```

Rufen Sie `Visit` an, um diesen Besucher für einen vorhandenen Ausdruck zu verwenden:

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

Ausdrucksbäume online lesen: https://riptutorial.com/de/csharp/topic/75/ausdrucksbaume

Kapitel 17: Ausnahmebehandlung

Examples

Grundsätzliche Ausnahmebehandlung

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Beachten Sie, dass die Behandlung aller Ausnahmen mit demselben Code oft nicht die beste Vorgehensweise ist.
Dies wird im Allgemeinen verwendet, wenn interne Routinen zur Behandlung von Ausnahmen als letzte Möglichkeit ausfallen.

Behandlung bestimmter Ausnahmetypen

```
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 */
}
```

Achten Sie darauf, dass Ausnahmen in der Reihenfolge ausgewertet werden und Vererbung angewendet wird. Sie müssen also mit den spezifischsten beginnen und mit ihrem Vorfahren enden. Zu einem bestimmten Zeitpunkt wird nur ein catch-Block ausgeführt.

Verwenden des Ausnahmeobjekts

Sie dürfen Ausnahmen in Ihrem eigenen Code erstellen und werfen. Das Instanziieren einer Ausnahme erfolgt auf dieselbe Weise wie jedes andere C # -Objekt.

```
Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

Sie können dann das `throw` Schlüsselwort verwenden, um die Ausnahme auszulösen:

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

Hinweis: Wenn Sie eine neue Ausnahme in einen catch-Block werfen, stellen Sie sicher, dass die ursprüngliche Ausnahme als "innere Ausnahme" übergeben wird, z

```
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());
    }
}
```

In diesem Fall wird davon ausgegangen, dass die Ausnahme nicht behandelt werden kann, aber der Nachricht einige nützliche Informationen hinzugefügt werden (und auf die ursprüngliche Ausnahme kann weiterhin über `ex.InnerException` durch einen äußeren Ausnahmeblock

zugegriffen werden).

Es wird etwas zeigen wie:

> System.DivideByZeroException: Kann nicht durch b geteilt werden, da es null ist --->
> System.DivideByZeroException: Es wurde versucht, durch Null zu teilen.
> bei UserQuery.g__DoSomething0_0 () in C: [...] \ LINQPadQuery.cs: Zeile 36
> --- Ende der inneren Ausnahmestapelverfolgung ---
> bei UserQuery.g__DoSomething0_0 () in C: [...] \ LINQPadQuery.cs: Zeile 42
> bei UserQuery.Main () in C: [...] \ LINQPadQuery.cs: Zeile 55

Wenn Sie dieses Beispiel in LinqPad ausprobieren, werden Sie feststellen, dass die Zeilennummern keine große Bedeutung haben (sie helfen Ihnen nicht immer). Das Übergeben eines hilfreichen Fehlertextes, wie oben vorgeschlagen, verringert jedoch häufig die Zeit, um den Ort des Fehlers aufzuspüren, was in diesem Beispiel eindeutig die Zeile ist

> c = a / b;

in der Funktion `DoSomething()` .

Versuchen Sie es in .NET Fiddle

Zum Schluss blockieren

```
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 */
}
```

Der `try / catch / finally` Block kann beim Lesen von Dateien sehr praktisch sein.
Zum Beispiel:

```
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.
}
```

Auf einen try-Block muss entweder ein `catch` oder ein `finally` Block folgen. Da es jedoch keinen

catch-Block gibt, führt die Ausführung zum Abbruch. Vor der Beendigung werden die Anweisungen innerhalb des finally-Blocks ausgeführt.

Beim Lesen von Dateien hätten wir einen `using` Block verwenden können, da `FileStream` (was `OpenRead` zurückgibt) `IDisposable` implementiert.

Selbst wenn im `try` Block eine `return` Anweisung vorhanden ist, wird der `finally` Block normalerweise ausgeführt. Es gibt einige Fälle, in denen dies nicht möglich ist:

- Wenn ein StackOverflow auftritt .
- `Environment.FailFast`
- Der Bewerbungsprozess wird in der Regel von einer externen Quelle abgebrochen.

IErrorHandler für WCF-Services implementieren

Die Implementierung von IErrorHandler für WCF-Dienste ist eine hervorragende Möglichkeit, die Fehlerbehandlung und Protokollierung zu zentralisieren. Die hier gezeigte Implementierung sollte alle nicht behandelten Ausnahmen abfangen, die als Ergebnis eines Aufrufs an einen Ihrer WCF-Dienste ausgelöst werden. In diesem Beispiel wird auch gezeigt, wie ein benutzerdefiniertes Objekt zurückgegeben wird und wie JSON anstelle der Standard-XML zurückgegeben wird.

Implementieren Sie IErrorHandler:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
```

```
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new
List<Type> { typeof(BaseDataResponseContract) }));

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process
them differently
            }

            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name,
webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

In diesem Beispiel hängen wir den Handler an das Serviceverhalten an. Sie könnten dies auch auf ähnliche Weise an IEndpointBehavior, IContractBehavior oder IOperationBehavior anhängen.

An Serviceverhalten anhängen:

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
```

```
BindingParameterCollection bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace
```

Configs in Web.config:

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

Hier sind ein paar Links, die zu diesem Thema hilfreich sein können:

https://msdn.microsoft.com/de-de/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/which-vier-verhalten-interfaces-exist-für-interacting-mit-aservice-or-client-description-what-methoden- umsetzen und

Andere Beispiele:

IErrorHandler gibt einen falschen Nachrichtentext zurück, wenn der HTTP-Statuscode nicht autorisiert ist

IErrorHandler scheint meine Fehler in der WCF nicht zu behandeln. Ideen?

Wie kann man einen benutzerdefinierten WCF-Fehlerbehandler veranlassen, eine JSON-Antwort mit Nicht-OK-HTTP-Code zurückzugeben?

Wie legen Sie den Content-Type-Header für eine HttpClient-Anforderung fest?

Benutzerdefinierte Ausnahmen erstellen

Sie können benutzerdefinierte Ausnahmen implementieren, die wie jede andere Ausnahme ausgelöst werden können. Dies ist sinnvoll, wenn Sie Ihre Ausnahmen während der Laufzeit von anderen Fehlern unterscheiden möchten.

In diesem Beispiel erstellen wir eine benutzerdefinierte Ausnahme, um die Probleme zu beheben, die die Anwendung beim Analysieren einer komplexen Eingabe haben kann.

Angepasste Ausnahmeklasse erstellen

Um eine benutzerdefinierte Ausnahme zu erstellen, erstellen Sie eine Unterklasse von `Exception` :

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

Benutzerdefinierte Ausnahmen werden sehr nützlich, wenn Sie dem Catcher zusätzliche Informationen zur Verfügung stellen möchten:

```
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;}
}
```

Wenn Sie nun `catch(ParserException x)` Sie über eine zusätzliche Semantik, um die

Ausnahmebehandlung zu `catch(ParserException x)` .

Benutzerdefinierte Klassen können die folgenden Funktionen implementieren, um zusätzliche Szenarien zu unterstützen.

erneut werfen

Während des Analysevorgangs ist die ursprüngliche Ausnahme immer noch von Interesse. In diesem Beispiel handelt es sich um eine `FormatException` da der Code versucht, eine Zeichenfolge zu analysieren, von der erwartet wird, dass sie eine Zahl ist. In diesem Fall sollte die benutzerdefinierte Ausnahme die Aufnahme der ' **InnerException** ' unterstützen:

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

Serialisierung

In einigen Fällen müssen Ihre Ausnahmen AppDomain-Grenzen überschreiten. Dies ist der Fall, wenn der Parser in einer eigenen AppDomain ausgeführt wird, um das erneute Laden neuer Parserkonfigurationen zu unterstützen. In Visual Studio können Sie eine `Exception` Vorlage zum Generieren von Code verwenden.

```
[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 verwenden

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

Sie können auch benutzerdefinierte Ausnahmen verwenden, um Ausnahmen abzufangen und zu verpacken. Auf diese Weise können viele verschiedene Fehler in einen einzelnen Fehlertyp umgewandelt werden, der für die Anwendung nützlicher ist:

```
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);
}
```

Wenn Sie Ausnahmen behandeln, indem Sie Ihre eigenen benutzerdefinierten Ausnahmen `InnerException` , sollten Sie im Allgemeinen einen Verweis auf die ursprüngliche Ausnahme in die `InnerException` Eigenschaft `InnerException` , wie oben gezeigt.

Sicherheitsbedenken

Wenn der Grund für die Ausnahme angezeigt wird, kann dies die Sicherheit beeinträchtigen, da Benutzer die inneren Abläufe Ihrer Anwendung sehen können. Dies kann eine schlechte Idee sein, die innere Ausnahme zu umgehen. Dies kann zutreffen, wenn Sie eine Klassenbibliothek erstellen, die von anderen verwendet wird.

So können Sie eine benutzerdefinierte Ausnahme auslösen, ohne die innere Ausnahme einzuwickeln:

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
```

```
}
```

Fazit

Beim Auslösen einer benutzerdefinierten Ausnahme (entweder beim Wrapping oder bei einer neuen, nicht verpackten Ausnahme) sollten Sie eine für den Aufrufer sinnvolle Ausnahme auslösen. Beispielsweise weiß ein Benutzer einer Klassenbibliothek möglicherweise nicht viel darüber, wie diese Bibliothek ihre interne Arbeit erledigt. Die Ausnahmen, die von den Abhängigkeiten der Klassenbibliothek ausgelöst werden, sind nicht sinnvoll. Der Benutzer wünscht vielmehr eine Ausnahme, die relevant ist, wie die Klassenbibliothek diese Abhängigkeiten fehlerhaft verwendet.

```
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);
}
```

Ausnahme Anti-Muster

Ausnahmen schlucken

Eine Ausnahme sollte immer auf folgende Weise wiederholt werden:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Durch erneutes Auslösen einer Ausnahme wie unten wird die ursprüngliche Ausnahme verschleiert und die ursprüngliche Stapelablaufverfolgung wird verloren gehen. Man sollte das niemals tun! Die Stapelverfolgung vor dem Fangen und Wiederherstellen geht verloren.

```
try
{
    ...
}
```

```
catch (Exception ex)
{
    ...
    throw ex;
}
```

Baseball-Ausnahmebehandlung

Ausnahmen sollten nicht als Ersatz für normale Flusssteuerungskonstrukte wie if-then-Anweisungen und while-Schleifen verwendet werden. Dieses Anti-Pattern wird manchmal als Baseball Exception Handling bezeichnet .

Hier ist ein Beispiel für das Anti-Pattern:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Hier ist ein besserer Weg, dies zu tun:

```
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());
```

catch (Ausnahme)

Es gibt fast keine Gründe (einige sagen: keine!), Um den generischen Ausnahmetyp in Ihrem Code abzufangen. Sie sollten nur die Ausnahmetypen abfangen, die Sie erwarten, da Sie sonst Fehler in Ihrem Code ausblenden.

```
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?
}
```

Besser machen:

```
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");
}
```

Wenn eine andere Ausnahme auftritt, lassen wir die Anwendung absichtlich abstürzen, sodass sie direkt in den Debugger eintritt und das Problem behoben werden kann. Wir dürfen kein Programm ausliefern, bei dem andere Ausnahmen als diese ohnehin auftreten. Es ist also kein Problem, einen Absturz zu haben.

Das folgende ist auch ein schlechtes Beispiel, da es Ausnahmen verwendet, um Programmierfehler zu umgehen. Dafür sind sie nicht gedacht.

```
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.
}
```

Ausnahmen / mehrere Ausnahmen von einer Methode zusammenfassen

Wer sagt, dass Sie nicht mehrere Ausnahmen in einer Methode werfen können. Wenn Sie es nicht gewohnt sind, mit AggregateExceptions herumzuspielen, können Sie versucht sein, Ihre eigene Datenstruktur zu erstellen, um viele Fehlfunktionen darzustellen. Es gibt natürlich eine andere Datenstruktur, bei der es sich nicht um eine Ausnahme handelt, beispielsweise die Ergebnisse einer Validierung. Selbst wenn Sie mit AggregateExceptions spielen, befinden Sie sich möglicherweise auf der Empfängerseite und haben immer damit zu tun, dass sie für Sie nicht von Nutzen sind.

Es ist ziemlich plausibel, eine Methode ausführen zu lassen, und obwohl es insgesamt ein Fehler sein wird, sollten Sie mehrere Dinge hervorheben, die in den ausgelösten Ausnahmen fehlerhaft waren. Ein Beispiel für dieses Verhalten ist, wie parallele Methoden funktionieren, wenn eine Task in mehrere Threads aufgeteilt wurde und eine beliebige Anzahl von ihnen Ausnahmen auslösen könnte, und dies muss gemeldet werden. Hier ein dummes Beispiel, wie Sie davon profitieren können:

```
    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);
        }
    }
```

Verschachtelung von Ausnahmen & Try-Catch-Blöcke.

Einer kann eine Ausnahme verschachteln / `try catch` block in den anderen einsetzen.

Auf diese Weise können kleine Codeblöcke verwaltet werden, die funktionieren, ohne dass der

gesamte Mechanismus beeinträchtigt wird.

```
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
}
```

Hinweis: Vermeiden Sie das Verschlucken von Ausnahmen, wenn Sie in den übergeordneten Fangblock werfen

Best Practices

Cheatsheet

TUN	NICHT
Kontrollfluss mit Kontrollanweisungen	Kontrollieren Sie den Fluss mit Ausnahmen
Verfolgen Sie ignorierte (absorbierte) Ausnahmen durch Protokollierung	Ausnahme ignorieren
Wiederholen Sie die Ausnahme mit `throw`	Ausnahme erneut auslösen - `throw new ArgumentNullException()` oder `throw ex`
Vorgegebene Systemausnahmen auslösen	Eigene Ausnahmen auslösen, die vordefinierten Systemausnahmen ähneln
Löst eine benutzerdefinierte / vordefinierte Ausnahme aus, wenn dies für die Anwendungslogik entscheidend ist	Werfen Sie benutzerdefinierte / vordefinierte Ausnahmen, um eine Warnung im Fluss anzugeben
Fangen Sie Ausnahmen auf, die Sie behandeln möchten	Fange jede Ausnahme

Verwalten Sie die Geschäftslogik NICHT mit Ausnahmen.

Die Flusskontrolle sollte NICHT durch Ausnahmen erfolgen. Verwenden Sie stattdessen bedingte Anweisungen. Wenn eine Kontrolle eindeutig mit der `if-else` Anweisung durchgeführt werden kann, dürfen Sie keine Ausnahmen verwenden, da dies die Lesbarkeit und Leistung verringert.

Betrachten Sie den folgenden Ausschnitt von Herrn Bad Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

Wenn die Ausführung `Console.WriteLine(myObject.ToString());` erreicht `Console.WriteLine(myObject.ToString());` Die Anwendung löst eine NullReferenceException aus. Mr. Bad Practices erkannte, dass `myObject` null ist, und bearbeitete sein Snippet, um die `NullReferenceException` `myObject` und zu behandeln:

```
// 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();
    }
}
```

Da das vorherige Snippet nur die Ausnahmelogik behandelt, was soll ich tun, wenn `myObject` an dieser Stelle nicht null ist? Wo soll ich diesen Teil der Logik behandeln? Gleich nach `Console.WriteLine(myObject.ToString());` ? Wie wäre es nach dem `try...catch` Block?

Wie wäre es mit Mr. Best Practices? Wie würde er damit umgehen?

```
// 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();
}
```

Mr. Best Practices erzielte dieselbe Logik mit weniger Code und einer klaren und verständlichen Logik.

KEINE Ausnahmen erneut auswerfen

Ausnahmen erneut zu werfen ist teuer. Dies wirkt sich negativ auf die Leistung aus. Für Code, der regelmäßig fehlschlägt, können Sie Entwurfsmuster verwenden, um Leistungsprobleme zu minimieren. In diesem Thema werden zwei Entwurfsmuster beschrieben, die hilfreich sind, wenn Ausnahmen die Leistung erheblich beeinträchtigen können.

Nehmen Sie keine Ausnahmen ohne Protokollierung auf

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

Schluck niemals Ausnahmen. Das Ignorieren von Ausnahmen spart diesen Moment, erzeugt jedoch später ein Chaos für die Wartbarkeit. Wenn Sie Ausnahmen protokollieren, sollten Sie immer die Ausnahmeinstanz protokollieren, sodass die gesamte Stack-Ablaufverfolgung protokolliert wird und nicht nur die Ausnahmemeldung.

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

Fangen Sie keine Ausnahmen auf, mit denen Sie nicht umgehen können

Viele Ressourcen, wie z. B. diese , drängen Sie dringend, zu überlegen, warum Sie an dem Ort, an dem Sie sie abfangen, eine Ausnahme feststellen. Sie sollten eine Ausnahme nur dann abfangen, wenn Sie an dieser Stelle damit umgehen können. Wenn Sie dort etwas tun können, um das Problem zu verringern, beispielsweise einen alternativen Algorithmus ausprobieren, eine Verbindung zu einer Sicherungsdatenbank herstellen, einen anderen Dateinamen versuchen, 30 Sekunden warten und es erneut versuchen oder einen Administrator benachrichtigen, können Sie den Fehler abfangen und das tun. Wenn Sie nichts plausibel und vernünftig tun können, lassen Sie es einfach los und lassen Sie die Ausnahme auf einer höheren Ebene erledigen. Wenn die Ausnahme ausreichend katastrophal ist und es keine vernünftige Option gibt, außer dass das gesamte Programm aufgrund des Schweregrads des Problems abstürzt, lassen Sie es abstürzen.

```
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.
```

Unbehandelt und Thread-Ausnahme

AppDomain.UnhandledException Dieses Ereignis **benachrichtigt** über nicht **erfasste** Ausnahmen. Es ermöglicht der Anwendung, Informationen über die Ausnahme zu protokollieren, bevor der Standard-Handler des Systems die Ausnahme an den Benutzer meldet und die Anwendung beendet Es können Aktionen ausgeführt werden, z. B. Speichern von Programmdaten für eine spätere Wiederherstellung. Vorsicht wird empfohlen, da Programmdaten beschädigt werden können, wenn keine Ausnahmen behandelt werden.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

Application.ThreadException Dieses Ereignis ermöglicht es Ihrer Windows Forms-Anwendung, ansonsten nicht behandelte Ausnahmen zu behandeln, die in Windows Forms-Threads auftreten. Hängen Sie Ihre Ereignishandler an das ThreadException-Ereignis an, um diese Ausnahmen zu behandeln, wodurch Ihre Anwendung in einem unbekannten Zustand bleibt. Wenn möglich, sollten Ausnahmen von einem strukturierten Ausnahmebehandlungsblock behandelt werden.

```
    /// <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);
    }
```

Und schließlich die Ausnahmebehandlung

```
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
    }
```

Eine Ausnahme auslösen

Ihr Code kann und sollte häufig eine Ausnahme auslösen, wenn etwas Ungewöhnliches passiert ist.

```
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.
}
```

Ausnahmebehandlung online lesen:
https://riptutorial.com/de/csharp/topic/40/ausnahmebehandlung

Kapitel 18: BackgroundWorker

Syntax

- `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.`

Bemerkungen

Durch das Ausführen lang andauernder Vorgänge innerhalb des UI-Threads kann Ihre Anwendung nicht mehr reagieren und dem Benutzer angezeigt werden, dass sie nicht mehr funktioniert. Es wird bevorzugt, dass diese Aufgaben in einem Hintergrundthread ausgeführt werden. Nach dem Abschluss kann die Benutzeroberfläche aktualisiert werden.

Wenn Sie während des Vorgangs des BackgroundWorker Änderungen an der Benutzeroberfläche vornehmen möchten , müssen Sie die Änderungen am Benutzeroberflächenthread aufrufen, normalerweise mithilfe der Methode Control.Invoke für das Steuerelement, das Sie aktualisieren. Andernfalls wird Ihr Programm eine Ausnahme auslösen.

Der BackgroundWorker wird normalerweise nur in Windows Forms-Anwendungen verwendet. In WPF-Anwendungen werden Aufgaben dazu verwendet, Arbeit auf Hintergrundthreads abzuladen (möglicherweise in Kombination mit Async / await). Marshalling-Aktualisierungen für den UI-Thread werden normalerweise automatisch ausgeführt, wenn die zu aktualisierende Eigenschaft INotifyPropertyChanged implementiert, oder manuell mithilfe des Dispatcher des UI-Thread.

Examples

Event-Handler einem BackgroundWorker zuordnen

Nachdem die Instanz von BackgroundWorker deklariert wurde, müssen Eigenschaften und Ereignishandler für die durchgeführten Aufgaben angegeben werden.

```
    /* 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;
    }
```

Eigenschaften einem BackgroundWorker zuweisen

Dadurch kann der BackgroundWorker zwischen Aufgaben abgebrochen werden

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

Dadurch kann der Arbeiter den Fortschritt zwischen dem Abschluss der Aufgaben ...

```
bgWorker.WorkerReportsProgress = true;

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

Erstellen einer neuen BackgroundWorker-Instanz

Ein BackgroundWorker wird normalerweise zum Ausführen von Aufgaben verwendet, die manchmal zeitraubend sind, ohne den UI-Thread zu blockieren.

```
// 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() { ...
```

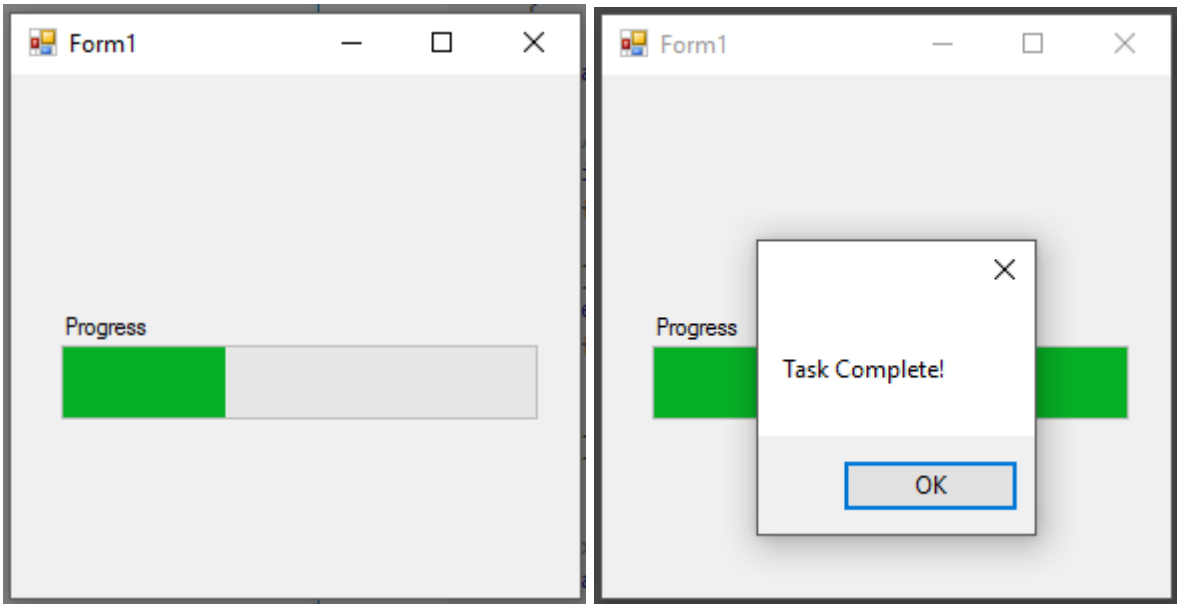
Verwenden eines BackgroundWorker zum Abschließen einer Aufgabe.

Das folgende Beispiel veranschaulicht die Verwendung eines BackgroundWorker zum Aktualisieren einer WinForms-Fortschrittsleiste. Der backgroundWorker aktualisiert den Wert der Fortschrittsleiste, ohne den UI-Thread zu blockieren. Dadurch wird eine reaktive UI angezeigt, während die Arbeit im Hintergrund ausgeführt wird.

```
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;
        }
    }


}
```

Das Ergebnis ist folgendes ...



BackgroundWorker online lesen: https://riptutorial.com/de/csharp/topic/1588/backgroundworker

Kapitel 19: Bedingte Anweisungen

Examples

If-Else-Anweisung

Die Programmierung erfordert im Allgemeinen oft eine `decision` oder eine `branch` innerhalb des Codes, um zu berücksichtigen, wie der Code unter verschiedenen Eingaben oder Bedingungen arbeitet. In der C # -Programmiersprache (und den meisten Programmiersprachen für diese Angelegenheit) ist der einfachste und manchmal nützlichste Weg, um einen Zweig in Ihrem Programm zu erstellen, die `If-Else` Anweisung.

Nehmen wir an, wir haben eine Methode (auch als Funktion bezeichnet), die einen int-Parameter annimmt, der eine Punktzahl von bis zu 100 darstellt, und die Methode gibt eine Meldung aus, die besagt, ob wir bestehen oder nicht.

```
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!");
    }
}
```

Wenn Sie sich diese Methode ansehen, stellen Sie möglicherweise diese Codezeile (`score >= 50`) in der `If` Anweisung fest. Dies kann als eine `boolean` Bedingung betrachtet werden. Wenn die Bedingung als " `true` ausgewertet wird, wird der Code ausgeführt, der sich zwischen dem `if { }` befindet.

Wenn diese Methode beispielsweise wie `PrintPassOrFail(60);` aufgerufen wurde: `PrintPassOrFail(60);` , die Ausgabe der Methode wäre ein Konsolendruck mit der Aufschrift ***Pass!*** da der Parameterwert von 60 größer oder gleich 50 ist.

Wenn die Methode jedoch wie `PrintPassOrFail(30);` aufgerufen wurde: `PrintPassOrFail(30);` Die Ausgabe der Methode würde den Ausdruck " ***Fail!"*** ausgeben. . Dies ist darauf zurückzuführen, dass der Wert 30 nicht größer oder gleich 50 ist. Daher wird der Code zwischen `else { }` anstelle der `If` Anweisung ausgeführt.

In diesem Beispiel haben wir gesagt, dass die *Punktzahl* auf 100 steigen sollte, was noch nicht berücksichtigt wurde. **Wenn Sie** berücksichtigen möchten, dass die *Punktzahl* nicht über 100 hinausgeht oder möglicherweise unter 0 fällt, **lesen Sie das** Beispiel der **If-Else-If-Else-Anweisung** .

If-Else If-Else-Anweisung

Dem Beispiel der **If-Else-Anweisung** folgend, ist es jetzt an der Zeit, die `Else If` Anweisung einzuführen. Die `Else If` Anweisung folgt direkt auf die `If` Anweisung in der **If-Else-If-Else-** Struktur, hat jedoch an sich eine ähnliche Syntax wie die `If` Anweisung. Es wird verwendet, um dem Code mehr Verzweigungen hinzuzufügen, als dies mit einer einfachen **If-Else-** Anweisung möglich ist.

In dem Beispiel von **If-Else Statement** wurde angegeben, dass die Punktzahl auf 100 steigt. Es wurden jedoch nie Kontrollen durchgeführt. Um dies zu beheben, können Sie die Methode aus der **If-Else-Anweisung folgendermaßen** ändern:

```
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!");
    }
}
```

Alle diese Anweisungen werden der Reihe nach von oben bis unten ausgeführt, bis eine Bedingung erfüllt ist. In diesem neuen Update der Methode haben wir zwei neue Zweige hinzugefügt, um jetzt die *außerhalb des Bereichs liegende* Punktzahl *zu berücksichtigen* .

Wenn wir beispielsweise die Methode in unserem Code als `PrintPassOFail(110);` , die Ausgabe wäre ein Konsolenausdruck, der sagt ***: Fehler: Score ist größer als 100!*** ; und wenn wir die Methode in unserem Code `PrintPassOrFail(-20);` B. `PrintPassOrFail(-20);` , würde die Ausgabe ***Error*** sagen ***: Score ist kleiner als 0!*** .

Anweisungen wechseln

Mit einer switch-Anweisung kann eine Variable auf Gleichheit mit einer Liste von Werten getestet werden. Jeder Wert wird als Fall bezeichnet und die einzuschaltende Variable wird für jeden Schaltfall geprüft.

Eine `switch` Anweisung ist oft prägnanter und verständlicher als die `if...else if... else..` -Anweisungen, wenn mehrere mögliche Werte für eine einzelne Variable getestet werden.

Die Syntax lautet wie folgt

```
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;
}
```

Es gibt einige Dinge, die bei der Verwendung der switch-Anweisung zu beachten sind

- Der in einer switch-Anweisung verwendete Ausdruck muss einen Integral- oder Aufzählungstyp haben oder von einem Klassentyp sein, in dem die Klasse eine einzige Konvertierungsfunktion in einen Integral- oder Aufzählungstyp hat.
- Sie können eine beliebige Anzahl von case-Anweisungen in einem Switch verwenden. Auf jeden Fall folgt der zu vergleichende Wert und ein Doppelpunkt. Die zu vergleichenden Werte müssen innerhalb jeder switch-Anweisung eindeutig sein.
- Eine switch-Anweisung kann einen optionalen Standardfall haben. Der Standardfall kann zum Ausführen einer Aufgabe verwendet werden, wenn keiner der Fälle wahr ist.
- Jeder Fall muss mit einer `break` Anweisung enden, es sei denn, es handelt sich um eine leere Anweisung. In diesem Fall wird die Ausführung im darunter liegenden Fall fortgesetzt. Die break-Anweisung kann auch weggelassen werden, wenn eine `return` , `throw` oder `goto case` Anweisung verwendet wird.

Beispiel kann bei den Noten angegeben werden

```
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;
}
```

If-Anweisungsbedingungen sind boolesche Standardausdrücke und -werte

Die folgende Aussage

```
if (conditionA && conditionB && conditionC) //...
```

ist genau äquivalent zu

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

Mit anderen Worten, die Bedingungen in der "if" -Anweisung bilden lediglich einen normalen booleschen Ausdruck.

Ein häufiger Fehler beim Schreiben von Bedingungsanweisungen ist der explizite Vergleich mit `true` und `false` :

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

Dies kann als neu geschrieben werden

```
if (conditionA && !conditionB && conditionC)
```

Bedingte Anweisungen online lesen: https://riptutorial.com/de/csharp/topic/3144/bedingte-anweisungen

Kapitel 20: Behandlung von FormatException beim Konvertieren von Zeichenfolgen in andere Typen

Examples

String in Ganzzahl konvertieren

Es gibt verschiedene Methoden, um eine `string` explizit in eine `integer` konvertieren, z.

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

Alle diese Methoden `FormatException` eine `FormatException` , wenn die Eingabezeichenfolge nicht numerische Zeichen enthält. Dafür müssen wir eine zusätzliche Ausnahmebehandlung (`try..catch`) schreiben, um sie in solchen Fällen zu behandeln.

Erklärung mit Beispielen:

Also sei unsere Eingabe:

```
string inputString = "10.2";
```

Beispiel 1: `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

Hinweis: *Gleiches gilt für die anderen genannten Methoden, nämlich -* *`Convert.ToInt16();`* *und* *`Convert.ToInt64();`*

Beispiel 2: `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

Wie umgehen wir das?

Wie bereits gesagt, benötigen wir für die Behandlung der Ausnahmen normalerweise einen `try..catch` wie unten gezeigt:

```
try
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}
```

Aber die Verwendung von `try..catch` wird nicht überall eine gute Praxis, und es können einige Szenarien, in denen wir wollten geben `0` , wenn der Eingang nicht stimmt, *(Wenn wir das obige Verfahren folgen müssen wir zuweisen `0` bis `convertedInt` aus der Fangblock).* Um solche Szenarien zu handhaben, können wir eine spezielle Methode namens `.TryParse()` .

Die `.TryParse()` -Methode verfügt über eine interne Ausnahmebehandlung, die Ihnen die Ausgabe an den `out` Parameter gibt und einen booleschen Wert zurückgibt, der den Konvertierungsstatus angibt *(`true` wenn die Konvertierung erfolgreich war; `false` wenn sie fehlgeschlagen ist)* Anhand des Rückgabewerts können wir den Konvertierungsstatus ermitteln. Schauen wir uns ein Beispiel an:

Verwendung 1: Speichern Sie den Rückgabewert in einer booleschen Variablen

```
 int convertedInt; // Be the required integer
 bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

Wir können die Variable `isSuccessConversion` nach der Ausführung überprüfen, um den Konvertierungsstatus zu überprüfen. Wenn es falsch ist, ist der Wert von `convertedInt` `0` *(keine Notwendigkeit, den Rückgabewert zu überprüfen, wenn `0` für einen Konvertierungsfehler gewünscht wird).*

Verwendung 2: Überprüfen Sie den Rückgabewert mit `if`

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

Verwendung 3: Ohne den Rückgabewert zu überprüfen, können Sie Folgendes verwenden, wenn Sie sich nicht für den Rückgabewert interessieren *(konvertiert oder nicht, `0` ist in Ordnung)*

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

Behandlung von FormatException beim Konvertieren von Zeichenfolgen in andere Typen online lesen: https://riptutorial.com/de/csharp/topic/2886/behandlung-von-formatexception-beim-

konvertieren-von-zeichenfolgen-in-andere-typen

Kapitel 21: Benannte Argumente

Examples

Benannte Argumente können Ihren Code klarer machen

Betrachten Sie diese einfache Klasse:

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

Vor C # 3.0 war es:

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

Sie können diesen Methodenaufruf mit **benannten Argumenten** noch deutlicher machen:

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

Benannte Argumente und optionale Parameter

Sie können benannte Argumente mit optionalen Parametern kombinieren.

Lassen Sie sich diese Methode ansehen:

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

Wenn Sie diese Methode *ohne* das Argument `retryCount` aufrufen `retryCount` :

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

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

Argumentreihenfolge ist nicht erforderlich

Sie können benannte Argumente in beliebiger Reihenfolge platzieren.

Beispielmethode:

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

Aufrufbeispiel:

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

Ergebnisse:

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

Named Arguments vermeidet Fehler bei optionalen Parametern

Verwenden Sie für optionale Parameter immer Named Arguments, um mögliche Fehler bei der Änderung der Methode zu vermeiden.

```
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
```

Der obige Code wird kompiliert und funktioniert einwandfrei, bis der Konstruktor eines Tages wie folgt geändert wird:

```
//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");
```

Best Practice, um Fehler zu vermeiden, wenn "jemand anderes im Team" Fehler gemacht hat:

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

Benannte Argumente online lesen: https://riptutorial.com/de/csharp/topic/2076/benannte-argumente

Kapitel 22: Benannte und optionale Argumente

Bemerkungen

Benannte Argumente

Ref: MSDN Mit benannten Argumenten können Sie ein Argument für einen bestimmten Parameter angeben, indem Sie das Argument mit dem Namen des Parameters und nicht mit der Position des Parameters in der Parameterliste verknüpfen.

Wie von MSDN gesagt, ein benanntes Argument,

- Ermöglicht die Übergabe des Arguments an die Funktion durch Zuordnen des Parameternamens.
- Es ist nicht notwendig, sich an die Parameterposition zu erinnern, die uns nicht immer bekannt ist.
- In der Parameterliste der aufgerufenen Funktion müssen Sie nicht nach der Reihenfolge der Parameter suchen.
- Wir können Parameter für jedes Argument anhand seines Namens angeben.

Optionale Argumente

Ref: MSDN Die Definition einer Methode, eines Konstruktors, eines Indexers oder eines Delegaten kann angeben, dass seine Parameter erforderlich sind oder dass sie optional sind. Jeder Aufruf muss Argumente für alle erforderlichen Parameter bereitstellen, kann jedoch Argumente für optionale Parameter auslassen.

Wie von MSDN gesagt, ein optionales Argument,

- Wir können das Argument in dem Aufruf weglassen, wenn dieses Argument ein optionales Argument ist
- Jedes optionale Argument hat einen eigenen Standardwert
- Es wird ein Standardwert verwendet, wenn wir den Wert nicht angeben
- Ein Standardwert für ein optionales Argument muss a sein
    - Konstanter Ausdruck.
    - Muss ein Werttyp wie enum oder struct sein.
    - Muss ein Ausdruck des Formularvorschlags sein (valueType)
- Sie muss am Ende der Parameterliste gesetzt werden

Examples

Benannte Argumente

Betrachten Sie folgendes ist unser Funktionsaufruf.

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

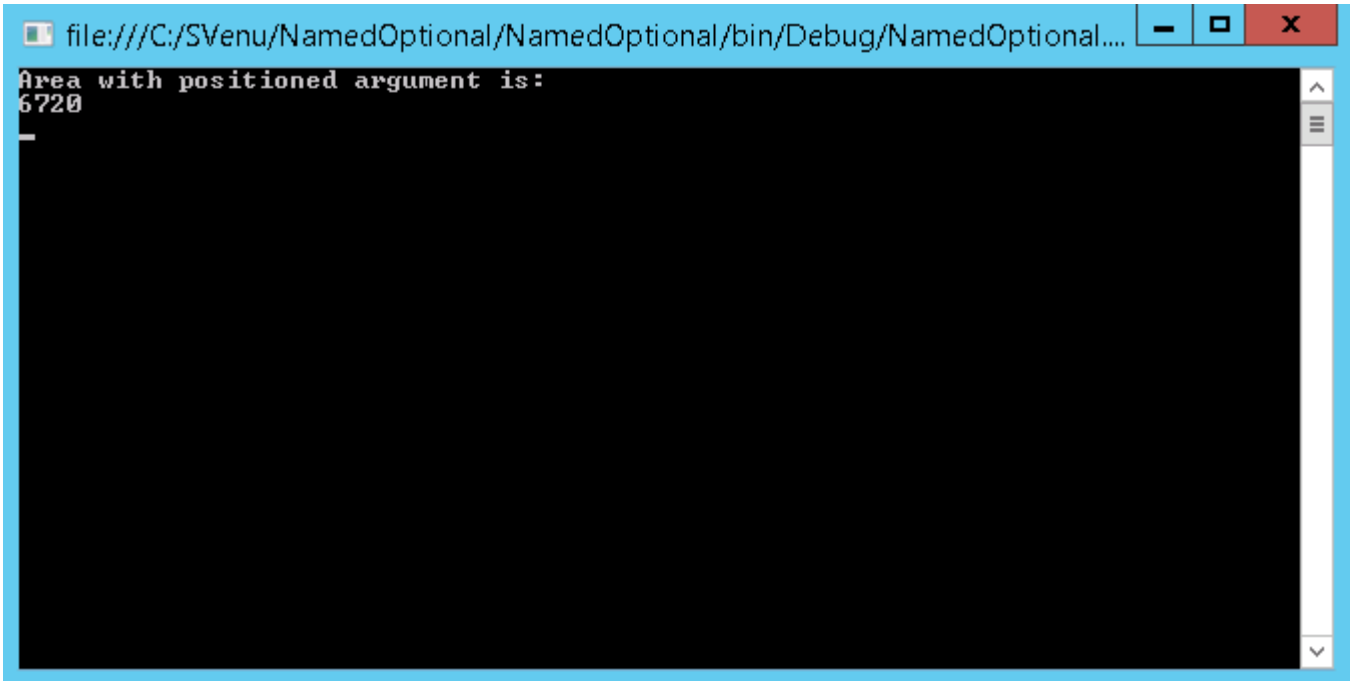
In diesem Fall lautet unser erstes Argument Länge (dh 120) und das zweite Argument Breite (dh 56). Und wir berechnen die Fläche anhand dieser Funktion. Und das Folgende ist die Funktionsdefinition.

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

Beim ersten Funktionsaufruf haben wir die Argumente nur an ihrer Position übergeben. Recht?

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

Wenn Sie dies ausführen, erhalten Sie eine Ausgabe wie folgt.



Hier kommen nun die Merkmale eines benannten Arguments. Bitte sehen Sie den vorhergehenden Funktionsaufruf.

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

```
Console.Read();
```

Hier geben wir die benannten Argumente im Methodenaufruf an.

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

Wenn Sie dieses Programm ausführen, erhalten Sie dasselbe Ergebnis. Wir können die Namen beim Methodenaufruf umgekehrt angeben, wenn wir die genannten Argumente verwenden.

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

Wenn Sie ein benanntes Argument verwenden, wird die Lesbarkeit Ihres Codes verbessert, wenn Sie dieses in Ihrem Programm verwenden. Es sagt einfach, was dein Argument sein soll oder was es ist.

Sie können auch die Positionsargumente angeben. Das heißt, eine Kombination aus Positionsargument und benanntem Argument.

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

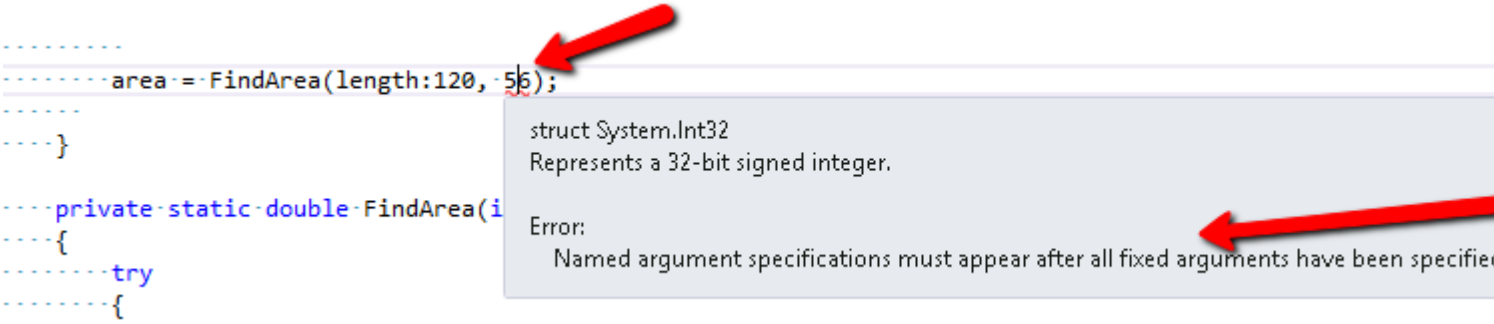
Im obigen Beispiel haben wir 120 als Länge und 56 als benanntes Argument für die Parameterbreite übergeben.

Es gibt auch einige Einschränkungen. Wir werden jetzt die Einschränkung eines benannten Arguments diskutieren.

Einschränkung der Verwendung eines benannten Arguments

Benannte Argumentspezifikationen müssen erscheinen, nachdem alle festen Argumente angegeben wurden.

Wenn Sie ein benanntes Argument vor einem festen Argument verwenden, wird ein Fehler beim Kompilieren wie folgt angezeigt.



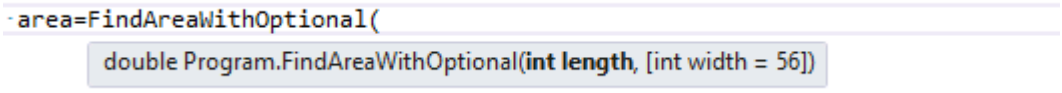
Benannte Argumentspezifikationen müssen erscheinen, nachdem alle festen Argumente angegeben wurden

Optionale Argumente

Betrachten wir als Vorgänger unsere Funktionsdefinition mit optionalen Argumenten.

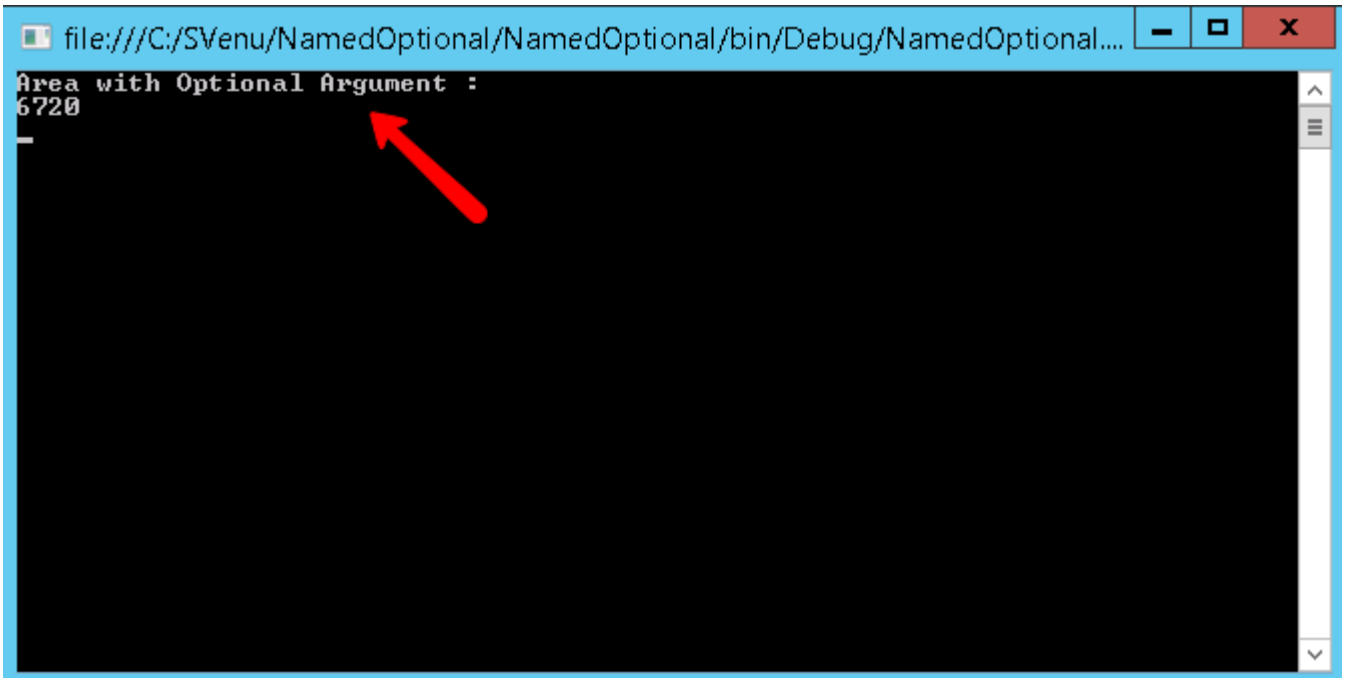
```
private static double FindAreaWithOptional(int length, int width=56)
       {
           try
           {
               return (length * width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Hier haben wir den Wert für width als optional festgelegt und als 56 angegeben. Wenn Sie beachten, zeigt der IntelliSense selbst das optionale Argument, wie in der folgenden Abbildung gezeigt.




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

Beachten Sie, dass beim Kompilieren keine Fehler aufgetreten sind, und Sie erhalten eine Ausgabe wie folgt.



Optionales Attribut verwenden.

Eine andere Möglichkeit, das optionale Argument zu implementieren, ist die Verwendung des Schlüsselworts `[Optional]` . Wenn Sie den Wert für das optionale Argument nicht übergeben, wird der Standardwert dieses Datentyps diesem Argument zugewiesen. Das `Optional` Schlüsselwort ist im Namespace "Runtime.InteropServices" vorhanden.

```
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
```

Wenn wir die Funktion aufrufen, erhalten wir 0, da das zweite Argument nicht übergeben wird und der Standardwert von int 0 ist. Daher ist das Produkt 0.

Benannte und optionale Argumente online lesen:
https://riptutorial.com/de/csharp/topic/5220/benannte-und-optionale-argumente

Kapitel 23: BigInteger

Bemerkungen

Wann verwenden?

`BigInteger` Objekte haben naturgemäß sehr viel RAM. Daher sollten sie nur verwendet werden, wenn es absolut notwendig ist, dh für Zahlen in wahrhaft astronomischem Maßstab.

Darüber hinaus sind alle arithmetischen Operationen an diesen Objekten um eine Größenordnung langsamer als ihre primitiven Pendants. Dieses Problem wird umso größer, je größer die Anzahl ist, da sie keine feste Größe haben. Es ist daher möglich, dass ein gefährlicher `BigInteger` einen Absturz verursacht, indem er den gesamten verfügbaren RAM verbraucht.

Alternativen

Wenn Geschwindigkeit für Ihre Lösung unerlässlich ist, ist es möglicherweise effizienter, diese Funktionalität selbst zu implementieren, indem Sie eine Klasse verwenden, die ein `Byte[]` und die erforderlichen Operatoren selbst überlastet. Dies erfordert jedoch einen erheblichen zusätzlichen Aufwand.

Examples

Berechnen Sie die erste 1.000-stellige Fibonacci-Zahl

Include `using System.Numerics` und Hinzufügen eines Verweises auf `System.Numerics` zum Projekt.

```
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);
        }
    }
}
```

Dieser einfache Algorithmus durchläuft die Fibonacci-Zahlen bis zu einer Länge von mindestens 1000 Dezimalstellen und gibt sie dann aus. Dieser Wert ist wesentlich größer als selbst ein `ulong` halten könnte.

Theoretisch ist die einzige Beschränkung für die `BigInteger` Klasse die Menge an RAM, die Ihre Anwendung `BigInteger` kann.

Hinweis: `BigInteger` ist nur in .NET 4.0 und höher verfügbar.

BigInteger online lesen: https://riptutorial.com/de/csharp/topic/5654/biginteger

Kapitel 24: Binäre Serialisierung

Bemerkungen

Das binäre Serialisierungsmodul ist Teil des .NET-Frameworks, die hier aufgeführten Beispiele sind jedoch spezifisch für C #. Im Vergleich zu anderen im .NET-Framework integrierten Serialisierungs-Engines ist der binäre Serialisierer schnell und effizient und erfordert in der Regel sehr wenig zusätzlichen Code, um ihn zu aktivieren. Es ist jedoch auch weniger tolerant gegenüber Codeänderungen. Das heißt, wenn Sie ein Objekt serialisieren und dann eine geringfügige Änderung an der Definition des Objekts vornehmen, wird es wahrscheinlich nicht korrekt deserialisiert.

Examples

Ein Objekt serialisierbar machen

Fügen Sie das Attribut `[Serializable]` , um ein gesamtes Objekt für die binäre Serialisierung zu markieren:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

Alle Member werden serialisiert, es sei denn, wir verwenden das Attribut `[NonSerialized]` explizit. In unserem Beispiel sind `X` , `Y` , `Z` und `Name` alle serialisiert.

Alle Mitglieder müssen bei der Deserialisierung anwesend sein, sofern sie nicht mit `[NonSerialized]` oder `[OptionalField]` . In unserem Beispiel sind `X` , `Y` und `Z` erforderlich, und die Deserialisierung schlägt fehl, wenn sie nicht im Stream vorhanden sind. `DontSerializeThis` wird immer auf `default(decimal)` (0) gesetzt. Wenn `Name` im Stream vorhanden ist, wird er auf diesen Wert gesetzt, andernfalls wird er auf `default(string)` (was null ist) gesetzt. Der Zweck von `[OptionalField]` besteht darin, ein wenig `[OptionalField]` bereitzustellen.

Serialisierungsverhalten mit Attributen steuern

Wenn Sie das Attribut `[NonSerialized]` , hat dieses Mitglied nach der Deserialisierung immer seinen Standardwert (z. B. 0 für ein `int` , null für `string` , false für ein `bool` usw.), unabhängig von

der Initialisierung im Objekt selbst (Konstruktoren, Deklarationen usw.). Um dies zu kompensieren, werden die Attribute `[OnDeserializing]` (kurz BEFORE Deserializing) und `[OnDeserialized]` (kurz AFTER Deserializing) zusammen mit ihren Gegenstücken `[OnSerializing]` und `[OnSerialized]` bereitgestellt.

Angenommen, wir möchten unserem Vektor eine "Bewertung" hinzufügen und sicherstellen, dass der Wert immer bei 1 beginnt. Die Art und Weise, wie sie unten geschrieben ist, wird nach der Deserialisierung 0 sein:

```
[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;
    }
}
```

Um dieses Problem zu beheben, können Sie einfach die folgende Methode innerhalb der Klasse hinzufügen, um sie auf 1 zu setzen:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Oder wenn wir einen berechneten Wert festlegen möchten, können wir warten, bis die Deserialisierung abgeschlossen ist, und dann festlegen:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

Ebenso können Sie mit `[OnSerializing]` und `[OnSerialized]` steuern, wie Dinge ausgeschrieben werden.

Mehr Kontrolle durch Implementierung von ISerializable

Dies würde mehr Kontrolle über die Serialisierung, das Speichern und Laden von Typen erhalten

Implementieren Sie die ISerializable-Schnittstelle und erstellen Sie einen leeren Konstruktor zum Kompilieren

```
[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));
    }
}
```

Für die Datenserialisierung können Sie den gewünschten Namen und den gewünschten Typ angeben

```
info.AddValue("_name", _name, typeof(string));
```

Wenn die Daten deserialisiert sind, können Sie den gewünschten Typ lesen

```
_name = (string)info.GetValue("_name", typeof(string));
```

Serialisierungssurrogate (Implementieren von ISerializationSurrogate)

Implementiert einen Serialisierungssurrogat-Selektor, mit dem ein Objekt die Serialisierung und Deserialisierung eines anderen Objekts durchführen kann

Außerdem kann eine Klasse, die selbst nicht serialisierbar ist, ordnungsgemäß serialisiert oder deserialisiert werden

Implementieren Sie die ISerializationSurrogate-Schnittstelle

```
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;
    }
}
```

Dann müssen Sie Ihren IFormatter über die Surrogate informieren, indem Sie einen SurrogateSelector definieren, initialisieren und Ihrem IFormatter zuweisen

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

Auch wenn die Klasse nicht als serialisierbar gekennzeichnet ist.

```
//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

Die komplette Lösung

```
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);
        }
    }
}
```

Serialisierungsbinder

In der Mappe können Sie überprüfen, welche Typen in Ihre Anwendungsdomäne geladen werden

Erstellen Sie eine von SerializationBinder geerbte Klasse

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Jetzt können wir prüfen, welche Typen geladen werden und auf dieser Basis entscheiden, was wir wirklich erhalten möchten

Damit Sie eine Sammelmappe verwenden können, müssen Sie sie dem BinaryFormatter hinzufügen.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

Die komplette Lösung

```
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);
        }
    }
}
```

Einige gotchas in Rückwärtskompatibilität

Dieses kleine Beispiel zeigt, wie Sie die Abwärtskompatibilität in Ihren Programmen verlieren können, wenn Sie sich vorher nicht darum kümmern. Und Möglichkeiten, den Serialisierungsprozess besser zu kontrollieren

Zuerst schreiben wir ein Beispiel für die erste Version des Programms:

Version 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

Und jetzt nehmen wir an, dass in der zweiten Version des Programms eine neue Klasse hinzugefügt wurde. Und wir müssen es in einem Array speichern.

Nun sieht der Code so aus:

Version 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

Und Code zum Serialisieren und Deserialisieren

```
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);
}
```

Was passiert also, wenn Sie die Daten im Programm von v2 serialisieren und versuchen, sie im Programm von v1 zu deserialisieren?

Sie erhalten eine Ausnahme:

```
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()
```

Warum?

Der ObjectManager verfügt über eine andere Logik zum Auflösen von Abhängigkeiten für Arrays sowie für Referenz- und Werttypen. Wir haben eine Reihe neuer Referenztypen hinzugefügt, die in unserer Baugruppe fehlen.

Wenn ObjectManager versucht, Abhängigkeiten aufzulösen, wird das Diagramm erstellt. Wenn das Array erkannt wird, kann es nicht sofort repariert werden, sodass eine Dummy-Referenz erstellt und das Array später korrigiert wird.

Und da dieser Typ nicht in der Baugruppe ist und Abhängigkeiten nicht behoben werden können. Aus irgendeinem Grund wird das Array nicht aus der Liste der Elemente für die Fixes entfernt. Am Ende wird eine Ausnahme "IncorrectNumberOfFixups" ausgelöst.

Es handelt sich um einige "Gotchas" im Prozess der Serialisierung. Aus irgendeinem Grund funktioniert es nicht korrekt nur für Arrays mit neuen Referenztypen.

```
A Note:
Similar code will work correctly if you do not use arrays with new classes
```

Und der erste Weg, um das Problem zu beheben und die Kompatibilität aufrechtzuerhalten?

- Verwenden Sie anstelle von Klassen eine Sammlung neuer Strukturen oder verwenden Sie ein Wörterbuch (mögliche Klassen), da es sich bei einem Wörterbuch um eine Sammlung von keyvaluepair (dessen Struktur) handelt.
- Verwenden Sie ISerializable, wenn Sie den alten Code nicht ändern können

Binäre Serialisierung online lesen: https://riptutorial.com/de/csharp/topic/4120/binare-serialisierung

Kapitel 25: Bindungsliste

Examples

N * 2-Iteration vermeiden

Dies wird in einem Windows Forms-Ereignishandler abgelegt

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

Die Ausführung, das Beheben und Ausführen der folgenden Anweisungen dauert lange:

```
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();
```

Element zur Liste hinzufügen

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

Bindungsliste online lesen: https://riptutorial.com/de/csharp/topic/182/bindungsliste--t-

Kapitel 26: C # 3.0-Funktionen

Bemerkungen

C # -Version 3.0 wurde als Teil von .NET-Version 3.5 veröffentlicht. Viele der mit dieser Version hinzugefügten Funktionen unterstützen LINQ (Language INtegrated Queries).

Liste der hinzugefügten Funktionen:

- LINQ
- Lambda-Ausdrücke
- Erweiterungsmethoden
- Anonyme Typen
- Implizit typisierte Variablen
- Objekt- und Collection-Initialisierer
- Automatisch implementierte Eigenschaften
- Ausdrucksbäume

Examples

Implizit typisierte Variablen (var)

Mit dem Schlüsselwort `var` kann ein Programmierer beim Kompilieren implizit eine Variable eingeben. `var` deklarationen haben den gleichen Typ wie explizit deklarierte Variablen.

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

Die Typen der obigen Variablen sind `int` , `double` , `StringBuilder` bzw. ein anonymer Typ.

Es ist wichtig zu beachten, dass eine `var` Variable nicht dynamisch typisiert wird. `SquaredNumber = Builder` ist ungültig, da Sie versuchen, ein `int` auf eine Instanz von `StringBuilder`

Sprachintegrierte Abfragen (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
```

Beispiel aus dem Wikipedia-Artikel zu C # 3.0, LINQ-Unterabschnitt

In Beispiel 1 wird eine Abfragesyntax verwendet, die ähnlich wie SQL-Abfragen aussehen sollte.

```
//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
```

Beispiel aus dem Wikipedia-Artikel zu C # 3.0, LINQ-Unterabschnitt

In Beispiel 2 wird die Methodensyntax verwendet, um dasselbe Ergebnis wie in Beispiel 1 zu erzielen.

Beachten Sie, dass in C # die LINQ-Abfragesyntax syntaktischer Zucker für die LINQ-Methodensyntax ist. Der Compiler übersetzt die Abfragen zum Zeitpunkt des Kompilierens in Methodenaufrufe. Einige Abfragen müssen in Methodensyntax ausgedrückt werden. Aus MSDN - "Beispielsweise müssen Sie einen Methodenaufruf verwenden, um eine Abfrage auszudrücken, die die Anzahl der Elemente abruft, die einer angegebenen Bedingung entsprechen."

Lambda-Ausdrücke

Lambda-Ausdrücke sind eine Erweiterung anonymer Methoden , die implizit typisierte Parameter und Rückgabewerte zulassen. Ihre Syntax ist weniger ausführlich als anonyme Methoden und folgt einem funktionalen Programmierstil.

```
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 );
    }
}
```

Der obige Code gibt die Summe der Quadrate der Zahlen 1 bis 10 an die Konsole aus.

Der erste Lambda-Ausdruck quadriert die Zahlen in der Liste. Da es nur 1 Parameter gibt, kann eine Klammer weggelassen werden. Sie können Klammern angeben, wenn Sie möchten:

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

oder geben Sie den Parameter explizit ein, aber dann sind Klammern erforderlich:

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

Der Lambda-Körper ist ein Ausdruck und hat eine implizite Rendite. Sie können auch einen Statement-Body verwenden, wenn Sie möchten. Dies ist nützlich für komplexere Lambdas.

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

Die select-Methode gibt ein neues IEnumerable mit den berechneten Werten zurück.

Der zweite Lambda-Ausdruck summiert die Zahlen in der Liste, die von der select-Methode zurückgegeben werden. Klammern sind erforderlich, da mehrere Parameter vorhanden sind. Die Typen der Parameter werden explizit angegeben, dies ist jedoch nicht erforderlich. Die untenstehende Methode ist gleichwertig.

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

Wie ist dieser hier:

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

Anonyme Typen

Anonyme Typen bieten eine bequeme Möglichkeit, einen Satz schreibgeschützter Eigenschaften in ein einzelnes Objekt zu kapseln, ohne zuerst einen Typ explizit definieren zu müssen. Der Typname wird vom Compiler generiert und ist auf Quellcodeebene nicht verfügbar. Der Typ jeder Eigenschaft wird vom Compiler abgeleitet.

Sie können anonyme Typen erstellen, indem Sie das `new` Schlüsselwort gefolgt von einer geschweiften Klammer *(`{`) verwenden* . In den geschweiften Klammern können Sie Eigenschaften wie im folgenden Code definieren.

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

Es ist auch möglich, ein Array anonymer Typen zu erstellen. Siehe Code unten:

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

Oder verwenden Sie es mit LINQ-Abfragen:

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

C # 3.0-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/3820/c-sharp-3-0-funktionen

Kapitel 27: C # 4.0-Funktionen

Examples

Optionale Parameter und benannte Argumente

Wir können das Argument in dem Aufruf weglassen, wenn dieses Argument ein optionales Argument ist. Jedes optionale Argument hat seinen eigenen Standardwert. Es wird ein Standardwert verwendet, wenn der Wert nicht angegeben wird. Ein Standardwert eines optionalen Arguments muss ein sein

1. Konstanter Ausdruck.
2. Muss ein Werttyp wie enum oder struct sein.
3. Muss ein Ausdruck des Formularvorschlags sein (valueType)

Sie muss am Ende der Parameterliste gesetzt werden

Methodenparameter mit Standardwerten:

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

Wie von MSDN gesagt, ein benanntes Argument,

Ermöglicht die Übergabe des Arguments an die Funktion durch Zuordnen des Parameternamens. Es ist nicht notwendig, sich die Parameterposition zu merken, die uns nicht immer bekannt ist. In der Parameterliste der aufgerufenen Funktion müssen Sie nicht nach der Reihenfolge der Parameter suchen. Wir können Parameter für jedes Argument anhand seines Namens angeben.

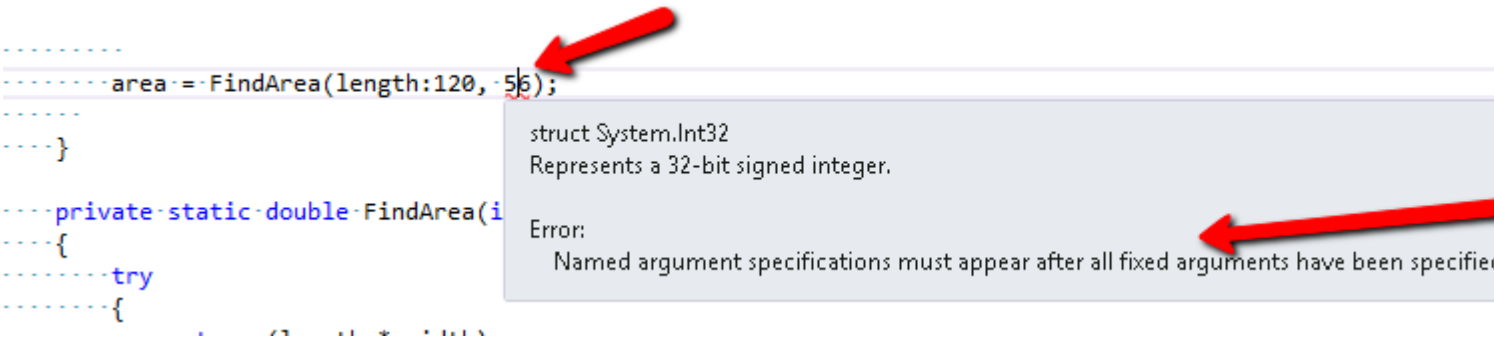
Benannte Argumente:

```
// 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);
```

Einschränkung der Verwendung eines benannten Arguments

Benannte Argumentspezifikationen müssen erscheinen, nachdem alle festen Argumente angegeben wurden.

Wenn Sie ein benanntes Argument vor einem festen Argument verwenden, wird ein Fehler beim Kompilieren wie folgt angezeigt.



Benannte Argumentspezifikationen müssen erscheinen, nachdem alle festen Argumente angegeben wurden

Abweichung

Bei generischen Schnittstellen und Delegaten können die Typparameter mit den `out` und `in` Schlüsselwörtern als *kovariant* oder *kontravariant* markiert werden. Diese Deklarationen werden dann für implizite und explizite Typkonvertierungen sowie sowohl für die Kompilierzeit als auch für die Laufzeit berücksichtigt.

Beispielsweise wurde die vorhandene Schnittstelle `IEnumerable<T>` als kovariant definiert:

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

Die bestehende Schnittstelle IComparer wurde als kontravariant definiert:

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

Optionales ref-Schlüsselwort bei Verwendung von COM

Das ref-Schlüsselwort für Aufrufer von Methoden ist jetzt optional, wenn in von COM-Schnittstellen bereitgestellte Methoden aufgerufen wird. Gegeben eine COM-Methode mit der Signatur

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

Der Aufruf kann nun als beides geschrieben werden

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

Dynamische Member-Suche

Eine neue Pseudo- `dynamic` wird in das C # -System eingeführt. Es wird als `System.Object` behandelt, aber zusätzlich ist jeder Memberzugriff (Methodenaufruf, Feld-, Eigenschafts- oder

Indexerzugriff oder ein Delegatenaufruf) oder die Anwendung eines Operators auf einen solchen Wert zulässig, ohne dass eine Typüberprüfung erfolgt und seine Auflösung wird auf die Laufzeit verschoben. Dies ist als Entenschreiben oder spätes Binden bekannt. Zum Beispiel:

```
// 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
```

In diesem Fall wird der dynamische Typ verwendet, um eine ausführlichere Reflexion zu vermeiden. Es verwendet immer noch Reflection unter der Haube, ist jedoch normalerweise schneller, dank Caching.

Diese Funktion zielt hauptsächlich auf die Interoperabilität mit dynamischen Sprachen ab.

```
// 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();
```

Dynamische Typen haben auch in meist statisch typisiertem Code Anwendungen. Beispielsweise ist Doppelversand möglich, ohne das Besuchermuster zu implementieren.

C # 4.0-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/3093/c-sharp-4-0-funktionen

Kapitel 28: C # 5.0-Funktionen

Syntax

- ***Async & Warten***
- public **Task** MyTask **Async** () {doSomething (); }

  Erwarte MyTaskAsync ();
- public **Task** `<string>` MyStringTask **Async** () {return getSomeString (); }

  string MyString = Erwarte MyStringTaskAsync ();
- ***Anruferinformationsattribute***
- public void MyCallerAttributes (Zeichenfolge MyMessage,

  [CallerMemberName] Zeichenfolge MemberName = "",

  [CallerFilePath] string SourceFilePath = "",

  [CallerLineNumber] int LineNumber = 0)
- Trace.WriteLine ("My Message:" + MyMessage);

  Trace.WriteLine ("Member:" + MemberName);

  Trace.WriteLine ("Source File Path:" + SourceFilePath);

  Trace.WriteLine ("Line Number:" + LineNumber);

Parameter

Methode / Modifikator mit Parameter	Einzelheiten
`Type<T>`	`T` ist der Rückgabetyp

Bemerkungen

C # 5.0 ist mit Visual Studio .NET 2012 gekoppelt

Examples

Async & Warten

`async` und `await` sind zwei Operatoren, die die Leistung verbessern sollen, indem sie Threads `async` und `await` , bis die Operationen abgeschlossen sind, bevor sie fortfahren.

Hier ein Beispiel, wie Sie einen String abrufen, bevor Sie seine Länge zurückgeben:

```
//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);
}
```

Hier ist ein weiteres Beispiel für das Herunterladen einer Datei und das Behandeln, was passiert, wenn sich der Fortschritt geändert hat und der Download abgeschlossen ist (es gibt zwei Möglichkeiten, dies durchzuführen):

Methode 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!")
}
```

Methode 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();
}
```

Anruferinformationsattribute

CIAs sind als einfache Methode gedacht, um Attribute von dem abzurufen, was die Zielmethode aufruft. Es gibt wirklich nur eine Möglichkeit, sie zu verwenden, und es gibt nur drei Attribute.

Beispiel:

```
//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);
}
```

Beispielausgabe:

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

C # 5.0-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/4584/c-sharp-5-0-funktionen

Kapitel 29: C # 6.0-Funktionen

Einführung

Diese sechste Iteration der C # -Sprache wird vom Roslyn-Compiler bereitgestellt. Dieser Compiler wurde mit Version 4.6 von .NET Framework veröffentlicht. Er kann jedoch abwärtskompatibel Code generieren, um frühere Framework-Versionen als Ziel festzulegen. C # -Version 6-Code kann vollständig abwärtskompatibel zu .NET 4.0 kompiliert werden. Es kann auch für frühere Frameworks verwendet werden, einige Funktionen, die zusätzliche Framework-Unterstützung erfordern, funktionieren jedoch möglicherweise nicht ordnungsgemäß.

Bemerkungen

Die sechste Version von C # wurde im Juli 2015 zusammen mit Visual Studio 2015 und .NET 4.6 veröffentlicht.

Es enthält nicht nur einige neue Sprachfunktionen, sondern auch eine vollständige Umschreibung des Compilers. Zuvor war `csc.exe` eine native Win32-Anwendung, die in C ++ geschrieben wurde. Mit C # 6 ist es jetzt eine in C # geschriebene .NET-verwaltete Anwendung. Diese Umschreibung wurde als Projekt "Roslyn" bezeichnet und der Code ist jetzt Open Source und auf GitHub verfügbar.

Examples

Betreiber Nameof

Der `nameof` Operator gibt den Namen eines `nameof` als `string` . Dies ist nützlich, wenn Ausnahmen im Zusammenhang mit Methodenargumenten `INotifyPropertyChanged` und auch `INotifyPropertyChanged` implementiert `INotifyPropertyChanged` .

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

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

Der `nameof` Operator wird zur Kompilierzeit ausgewertet und wandelt den Ausdruck in ein String-Literal um. Dies ist auch nützlich für Zeichenfolgen, die nach ihrem Member benannt werden, das sie verfügbar macht. Folgendes berücksichtigen:

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

```
}
```

Da `nameof` Ausdrücke Kompilierung-Konstanten sind, können sie in Attribute, werden verwendet `case` Etiketten, `switch` Anweisungen, und so weiter.

Es ist praktisch, `nameof` mit `Enum` s zu verwenden. Anstatt:

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

es ist möglich zu verwenden:

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

Die Ausgabe wird in beiden Fällen `One` .

Der `nameof` Operator kann mit einer statischen Syntax auf nicht statische Member zugreifen. Statt zu tun:

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

Kann ersetzt werden durch:

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

Die Ausgabe wird in beiden Beispielen `Length` . Letzteres verhindert jedoch die Erstellung unnötiger Instanzen.

Obwohl der `nameof` Operators mit den meisten Sprachkonstrukten funktioniert, gibt es einige Einschränkungen. Beispielsweise können Sie den Operator `nameof` für offene generische Typen oder Methodenrückgabewerte verwenden:

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

Wenn Sie ihn auf einen generischen Typ anwenden, wird der generische Typparameter ignoriert:

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

Weitere Beispiele finden Sie zu diesem Thema gewidmet `nameof` .

Problemumgehung für frühere Versionen (mehr Details)

Obwohl der `nameof` Operator für Versionen vor 6.0 nicht in C # vorhanden ist, kann mit `MemberExpression` ähnliche Funktionalität wie in der folgenden verwendet werden:

6,0

Ausdruck:

```
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;
}
```

Verwendungszweck:

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

Beachten Sie, dass bei diesem Ansatz bei jedem Aufruf ein Ausdrucksbaum erstellt wird. `nameof` ist die Leistung im Vergleich zum Operator `nameof` , der zur Kompilierzeit ausgewertet wird, deutlich schlechter und hat zur Laufzeit `nameof` Overhead.

Mitglieder mit Ausdrucksfunktion

Expression-body-Funktionsmitglieder erlauben die Verwendung von Lambda-Ausdrücken als Elementkörper. Bei einfachen Mitgliedern kann dies zu sauberem und lesbarerem Code führen.

Funktionen mit Ausdrucksfunktionen können für Eigenschaften, Indexer, Methoden und Operatoren verwendet werden.

Eigenschaften

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

Ist äquivalent zu:

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

Wenn eine Ausdrucksfunktion mit einer Eigenschaft verwendet wird, wird die Eigenschaft als reine Getter-Eigenschaft implementiert.

Demo anzeigen

Indexer

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

Ist äquivalent zu:

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

Methoden

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

Ist äquivalent zu:

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

Welche auch mit `void` methoden verwendet `void` können:

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

Eine Überschreibung von `ToString` könnte der `Pair<T>` -Klasse hinzugefügt werden:

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

Darüber hinaus funktioniert dieser vereinfachte Ansatz mit dem `override` :

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

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

Operatoren

Dies kann auch von Operatoren verwendet werden:

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

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

Einschränkungen

Mitglieder mit Ausdrucksfunktion haben einige Einschränkungen. Sie können nicht blockieren Aussagen und andere Aussagen enthalten , die Blöcke enthalten: `if` , `switch` , `for` , `foreach` , `while` , `do` , `try` , usw.

Einige `if` Anweisungen können durch ternäre Operatoren ersetzt werden. Einige `for` und `foreach` - Anweisungen können auf LINQ - Abfragen umgewandelt werden, zum Beispiel:

```
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());
```

In allen anderen Fällen kann die alte Syntax für Funktionsmitglieder verwendet werden.

Funktionsbesetzte mit Ausdrucksfunktion können `async` / `await` , sind jedoch häufig überflüssig:

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

Kann ersetzt werden durch:

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

Ausnahmefilter

Ausnahmefilter geben Entwicklern die Möglichkeit, einem catch- Block eine Bedingung (in Form eines `boolean` Ausdrucks) hinzuzufügen, sodass der `catch` nur ausgeführt werden kann, wenn die Bedingung als `true` ausgewertet wird.

Ausnahmefilter ermöglichen die Weitergabe von Debug-Informationen in der ursprünglichen Ausnahme, wobei die `if` Anweisung innerhalb eines `catch` Blocks verwendet wird und die Ausnahme erneut ausgelöst wird, um die Weitergabe von Debug-Informationen in der ursprünglichen Ausnahme zu stoppen. Bei Ausnahmefiltern breitet sich die Ausnahme im Aufrufstapel weiter nach oben aus, *sofern* die Bedingung *nicht* erfüllt ist. Ausnahmefilter erleichtern das Debugging daher erheblich. Anstatt bei der `throw` Anweisung anzuhalten, stoppt der Debugger bei der Anweisung, die die Ausnahme auslöst, wobei der aktuelle Status und alle lokalen Variablen erhalten bleiben. Crash Dumps sind auf ähnliche Weise betroffen.

> Ausnahmefilter werden von der **CLR von** Anfang an unterstützt und sind seit über einem Jahrzehnt über VB.NET und F # verfügbar, indem ein Teil des Ausnahmebehandlungsmodells der CLR verfügbar gemacht wird. Erst nach der Veröffentlichung von C # 6.0 war die Funktionalität auch für C # -Entwickler verfügbar.

Ausnahmefilter verwenden

Ausnahmefilter werden verwendet, indem eine `when` Klausel an den `catch` Ausdruck `catch` . Es ist möglich, einen beliebigen Ausdruck zu verwenden, der einen `bool` in einer `when` Klausel `bool` (mit Ausnahme von await). Auf die deklarierte Exception-Variable `ex` aus der `when` Klausel zugegriffen werden:

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

Mehrere `catch` Blöcke mit `when` Klauseln können kombiniert werden. Die erste `when` Klausel, die `true` zurückgibt, bewirkt, dass die Ausnahme abgefangen wird. Sein `catch` Block wird eingegeben, während die anderen `catch` Klauseln ignoriert werden (ihre `when` Klauseln werden nicht ausgewertet). Zum Beispiel:

```
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
{ ... }
```

Riskante Wannenklausel

> **Vorsicht**
>
> Die Verwendung von Ausnahmefiltern kann riskant sein: Wenn eine `Exception` aus der `when` Klausel ausgelöst wird, wird die `Exception` aus der `when` Klausel ignoriert und als `false` behandelt. Dieser Ansatz ermöglicht es Entwicklern zu schreiben , `when` Klausel ohne Pflege ungültiger Fälle zu nehmen.

Das folgende Beispiel veranschaulicht ein solches Szenario:

```
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);
}
```

Demo anzeigen

Beachten Sie, dass Ausnahmefilter die verwirrenden Probleme mit der Zeilennummer vermeiden, die mit der Verwendung von `throw` wenn fehlerhafter Code sich in derselben Funktion befindet. In diesem Fall wird die Zeilennummer beispielsweise als 6 anstelle von 3 angegeben:

```
1. int a = 0, b = 0;
2. try {
```

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

Die Ausnahmezeilennummer wird als 6 gemeldet, da der Fehler mit der `throw` Anweisung in Zeile 6 abgefangen und erneut ausgelöst wurde.

Dasselbe passiert nicht mit Ausnahmefiltern:

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

In diesem Beispiel `a` 0 ist, dann `catch` wird Klausel ignoriert aber 3 als Zeilennummer angegeben. Dies liegt daran, dass sie **den Stapel nicht abwickeln** . Insbesondere wird die Ausnahme *nicht in* Zeile 5 *abgefangen* , weil `a` tatsächlich gleich `0` und daher keine Möglichkeit besteht, dass die Ausnahme erneut in Zeile 6 ausgelöst wird, da Zeile 6 nicht ausgeführt wird.

Protokollierung als Nebeneffekt

Methodenaufrufe in der Bedingung können Nebeneffekte verursachen, sodass Ausnahmefilter verwendet werden können, um Code mit Ausnahmen auszuführen, ohne sie zu erfassen. Ein häufiges Beispiel, das dies nutzt, ist eine `Log` Methode, die immer `false` zurückgibt. Dadurch können Protokollinformationen während des Debuggens verfolgt werden, ohne dass die Ausnahme erneut ausgelöst werden muss.

> **Beachten Sie, dass** dies zwar eine bequeme Art der Protokollierung zu sein scheint, aber es kann riskant sein, insbesondere wenn Protokollierungsbaugruppen von Drittanbietern verwendet werden. Dies kann zu Ausnahmen führen, wenn in nicht offensichtlichen Situationen protokolliert wird, die möglicherweise nicht leicht erkannt werden (siehe **Abschnitt Risikobereitschaft `when(...)`** oben).

```
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;
}
```

Demo anzeigen

In früheren Versionen von C # bestand der übliche Ansatz darin, die Ausnahme zu protokollieren und erneut auszulösen.

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);
}
```

Demo anzeigen

Die `finally` blockieren

Der `finally` Block wird jedes Mal ausgeführt, ob die Ausnahme ausgelöst wird oder nicht. Eine Feinheit mit Ausdrücken in `when` wird Ausnahme Filter ausgeführt werden weiter den Stapel *vor* dem inneren Eingabe `finally` blockiert. Dies kann zu unerwarteten Ergebnissen und Verhalten führen, wenn Code versucht, den globalen Status (z. B. den Benutzer oder die Kultur des aktuellen Threads) zu ändern und ihn in einen `finally` Block zurückzusetzen.

Beispiel: `finally` blockieren

```
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");
    }
}
```

Produzierte Ausgabe:

Start
EvaluatesTo: True
Innerlich zum Schluss
Fang
Äußer Endlich

Demo anzeigen

Wenn im `SomeOperation` Beispiel die Methode `SomeOperation` nicht die Änderung der globalen Zustandsänderungen in die `when` Klauseln des Aufrufers " `SomeOperation` " möchte, sollte sie auch einen `catch` Block zum Ändern des Zustands enthalten. Zum Beispiel:

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

```
        Console.WriteLine("Inner Finally");
    }
}
```

Es ist auch üblich, dass `IDisposable` Helfer-Klassen die Semantik der Verwendung von Blöcken nutzen, um dasselbe Ziel zu erreichen, da `IDisposable.Dispose` immer aufgerufen wird, bevor eine innerhalb eines `using` Blocks aufgerufene Ausnahme den Stack sprudelt.

Auto-Property-Initialisierer

Einführung

Eigenschaften können nach dem Schließen mit dem Operator `=` initialisiert werden `}` . Die folgende `Coordinate` zeigt die verfügbaren Optionen zum Initialisieren einer Eigenschaft:

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
}
```

Accessoren mit unterschiedlicher Sichtbarkeit

Sie können Auto-Eigenschaften initialisieren, deren Accessoren unterschiedlich sichtbar sind. Hier ein Beispiel mit einem geschützten Setter:

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

Der Accessor kann auch `internal` , `internal protected` oder `private` .

Schreibgeschützte Eigenschaften

Neben der Flexibilität bei der Sichtbarkeit können Sie auch schreibgeschützte Auto-Eigenschaften initialisieren. Hier ist ein Beispiel:

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

Dieses Beispiel zeigt auch, wie eine Eigenschaft mit einem komplexen Typ initialisiert wird. Außerdem können Auto-Eigenschaften nicht nur schreibgeschützt sein, so dass auch die Initialisierung nur zum Schreiben ausgeschlossen ist.

Alter Stil (vor C # 6.0)

Vor C # 6 erforderte dies viel ausführlicheren Code. Wir haben eine zusätzliche Variable namens Backing-Eigenschaft für die Eigenschaft verwendet, um einen Standardwert anzugeben oder die öffentliche Eigenschaft wie folgt zu initialisieren:

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;
    }
}
```

Hinweis: *Vor C # 6.0 konnten Sie die* ***automatisch implementierten Eigenschaften*** *Lesen und Schreiben (Eigenschaften mit einem Getter und einem Setter) innerhalb des Konstruktors immer noch initialisieren, die Eigenschaft konnte jedoch nicht mit ihrer Deklaration inline initialisiert werden*

Demo anzeigen

Verwendungszweck

Initialisierer müssen statische Ausdrücke auswerten, genau wie Feldinitialisierer. Wenn Sie auf nicht statische Member verweisen müssen, können Sie Eigenschaften in Konstruktoren wie zuvor initialisieren oder Eigenschaften mit Ausdruck verwenden. Nicht statische Ausdrücke wie die unten stehende (auskommentiert) erzeugen einen Compiler-Fehler:

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

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

Statische Methoden **können** jedoch zum Initialisieren von Auto-Eigenschaften verwendet werden:

```
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;
    }
}
```

Diese Methode kann auch auf Eigenschaften mit unterschiedlichen Zugriffsstufen angewendet werden:

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

Der Auto-Property-Initializer ermöglicht die Zuweisung von Eigenschaften direkt in ihrer Deklaration. Bei schreibgeschützten Eigenschaften werden alle Anforderungen berücksichtigt, um sicherzustellen, dass die Eigenschaft unveränderlich ist. Betrachten Sie beispielsweise die `FingerPrint` Klasse im folgenden Beispiel:

```
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;
}
```

Demo anzeigen

Warnhinweise

Achten Sie darauf, Auto-Property- oder Feldinitialisierer nicht mit ähnlich aussehenden Ausdruckskörpermethoden zu verwechseln, die `=>` im Gegensatz zu `=` , und Felder, die `{ get; }` .

Zum Beispiel sind die folgenden Deklarationen unterschiedlich.

```
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>();
}
```

Fehlt `{ get; }` in der Eigenschaftsdeklaration führt zu einem öffentlichen Feld. Sowohl das

schreibgeschützte Auto-Eigenschaft- `Users1` als auch das Read-Write-Feld `Users2` werden nur einmal initialisiert. Ein öffentliches Feld ermöglicht jedoch das Ändern der Erfassungsinstanz von außerhalb der Klasse, was normalerweise unerwünscht ist. Um eine schreibgeschützte Auto-Eigenschaft mit dem Ausdruck body in eine schreibgeschützte Eigenschaft mit Initializer zu ändern, müssen Sie nicht nur `>` from `=>` entfernen, sondern auch `{ get; }` .

Das andere Symbol (`=>` anstelle von `=`) in `Users3` führt dazu, dass bei jedem Zugriff auf die Eigenschaft eine neue Instanz des `HashSet<UserDto>` die zwar das gültige C # (aus der Sicht des Compilers) wahrscheinlich nicht das gewünschte Verhalten darstellt Wird für ein Sammlungsmitglied verwendet.

Der obige Code entspricht:

```
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>(); } }
}
```

Index-Initialisierer

Indexinitialisierer ermöglichen das gleichzeitige Erstellen und Initialisieren von Objekten mit Indizes.

Dies macht das Initialisieren von Wörterbüchern sehr einfach:

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

Jedes Objekt mit einem indizierten Getter oder Setter kann mit dieser Syntax verwendet werden:

```
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();
    }
}
```

Ausgabe:

Index: foo, Wert: 34
Index: Bar, Wert: 42

Demo anzeigen

Wenn die Klasse über mehrere Indexer verfügt, können sie alle in einer einzigen Anweisungsgruppe zugewiesen werden:

```
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"
        };
    }
}
```

Ausgabe:

> Index: foo, Wert: 34
> Index: Bar, Wert: 42
> Index: 10, Wert: Zehn
> Index: 42, Wert: Sinn des Lebens

Es sollte beachtet werden, dass der Indexer- `set` Accessor sich möglicherweise anders verhält als eine `Add` Methode (die in Collection-Initialisierern verwendet wird).

Zum Beispiel:

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one
```

gegen:

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

String-Interpolation

Durch die String-Interpolation kann der Entwickler `variables` und Text zu einem String kombinieren.

Basisbeispiel

Es werden zwei `int` Variablen erstellt: `foo` und `bar` .

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

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

Console.WriteLine(resultString);
```

Ausgabe :

> Das Foo ist 34 und die Bar ist 42.

Demo anzeigen

Klammern innerhalb von Strings können wie folgt verwendet werden:

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

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

Dies erzeugt die folgende Ausgabe:

> Das Foo ist {Foo} und die Bar ist {Bar}.

Verwenden der Interpolation mit wörtlichen String-Literalen

Die Verwendung von `@` vor dem String bewirkt, dass der String wörtlich interpretiert wird. So bleiben beispielsweise Unicode-Zeichen oder Zeilenumbrüche genauso wie sie eingegeben wurden. Dies wirkt sich jedoch nicht auf die Ausdrücke in einer interpolierten Zeichenfolge aus, wie im folgenden Beispiel gezeigt:

```
Console.WriteLine($@"In case it wasn't clear:
\u00B9
The foo
is {foo},
and the bar
is {bar}.");
```

Ausgabe:

> Falls es nicht klar war:
> \ u00B9
> Das foo
> ist 34,
> und die Bar
> ist 42.

Demo anzeigen

Ausdrücke

Bei der String-Interpolation können auch *Ausdrücke* in geschweiften Klammern `{}` ausgewertet werden. Das Ergebnis wird an der entsprechenden Stelle in der Zeichenfolge eingefügt. Um beispielsweise das Maximum von `foo` und `bar` zu berechnen und einzufügen, verwenden Sie `Math.Max` innerhalb der geschweiften Klammern:

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

Ausgabe:

> Und der größere ist: 42

Hinweis: Alle führenden oder nachgestellten Leerzeichen (einschließlich Leerzeichen, Tabulatorzeichen und CRLF / Zeilenumbruch) zwischen der geschweiften Klammer und dem Ausdruck werden vollständig ignoriert und nicht in die Ausgabe aufgenommen

Demo anzeigen

Als weiteres Beispiel können Variablen als Währung formatiert werden:

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

Ausgabe:

> Foo als Währung mit 4 Dezimalstellen formatiert: 34,0000 USD

Demo anzeigen

Oder sie können als Datumsangaben formatiert werden:

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

Ausgabe:

> Heute ist: Montag, 20. Juli - 2015

Demo anzeigen

Anweisungen mit einem bedingten (ternären) Operator können auch innerhalb der Interpolation ausgewertet werden. Diese müssen jedoch in Klammern eingeschlossen werden, da der Doppelpunkt andernfalls verwendet wird, um die Formatierung wie oben gezeigt anzuzeigen:

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

Ausgabe:

> Bar ist größer als Foo!

Demo anzeigen

Bedingte Ausdrücke und Formatbezeichner können gemischt werden:

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

Ausgabe:

> Umgebung: 32-Bit-Prozess

Escape-Sequenzen

Die umgekehrten Schrägstriche (`\`) und Anführungszeichen (`"`) werden durchgängig für interpolierte Zeichenfolgen wie für nicht interpolierte Zeichenfolgen verwendet.

```
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 \");
```

Ausgabe:

> Foo ist 34. In einer nicht-wortgetreuen Zeichenfolge müssen wir "und \ mit Backslashes" abbrechen.
> Foo ist 34. In einer wörtlichen Zeichenfolge müssen wir mit einem zusätzlichen Anführungszeichen ""

Um eine geschweifte Klammer `{` oder `}` in eine interpolierte Zeichenfolge aufzunehmen, verwenden Sie zwei geschweifte Klammern `{{` oder `}}` :

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

Ausgabe:

> {foo} ist: 34

Demo anzeigen

FormattableString-Typ

Der Typ eines `$"..."` Zeichenfolgeninterpolationsausdrucks ist nicht immer eine einfache Zeichenfolge. Der Compiler entscheidet je nach Kontext, welchen Typ er zuweisen soll:

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

Dies ist auch die Reihenfolge der Typeinstellung, wenn der Compiler auswählen muss, welche überladene Methode aufgerufen wird.

Ein neuer Typ , `System.FormattableString` , repräsentiert eine zusammengesetzte `System.FormattableString` zusammen mit den zu formatierenden Argumenten. Verwenden Sie diese Option, um Anwendungen zu schreiben, die die Interpolationsargumente speziell behandeln:

```
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();

    // ...
}
```

Rufen Sie die obige Methode auf mit:

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

Zum Beispiel könnte man sich entscheiden, die Leistungskosten für die Formatierung der Zeichenfolge nicht zu tragen, wenn die Protokollierungsstufe das Protokollelement bereits herausfiltert.

Implizite Konvertierungen

Es gibt implizite Typkonvertierungen aus einer interpolierten Zeichenfolge:

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

Sie können auch eine `IFormattable` Variable `IFormattable` , mit der Sie die Zeichenfolge mit einem unveränderlichen Kontext konvertieren können:

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

Aktuelle und invariante Kulturmethoden

Wenn die Codeanalyse aktiviert ist, wird bei allen interpolierten Zeichenfolgen die Warnung CA1305 (`IFormatProvider`) `IFormatProvider` . Eine statische Methode kann verwendet werden, um die aktuelle Kultur anzuwenden.

```
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);
```

```
    }
}
```

Um eine korrekte Zeichenfolge für die aktuelle Kultur zu erstellen, verwenden Sie einfach den Ausdruck:

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

Hinweis : `Current` und `Invariant` können nicht als Erweiterungsmethoden erstellt werden, da der Compiler dem *interpolierten Zeichenfolgenausdruck* standardmäßig den Typ `String` zuweist, wodurch der folgende Code nicht kompiliert werden kann:

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

`FormattableString` Klasse enthält bereits die `Invariant()` -Methode. Die einfachste Möglichkeit, zur invarianten Kultur zu wechseln, ist die `using static` :

```
using static System.FormattableString;


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

Hinter den Kulissen

Interpolierte Zeichenfolgen sind nur ein syntaktischer Zucker für `String.Format()` . Der Compiler (Roslyn) verwandelt es hinter den Kulissen in ein `String.Format` :

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

Das obige wird in etwa so konvertiert:

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

String Interpolation und Linq

In Linq-Anweisungen können interpolierte Zeichenfolgen verwendet werden, um die Lesbarkeit weiter zu verbessern.

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

Kann umgeschrieben werden als:

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

Wiederverwendbare interpolierte Zeichenketten

Mit `string.Format` können Sie wiederverwendbare `string.Format` erstellen:

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

// ...

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

Interpolierte Zeichenfolgen werden jedoch nicht mit Platzhaltern kompiliert, die sich auf nicht vorhandene Variablen beziehen. Folgendes wird nicht kompiliert:

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

Erstellen `Func<>` stattdessen eine `Func<>` die Variablen verwendet und einen `String` zurückgibt:

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

// ...

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

String Interpolation und Lokalisierung

Wenn Sie Ihre Anwendung lokalisieren, fragen Sie sich möglicherweise, ob Sie die String-Interpolation zusammen mit der Lokalisierung verwenden können. Tatsächlich wäre es schön, die Möglichkeit , in Ressource zu speichern, zu müssen Dateien `String` s wie:

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

statt der viel weniger lesbaren:

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

`String` Interpolationsprozess findet *zur Kompilierungszeit statt* , im Gegensatz zur Formatierungszeichenfolge mit `string.Format` die *zur Laufzeit erfolgt* . Ausdrücke in einer

interpolierten Zeichenfolge müssen auf Namen im aktuellen Kontext verweisen und müssen in Ressourcendateien gespeichert werden. Das bedeutet, wenn Sie die Lokalisierung verwenden möchten, müssen Sie Folgendes tun:

```
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);
```

Wenn die Ressourcenzeichenfolgen für die oben verwendeten Sprachen korrekt in den einzelnen Ressourcendateien gespeichert sind, sollten Sie die folgende Ausgabe erhalten:

> Bonjour, mon nom est John
> Hallo, mein Name ist John
> Hallo mein Name ist John

Beachten Sie, dass dies bedeutet, dass der Name in jeder Sprache der lokalisierten Zeichenfolge folgt. Ist dies nicht der Fall, müssen Sie den Ressourcenzeichenfolgen Platzhalter hinzufügen und die obige Funktion ändern oder die Kulturinformationen in der Funktion abfragen und eine switch case-Anweisung angeben, die die verschiedenen Fälle enthält. Weitere Informationen zu Ressourcendateien finden Sie unter So verwenden Sie die Lokalisierung in C # .

Es empfiehlt sich, eine Standard-Ausweichsprache zu verwenden, die die meisten Leute verstehen, falls keine Übersetzung verfügbar ist. Ich schlage vor, Englisch als Standard-Ausweichsprache zu verwenden.

Rekursive Interpolation

Obwohl es nicht sehr nützlich ist, darf eine interpolierte `string` rekursiv in den geschweiften

Klammern eines anderen verwendet werden:

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

Ausgabe:

> String hat 27 Zeichen:
>
> Meine Klasse heißt MyClass.

Erwarte den Fang und endlich

Es ist möglich , die Verwendung `await` Ausdruck anzuwenden erwarten Betreiber zu Aufgaben oder Aufgaben (Of TResult) im `catch` und `finally` Blöcken in C # 6.

Es war nicht möglich , die verwenden `await` Ausdruck in dem `catch` und `finally` Blöcke in früheren Versionen aufgrund Compiler Einschränkungen. C # 6 macht viel einfacher async Aufgaben warten durch die damit `await` Ausdruck.

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

In C # 5 war es erforderlich, einen `bool` oder eine `Exception` außerhalb des Try `bool` deklarieren, um asynchrone Operationen auszuführen. Diese Methode wird im folgenden Beispiel gezeigt:

```
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 Ausbreitung

Die `?.` Operator und Operator `?[...]` werden als nullbedingter Operator bezeichnet . Es wird manchmal auch mit anderen Namen wie dem sicheren Navigationsoperator bezeichnet .

Das ist nützlich, denn wenn `.` Der Operator (member accessor) wird auf einen Ausdruck angewendet, der den `NullReferenceException` `null` `NullReferenceException` . Das Programm `NullReferenceException` eine `NullReferenceException` . Wenn der Entwickler stattdessen das `?.` Verwendet `?.` (nullbedingter) Operator: Der Ausdruck wird zu null ausgewertet, anstatt eine Ausnahme auszulösen.

Beachten Sie, dass wenn `?.` Operator wird verwendet und der Ausdruck ist nicht null, `?.` und `.` sind gleichwertig.

Grundlagen

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

Demo anzeigen

Wenn das `classroom` keinen Lehrer hat, gibt `GetTeacher()` möglicherweise `null` . Wenn der `NullReferenceException` `null` und auf die `Name` Eigenschaft zugegriffen wird, wird eine `NullReferenceException` ausgelöst.

Wenn wir diese Aussage ändern, um das `?.` Syntax, das Ergebnis des gesamten Ausdrucks ist `null` :

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

Demo anzeigen

Wenn `classroom` auch `null` kann, können Sie diese Anweisung auch folgendermaßen schreiben:

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

Demo anzeigen

Dies ist ein Beispiel für einen Kurzschluss: Wenn eine bedingte Zugriffsoperation, die den nullbedingten Operator verwendet, zu Null ausgewertet wird, wird der gesamte Ausdruck sofort zu Null ausgewertet, ohne den Rest der Kette zu verarbeiten.

Wenn das Terminal-Member eines Ausdrucks, der den Operator mit dem Wert null enthält, einen `Nullable<T>` , wird der Ausdruck zu einem `Nullable<T>` dieses Typs ausgewertet und kann daher nicht als direkter Ersatz für den Ausdruck ohne `?.` Verwendet werden `?.` .

```
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
```

Verwendung mit dem Null-Koaleszenz-Operator (??)

Sie können den nullbedingten Operator mit dem nullkoaleszierenden Operator (`??`) kombinieren, um einen Standardwert zurückzugeben, wenn der Ausdruck in `null` . Verwenden Sie unser Beispiel oben:

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

Verwenden Sie mit Indexern

Der nullbedingte Operator kann mit Indexern verwendet werden :

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

Im obigen Beispiel:

- Der erste `?.` stellt sicher, dass das `classroom` nicht `null` .

- Die zweite `?` stellt sicher, dass die gesamte `Students` Sammlung nicht `null` .
- Der dritte `?`. Danach stellt der Indexer sicher, dass der `[0]` Indexer kein `null` . Es ist zu beachten, dass dieser Vorgang immer **noch** eine `IndexOutOfRangeException` .

Verwenden Sie mit leeren Funktionen

Der bedingungslose Operator kann auch mit `void` Funktionen verwendet werden. In diesem Fall wird die Anweisung jedoch nicht auf `null` ausgewertet. Es wird nur eine `NullReferenceException` verhindert.

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

Verwendung mit Ereignisaufruf

Nehmen Sie die folgende Ereignisdefinition an:

```
private event EventArgs OnCompleted;
```

Wenn Sie ein Ereignis aufrufen, ist es üblich, zu überprüfen, ob das Ereignis `null` wenn keine Abonnenten vorhanden sind:

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

Da der nullbedingte Operator eingeführt wurde, kann der Aufruf auf eine einzige Zeile reduziert werden:

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

Einschränkungen

Ein nullbedingter Operator erzeugt einen r-Wert, nicht einen l-Wert. Das heißt, er kann nicht für die Zuweisung von Eigenschaften, für die Ereignisabonnementierung usw. verwendet werden.

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

Gotchas

Beachten Sie, dass:

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

ist **nicht** das Gleiche wie:

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

denn der erstere entspricht:

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

und letztere entspricht:

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

Trotz des ternären Operators `?:` Wird hier der Unterschied zwischen zwei Fällen erläutert. Diese Operatoren sind nicht gleichwertig. Dies kann mit dem folgenden Beispiel leicht demonstriert werden:

```
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;
        }
    }
}
```

Welche Ausgänge:

> Null Ausbreitung
> ich wurde gelesen
> 0
> Ternär

ich wurde gelesen
ich wurde gelesen
0

Demo anzeigen

Um mehrere Aufrufe zu vermeiden, wäre das Äquivalent folgendes:

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

Dieser Unterschied erklärt etwas, warum der Null-Ausbreitungsoperator in Ausdrucksbäumen noch nicht unterstützt wird .

Verwenden Sie statischen Typ

Die `using static [Namespace.Type]` Anweisung `using static [Namespace.Type]` ermöglicht das Importieren statischer Member von Typen und Aufzählungswerten. Erweiterungsmethoden werden als Erweiterungsmethoden (nur von einem Typ) importiert, nicht in den Bereich der obersten Ebene.

6,0

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

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

Live-Demo-Geige

6,0

```
using System;

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

Verbesserte Überlastauflösung

Der folgende Ausschnitt zeigt ein Beispiel für das Übergeben einer Methodengruppe (im Gegensatz zu einem Lambda), wenn ein Delegat erwartet wird. Überlastauflösung löst dieses Problem jetzt auf, anstatt einen mehrdeutigen Überlastungsfehler aufgrund der Fähigkeit von **C # 6** zu verursachen, den Rückgabetyp der übergebenen Methode zu überprüfen.

```
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;
    }
}
```

Ergebnisse:

6,0

Ausgabe

> Überladung mit Func <int> aufgerufen

Demo anzeigen

5,0

Error

> Fehler CS0121: Der Aufruf ist zwischen den folgenden Methoden oder Eigenschaften mehrdeutig: 'Program.Overloaded (System.Action)' und 'Program.Overloaded (System.Func)'

C # 6 kann auch den folgenden exakten Übereinstimmungsfall für Lambda-Ausdrücke gut verarbeiten, der zu einem Fehler in **C # 5 geführt hätte** .

```
using System;

class Program
{
```

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

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

Kleinere Änderungen und Bugfixes

Klammern sind jetzt um benannte Parameter verboten. Folgendes wird in C # 5 kompiliert, jedoch nicht in C # 6

5,0

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

Operanden von `is` und `as` dürfen keine Methodengruppen mehr sein. Folgendes wird in C # 5 kompiliert, jedoch nicht in C # 6

5,0

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

> Der native Compiler erlaubte dies (obwohl eine Warnung `1.Any is string` und `IDisposable.Dispose is object` nicht einmal die Kompatibilität der Erweiterungsmethoden, wodurch verrückte Dinge wie `1.Any is string` oder `IDisposable.Dispose is object` .

In dieser Referenz finden Sie Aktualisierungen zu Änderungen.

Verwenden einer Erweiterungsmethode für die Initialisierung der Sammlung

Die Initialisierungssyntax für `IEnumerable` kann verwendet werden, wenn eine Klasse instanziiert wird, die `IEnumerable` implementiert und über eine Methode namens `Add` die einen einzelnen Parameter verwendet.

In früheren Versionen musste diese `Add` Methode eine **Instanzmethode** für die zu initialisierende Klasse sein. In C # 6 kann es auch eine Erweiterungsmethode sein.

```
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
    }
}
```

Dies wird ausgegeben:

Element hinzugefügt mit Instanz hinzufügen: 1
Element hinzugefügt mit Instanz hinzufügen: 2
Element hinzugefügt mit Instanz hinzufügen: 3
Element hinzugefügt mit Erweiterungsmethode: 4
Element hinzugefügt mit Erweiterungsmethode: 5
Element hinzugefügt mit Erweiterungsmethode: 6

Deaktivieren Sie Warnungen-Verbesserungen

In C # 5.0 und früheren Versionen konnte der Entwickler Warnungen nur nach Nummer unterdrücken. Mit der Einführung der Roslyn Analyzer benötigt C # eine Möglichkeit, Warnungen, die von bestimmten Bibliotheken ausgegeben werden, zu deaktivieren. Mit C # 6.0 kann die Pragma-Direktive Warnungen anhand des Namens unterdrücken.

Vor:

```
#pragma warning disable 0501
```

C # 6.0:

```
#pragma warning disable CS0501
```

C # 6.0-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/24/c-sharp-6-0-funktionen

Kapitel 30: C # 7.0-Funktionen

Einführung

C # 7.0 ist die siebte Version von C #. Diese Version enthält einige neue Funktionen: Sprachunterstützung für Tupel, lokale Funktionen, `out var` Deklarationen, Zifferntrennzeichen, Binärliterale, Musterabgleich, Wurfausdrücke, `ref return` und `ref local` Liste der `ref local` und erweiterten Ausdruckselemente.

Offizielle Referenz: Was ist neu in C # 7?

Examples

aus var deklaration

Ein übliches Muster in C # ist die Verwendung von `bool TryParse(object input, out object value)`, um Objekte sicher zu analysieren.

Die `out var` Deklaration ist eine einfache Funktion zur Verbesserung der Lesbarkeit. Es ermöglicht, dass eine Variable gleichzeitig als deklarierter Parameter deklariert wird.

Eine auf diese Weise deklarierte Variable bezieht sich auf den Rest des Körpers an der Stelle, an der sie deklariert wird.

Beispiel

Bei Verwendung von `TryParse` vor C # 7.0 müssen Sie vor dem Aufruf der Funktion eine Variable deklarieren, die den Wert erhalten soll:

7,0

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

Foo(value); // ok
```

In C # 7.0 können Sie die Deklaration der an den `out` Parameter übergebenen Variablen einschließen, sodass keine separate Variablendeklaration erforderlich ist:

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
```

Wenn einige der Parameter, die eine Funktion in `out` zurückgibt `out` nicht benötigt werden, können Sie den *Verwerfungsoperator* _ .

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

Eine `out var` Deklaration kann mit jeder vorhandenen Funktion verwendet werden, für die bereits `out` Parameter vorhanden sind. Die Funktionsdeklarationssyntax bleibt gleich und es sind keine zusätzlichen Anforderungen erforderlich, um die Funktion mit einer `out var` Deklaration kompatibel zu machen. Diese Funktion ist einfach syntaktischer Zucker.

Eine andere Funktion der `out var` Deklaration ist, dass sie mit anonymen Typen verwendet werden kann.

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);
}
```

In diesem Code erstellen wir ein `Dictionary` mit einem `int` Schlüssel und einem Array mit einem anonymen Typwert. In der vorherigen Version von C # war es nicht möglich, die `TryGetValue` Methode hier zu verwenden, da Sie die `out` Variable (die vom anonymen Typ ist!) `TryGetValue` . Mit `out var` wir jedoch den Typ der `out` Variablen nicht explizit angeben.

Einschränkungen

Beachten Sie, dass out-Var-Deklarationen in LINQ-Abfragen nur von begrenztem Nutzen sind, da Ausdrücke als Ausdrucks-Lambda-Körper interpretiert werden. Der Umfang der eingeführten Variablen ist daher auf diese Lambdas beschränkt. Der folgende Code funktioniert beispielsweise nicht:

```
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
```

Verweise

- Ursprünglicher Var-Deklarationsvorschlag auf GitHub

Binäre Literale

Das **0b-** Präfix kann verwendet werden, um binäre Literale darzustellen.

Binäre Literale ermöglichen das Erstellen von Zahlen aus Nullen und Einsen. Dadurch wird deutlich, welche Bits in der binären Darstellung einer Zahl gesetzt sind. Dies kann nützlich sein, wenn Sie mit binären Flags arbeiten.

Im Folgenden sind gleichwertige Möglichkeiten zum Angeben eines `int` mit dem Wert `34` (= $2^5 + 2^1$) angegeben:

```
// 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
```

Flags Aufzählungen

Das Festlegen von Flagwerten für eine `enum` war zuvor nur mit einer der drei Methoden in diesem Beispiel möglich:

```
[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
}
```

Bei binären Literalen ist es offensichtlicher, welche Bits gesetzt sind, und deren Verwendung erfordert kein Verständnis der Hexadezimalzahlen und der bitweisen Arithmetik:

```
[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
}
```

Zifferntrennzeichen

Der Unterstrich _ kann als Zifferntrennzeichen verwendet werden. Die Möglichkeit, Ziffern in großen numerischen Literalen zu gruppieren, hat einen erheblichen Einfluss auf die Lesbarkeit.

Der Unterstrich kann an einer beliebigen Stelle in einem numerischen Literal vorkommen, außer wie unten angegeben. Unterschiedliche Gruppierungen können in verschiedenen Szenarien oder mit unterschiedlichen numerischen Grundlagen sinnvoll sein.

Jede Ziffernfolge kann durch einen oder mehrere Unterstriche getrennt werden. Das _ ist sowohl in Dezimalzahlen als auch in Exponenten zulässig. Die Trennzeichen haben keine semantischen Auswirkungen - sie werden einfach ignoriert.

```
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;
```

Wenn das _ Trennzeichen nicht verwendet werden darf:

- am Anfang des Wertes (`_121`)
- am Ende des Wertes (`121_` oder `121.05_`)
- neben der Dezimalzahl (`10_.0`)
- neben dem Exponentenzeichen (`1.1e_1`)
- neben dem Typbezeichner (`10_f`)
- unmittelbar nach dem `0x` oder `0b` in Binär- und Hexadezimal-Literalen (kann geändert werden, um beispielsweise 0b_1001_1000 zuzulassen)

Sprachunterstützung für Tupel

Grundlagen

Ein **Tupel** ist eine geordnete, endliche Liste von Elementen. Tupel werden üblicherweise in der Programmierung als Mittel verwendet, um mit einer einzelnen Entität zusammenzuarbeiten, anstatt einzeln mit jedem der Elemente des Tupels zu arbeiten, und um einzelne Zeilen (dh "Datensätze") in einer relationalen Datenbank darzustellen.

In C # 7.0 können Methoden mehrere Rückgabewerte enthalten. Hinter den Kulissen verwendet der Compiler die neue ValueTuple- Struktur.

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

Randbemerkung: für das in Visual Studio 2017 zu arbeiten, müssen Sie die bekommen `System.ValueTuple` Paket.

Wenn ein Ergebnis für eine Methode, die ein Tupel zurückgibt, einer einzelnen Variablen zugewiesen wird, können Sie auf die Member über ihre definierten Namen in der Methodensignatur zugreifen:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Tuple Dekonstruktion

Tupel Dekonstruktion trennt ein Tupel in seine Teile.

Wenn Sie beispielsweise `GetTallies` und den Rückgabewert zwei separaten Variablen `GetTallies` , dekonstruieren Sie das Tupel in diese beiden Variablen:

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

`var` funktioniert auch:

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

Sie können auch eine kürzere Syntax verwenden, wobei `var` außerhalb von `()` :

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

Sie können auch in vorhandene Variablen zerlegen:

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

Das Tauschen ist jetzt viel einfacher (keine temporäre Variable erforderlich):

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

Interessanterweise kann jedes Objekt dekonstruiert werden, indem eine `Deconstruct` Methode in der Klasse definiert wird:

```
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;
```

In diesem Fall ruft die `person (localFirstName, localLastName) = person Deconstruct` für die `person` .

Dekonstruktion kann sogar in einer Erweiterungsmethode definiert werden. Dies entspricht dem oben genannten:

```
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;
```

Ein alternativer Ansatz für die `Person` Klasse besteht darin, den `Name` selbst als `Tuple` zu definieren. Folgendes berücksichtigen:

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

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

Dann können Sie eine Person so instanziieren (wo wir ein Tupel als Argument nehmen können):

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

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

Tupel-Initialisierung

Sie können Tupel auch beliebig im Code erstellen:

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

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

Beim Erstellen eines Tupels können Sie den Mitgliedern des Tupels Ad-hoc-Objektnamen zuweisen:

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

Inferenz eingeben

Mehrere Tupel, die mit derselben Signatur (übereinstimmende Typen und Anzahl) definiert wurden, werden als übereinstimmende Typen abgeleitet. Zum Beispiel:

```
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` können zurückgegeben werden, da die Deklaration der Variablen " `stats` und die Rückgabesignatur der Methode übereinstimmen.

Reflexions- und Tupelfeldnamen

Mitgliedsnamen sind zur Laufzeit nicht vorhanden. In Reflection werden Tupel mit derselben Anzahl und Elementtypen als identisch betrachtet, auch wenn die Mitgliedsnamen nicht übereinstimmen. Das Konvertieren eines Tupels in ein `object` und dann in ein Tupel mit denselben Mitgliedstypen, aber unterschiedlichen Namen, führt ebenfalls nicht zu einer Ausnahme.

Während die ValueTuple-Klasse selbst keine Informationen für Mitgliedsnamen speichert, sind die Informationen durch Reflektion in einem TupleElementNamesAttribute verfügbar. Dieses Attribut wird nicht auf das Tupel selbst angewendet, sondern auf Methodenparameter, Rückgabewerte, Eigenschaften und Felder. Dadurch können die Namen der Tupel-Elemente in allen Assemblys beibehalten werden. Wenn also eine Methode (String-Name, Int-Anzahl) zurückgibt, stehen Name und Anzahl der Namen den Aufrufern der Methode in einer anderen Assembly zur Verfügung, da der Rückgabewert mit TupleElementNameAttribute mit den Werten markiert wird "name" und "count".

Verwendung mit Generika und `async`

Die neuen Tuple-Features (mit dem zugrunde liegenden `ValueTuple` Typ) unterstützen Generics vollständig und können als generische Typparameter verwendet werden. Dadurch ist es möglich, sie mit dem `async` / `await` Muster zu verwenden:

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

    return (fooBar, num);
}
```

Verwenden Sie mit Sammlungen

Es kann von Vorteil sein, eine Sammlung von Tupeln in einem Szenario (als Beispiel) zu haben, in dem Sie versuchen, ein übereinstimmendes Tupel basierend auf Bedingungen zu finden, um eine Verzweigung des Codes zu vermeiden.

Beispiel:

```
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;
}
```

Mit den neuen Tupeln können:

```
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;
}
```

Obwohl die Benennung im obigen Beispieltupel ziemlich allgemein ist, ermöglicht die Idee der relevanten Labels ein tieferes Verständnis dessen, was im Code versucht wird, wenn auf "item1", "item2" und "item3" verwiesen wird.

Unterschiede zwischen ValueTuple und Tuple

Der Hauptgrund für die Einführung von `ValueTuple` ist die Leistung.

Modellname	`ValueTuple`	`Tuple`
Klasse oder Struktur	`struct`	`class`
Mutabilität (Werte nach Erstellung ändern)	veränderlich	unveränderlich
Benennen von Mitgliedern und Unterstützung anderer Sprachen	Ja	nein (TBD)

Verweise

- Ursprünglicher Tuples-Sprachfeature-Vorschlag auf GitHub
- Eine lauffähige VS 15-Lösung für C # 7.0-Funktionen
- NuGet Tuple-Paket

Lokale Funktionen

Lokale Funktionen werden innerhalb einer Methode definiert und stehen außerhalb nicht zur Verfügung. Sie haben Zugriff auf alle lokalen Variablen und unterstützen Iteratoren, `async` / `await`

und Lambda-Syntax. Auf diese Weise können für eine Funktion spezifische Wiederholungen funktionalisiert werden, ohne die Klasse zu überfordern. Als Nebeneffekt verbessert dies die Intellisense-Vorschlagsleistung.

Beispiel

```
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;
    }
}
```

Lokale Funktionen vereinfachen den Code für LINQ-Operatoren erheblich. In diesem Fall müssen Sie normalerweise Argumentprüfungen von der tatsächlichen Logik trennen, um Argumentprüfungen sofort durchzuführen, und nicht verzögert, bis die Iteration gestartet wird.

Beispiel

```
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;
    }
}
```

Lokale Funktionen unterstützen auch die `async` und `await` Schlüsselwörter.

Beispiel

```
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);
            }
        }
    }
}
```

Eine wichtige Sache, die Sie möglicherweise bemerkt haben, ist, dass lokale Funktionen unter der `return` Anweisung definiert werden können. Sie müssen **nicht** darüber definiert werden. Außerdem folgen lokale Funktionen normalerweise der Namenskonvention "lowerCamelCase", um sich leichter von Klassenbereichsfunktionen zu unterscheiden.

Musterabgleich

Pattern-Matching-Erweiterungen für C # ermöglichen viele Vorteile des Pattern-Matchings aus funktionalen Sprachen, jedoch auf eine Weise, die sich nahtlos in das Gefühl der zugrunde liegenden Sprache einfügt

Ausdruck `switch`

Pattern Matching erweitert die `switch` Anweisung um Typen zu aktivieren:

```
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` Ausdruck

Der Musterabgleich erweitert den `is` Operator, um nach einem Typ zu suchen und gleichzeitig eine neue Variable zu deklarieren.

Beispiel

7,0

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

kann umgeschrieben werden als:

7,0

```
if(o is string s)
{
```

```
    //Do something with s
};
```

Beachten Sie auch, dass der Gültigkeitsbereich der Mustervariablen `s` außerhalb des `if` Blocks liegt und das Ende des umschließenden Gültigkeitsbereichs erreicht. Beispiel:

```
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 return und ref local

Ref Returns und Ref-Locals sind nützlich, um Verweise auf Speicherblöcke zu bearbeiten und zurückzugeben, anstatt Speicher zu kopieren, ohne auf unsichere Zeiger zurückzugreifen.

Ref Return

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

Damit können Sie zwei Werte als Referenz übergeben, wobei einer von ihnen basierend auf einer Bedingung zurückgegeben wird:

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

Ref Local

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

```
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Unsichere Ref-Operationen

In `System.Runtime.CompilerServices.Unsafe` ein Satz unsicherer Operationen definiert, mit denen Sie die `ref` Werte im Grunde wie Zeiger bearbeiten können.

Zum Beispiel, interpretieren Sie eine Speicheradresse (`ref`) als einen anderen Typ neu:

```
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]);
```

`BitConverter.IsLittleEndian` Sie dabei jedoch auf die Endianness , überprüfen `BitConverter.IsLittleEndian` z. B. `BitConverter.IsLittleEndian` und behandeln Sie es entsprechend.

Oder durchlaufen Sie ein Array auf unsichere Weise:

```
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);
```

Oder das ähnliche `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);
```

Außerdem kann geprüft werden, ob zwei `ref` dieselbe Adresse haben, dh dieselbe Adresse:

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

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

Links

Roslyn Github Ausgabe

System.Runtime.CompilerServices.Unsafe für github

Ausdrücke werfen

C # 7.0 erlaubt das Werfen als Ausdruck an bestimmten Stellen:

```
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();
}
```

Wenn Sie vor C # 7.0 eine Ausnahme aus einem Ausdruckskörper auslösen möchten, müssen Sie Folgendes tun:

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

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

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

Oder

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

In C # 7.0 ist das Obige nun vereinfacht zu:

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

Liste der erweiterten Mitglieder mit Ausdruck

C # 7.0 fügt Accessoren, Konstruktoren und Finalisierer der Liste der Dinge hinzu, die Ausdruckskörper haben können:

```
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
    }
}
```

Siehe auch den Abschnitt out var-Deklaration für den Discard-Operator.

ValueTask

`Task<T>` ist eine **Klasse** und verursacht den unnötigen Overhead ihrer Zuordnung, wenn das Ergebnis sofort verfügbar ist.

`ValueTask<T>` ist eine **Struktur** und wurde eingeführt, um die Zuweisung eines `Task` Objekts zu verhindern, falls das Ergebnis der **asynchronen** Operation zum Zeitpunkt des **Warten** bereits verfügbar ist.

`ValueTask<T>` bietet also zwei Vorteile:

1. Leistungssteigerung

Hier ist ein `Task<T>` Beispiel:

- Erfordert eine Heapzuordnung
- Nimmt 120s mit JIT

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

Hier ist das analoge `ValueTask<T>` Beispiel:

- Keine Heapzuordnung , wenn das Ergebnis synchron bekannt ist (was es wegen des in diesem Fall nicht ist `Task.Delay` , ist aber oft in vielen realen `async` / `await` Szenarien)
- Nimmt 65s mit JIT

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

2. Erhöhte Flexibilität bei der Implementierung

Implementierungen einer asynchronen Schnittstelle, die synchron sein wollen, würden sonst gezwungen sein, entweder `Task.Run` oder `Task.FromResult` (was zu dem oben beschriebenen Performance- `Task.FromResult` führt). Daher besteht ein gewisser Druck gegenüber synchronen Implementierungen.

Mit `ValueTask<T>` können Implementierungen jedoch mehr frei wählen, ob sie synchron oder asynchron sind, ohne dass Anrufer beeinträchtigt werden.

Hier ist zum Beispiel eine Schnittstelle mit einer asynchronen Methode:

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

... und so könnte man diese Methode nennen:

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

Mit `ValueTask` der obige Code **entweder mit synchronen oder asynchronen Implementierungen** :

Synchrone Implementierung:

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

Asynchrone Implementierung

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

Anmerkungen

Obwohl `ValueTask` wurde, dass `ValueTask` Struktur zu C # 7.0 hinzugefügt wurde, wurde sie vorerst als andere Bibliothek beibehalten. Das ValueTask <T> `System.Threading.Tasks.Extensions` Paket kann von der Nuget Gallery heruntergeladen werden

C # 7.0-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/1936/c-sharp-7-0-funktionen

Kapitel 31: C # Authentifizierungshandler

Examples

Authentifizierungshandler

```
 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;
            }
        }
    }
```

C # Authentifizierungshandler online lesen: https://riptutorial.com/de/csharp/topic/5430/c-sharp-authentifizierungshandler

Kapitel 32: C # Skript

Examples

Einfache Code-Auswertung

Sie können jeden gültigen C # -Code auswerten:

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

Wenn type nicht angegeben ist, ist das Ergebnis `object` :

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

C # Skript online lesen: https://riptutorial.com/de/csharp/topic/3780/c-sharp-skript

Kapitel 33: Caching

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
```

Caching online lesen: https://riptutorial.com/de/csharp/topic/4383/caching

Kapitel 34: Casting

Bemerkungen

Casting ist nicht dasselbe wie das *Konvertieren* . Es ist möglich, den Zeichenfolgewert `"-1"` in einen ganzzahligen Wert (`-1`) umzuwandeln. Dies muss jedoch über Bibliotheksmethoden wie `Convert.ToInt32()` oder `Int32.Parse()` . Es kann nicht direkt mit der Casting-Syntax ausgeführt werden.

Examples

Umwandeln eines Objekts in einen Basistyp

Angesichts der folgenden Definitionen:

```
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";
    }
}
```

Umwandeln eines Objekts in ein Basistyp-Beispiel:

```
    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
```

Explizite Besetzung

Wenn Sie wissen, dass ein Wert von einem bestimmten Typ ist, können Sie ihn explizit in diesen Typ umwandeln, um ihn in einem Kontext zu verwenden, in dem dieser Typ benötigt wird.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

Wenn Sie versuchen, den `value` direkt an `Math.Abs()` , erhalten Sie eine Ausnahmebedingung zur Kompilierungszeit, da `Math.Abs()` keine Überladung hat, die ein `object` als Parameter übernimmt.

Wenn der `value` nicht in ein `int` , würde die zweite Zeile in diesem Beispiel eine `InvalidCastException`

Sicheres explizites Casting (`as`-Operator)

Wenn Sie nicht sicher sind, ob ein Wert von dem Typ ist, von dem Sie glauben, dass Sie ihn mit dem Operator `as` sicher umsetzen können Wenn der Wert nicht von diesem Typ ist, ist der resultierende Wert `null` .

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Beachten Sie, dass `null` keine Art haben, so dass das `as` Schlüsselwort wird sicher ergeben `null` , wenn Giessharzsystemen `null` Wert.

Implizite Besetzung

Ein Wert wird automatisch in den entsprechenden Typ umgewandelt, wenn der Compiler weiß, dass er immer in diesen Typ konvertiert werden kann.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

In diesem Beispiel mussten wir nicht die typische explizite Umwandlungssyntax verwenden, da der Compiler weiß, dass alle `int` in `object` s umgewandelt werden können. Tatsächlich könnten wir das Erstellen von Variablen vermeiden und `-1` direkt als Argument von `Console.WriteLine()` , das ein `object` erwartet.

```
Console.WriteLine(-1);
```

Kompatibilität ohne Abguss prüfen

Wenn Sie wissen möchten, ob der Typ eines Werts einen bestimmten Typ erweitert oder implementiert, Sie ihn jedoch nicht als diesen Typ umwandeln möchten, können Sie den Operator `is` .

```
if(value is int)
{
   Console.WriteLine(value + "is an int");
}
```

Explizite numerische Konvertierungen

Explizite Casting-Operatoren können verwendet werden, um Konvertierungen numerischer Typen durchzuführen, auch wenn sie sich nicht erweitern oder implementieren.

```
double value = -1.1;
int number = (int) value;
```

Beachten Sie, dass in Fällen, in denen der Zieltyp eine geringere Genauigkeit als der Originaltyp hat, die Genauigkeit verloren geht. Zum Beispiel wird `-1.1` als Doppelwert im obigen Beispiel zu `-1` als ganzzahliger Wert.

Außerdem sind numerische Konvertierungen von Kompilierzeittypen abhängig. Sie funktionieren daher nicht, wenn die numerischen Typen in Objekten "eingebettet" wurden.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

Konvertierungsoperatoren

In C # können Typen benutzerdefinierte *Konvertierungsoperatoren* definieren, mit denen Werte in explizite oder implizite Umwandlungen in andere Typen konvertiert werden können. Stellen Sie sich beispielsweise eine Klasse vor, die einen JavaScript-Ausdruck darstellen soll:

```
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 + ")");
    }
}
```

Wenn Sie eine JsExpression erstellen möchten, die einen Vergleich von zwei JavaScript-Werten

darstellt, können Sie Folgendes tun:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Wir können jedoch einige *explizite Konvertierungsoperatoren* zu `JsExpression` hinzufügen, um bei der *expliziten Umwandlung* eine einfache Konvertierung zu ermöglichen.

```
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)
```

Oder wir können diese Operatoren *implizit* ändern, um die Syntax viel einfacher zu machen.

```
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 Casting-Vorgänge

Angenommen, Sie haben Typen wie die folgenden:

```
interface IThing {  }
class Thing : IThing {  }
```

Mit LINQ können Sie eine Projektion erstellen, die den generischen Typ der Kompilierzeit eines `IEnumerable<>` über die `Enumerable.Cast<>()` und `Enumerable.OfType<>()` .

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

Wenn `things2` ausgewertet wird, versucht die `Cast<>()` Methode, alle Werte in `things` in `Thing` s zu

`things2` . Wenn ein Wert gefunden wird, der nicht umgewandelt werden kann, wird eine `InvalidCastException` ausgelöst.

Bei der `things3` führt die `OfType<>()` Methode dasselbe aus, mit der Ausnahme, dass ein Wert, der nicht umgewandelt werden kann, einfach weggelassen wird, anstatt eine Ausnahme `OfType<>()` .

Aufgrund des generischen Typs dieser Methoden können sie keine Konvertierungsoperatoren aufrufen oder numerische Konvertierungen durchführen.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

Sie können einfach einen Cast in einem `.Select()` als Workaround ausführen:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Casting online lesen: https://riptutorial.com/de/csharp/topic/2690/casting

Kapitel 35: CLSCompliantAttribute

Syntax

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

Parameter

Konstrukteur	Parameter
CLSCompliantAttribute (Boolean)	Initialisiert eine Instanz der CLSCompliantAttribute-Klasse mit einem booleschen Wert, der angibt, ob das angegebene Programmelement CLS-kompatibel ist.

Bemerkungen

Die Common Language Specification (CLS) ist eine Reihe von Basisregeln, die alle Sprachen, die auf die CLI abzielen (Sprache, die die Spezifikationen der Common Language Infrastructure bestätigt), bestätigen sollten, um mit anderen CLS-kompatiblen Sprachen zusammenzuarbeiten.

Liste der CLI-Sprachen

In den meisten Fällen sollten Sie Ihre Assembly als CLSCompliant kennzeichnen, wenn Sie Bibliotheken verteilen. Dieses Attribut garantiert Ihnen, dass Ihr Code von allen CLS-kompatiblen Sprachen verwendet werden kann. Das bedeutet, dass Ihr Code von jeder Sprache verwendet werden kann, die kompiliert und unter CLR (Common Language Runtime) ausgeführt werden kann.

Wenn Ihre Assembly mit `CLSCompliantAttribute` markiert `CLSCompliantAttribute` , überprüft der Compiler, ob Ihr Code gegen eine der CLS-Regeln verstößt, und gibt bei Bedarf eine **Warnung aus** .

Examples

Zugriffsmodifizierer, für den CLS-Regeln gelten

```
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;
        }

    }
}
```

Die Regeln für die CLS-Konformität gelten nur für öffentliche / geschützte Komponenten.

Verletzung der CLS-Regel: Vorzeichenlose Typen / 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;


    }
}
```

Verletzung der CLS-Regel: Gleiche Benennung

```
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 unterscheidet nicht zwischen Groß- und Kleinschreibung

Verletzung der CLS-Regel: Kennung _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

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

}
```

Sie können die Variable nicht mit _ starten.

Verletzung der CLS-Regel: Erbt von der Klasse CLSComplaint

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

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

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

}
```

CLSCompliantAttribute online lesen:
https://riptutorial.com/de/csharp/topic/7214/clscompliantattribute

Kapitel 36: Code-Verträge

Syntax

1. Vertrag.Anforderungen (Bedingung, Benutzermeldung)

   Vertrag.Anforderungen (Bedingung, Benutzermeldung)

   Vertrag.Ergebnis <T>

   Contract.Ensures ()

   Vertragsvarianten ()

Bemerkungen

.NET unterstützt die Idee von Design by Contract über die Klasse Contracts, die im Namespace System.Diagnostics gefunden und in .NET 4.0 eingeführt wurde. Die Code Contracts-API enthält Klassen für statische und Laufzeitüberprüfungen von Code und ermöglicht Ihnen das Definieren von Vorbedingungen, Nachbedingungen und Invarianten innerhalb einer Methode. Die Vorbedingungen legen die Bedingungen fest, die die Parameter erfüllen müssen, bevor eine Methode ausgeführt werden kann, Nachbedingungen, die nach Abschluss einer Methode überprüft werden, und die Invarianten definieren die Bedingungen, die sich während der Ausführung einer Methode nicht ändern.

Warum werden Code-Verträge benötigt?

Das Nachverfolgen von Problemen einer Anwendung bei laufender Anwendung ist eines der Hauptanliegen aller Entwickler und Administratoren. Das Tracking kann auf verschiedene Arten durchgeführt werden. Zum Beispiel -

- Sie können die Ablaufverfolgung für unsere Anwendung anwenden und die Details einer Anwendung abrufen, wenn die Anwendung ausgeführt wird
- Sie können den Ereignisprotokollierungsmechanismus verwenden, wenn Sie die Anwendung ausführen. Die Meldungen können mit der Ereignisanzeige eingesehen werden
- Sie können die Leistungsüberwachung nach einem bestimmten Zeitintervall anwenden und Live-Daten aus Ihrer Anwendung schreiben.

Code Contracts verwendet einen anderen Ansatz zum Verfolgen und Verwalten von Problemen in einer Anwendung. Anstatt alles, was von einem Methodenaufruf zurückgegeben wird, zu überprüfen, stellen Code Contracts mithilfe von Vorbedingungen, Nachbedingungen und Invarianten für Methoden sicher, dass alles, was in Ihre Methoden eingeht und diese verlassen, korrekt ist.

Examples

Voraussetzungen

```
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);
        }
    }
}
```

Nachbedingungen

```
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;
}
```

Invarianten

```
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);
        }
    }
}
```

Verträge auf der Schnittstelle definieren

```
[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;
        }
    }
}
```

Im obigen Code haben wir eine Schnittstelle namens `IValidation` mit einem Attribut `[ContractClass]` . Dieses Attribut nimmt die Adresse einer Klasse an, in der wir einen Vertrag für eine Schnittstelle implementiert haben. Die Klasse `ValidationContract` verwendet Eigenschaften, die in der Schnittstelle definiert sind, und prüft mithilfe von `Contract.Requires<T>` auf die Nullwerte. `T` ist eine Ausnahmeklasse.

Wir haben auch den get-Accessor mit einem Attribut `[Pure]` . Das pure-Attribut stellt sicher, dass die Methode oder eine Eigenschaft den `IValidation` einer Klasse, in der die `IValidation` Schnittstelle implementiert ist, nicht ändert.

Code-Verträge online lesen: https://riptutorial.com/de/csharp/topic/4241/code-vertrage

Kapitel 37: Collection-Initialisierer

Bemerkungen

Die einzige Voraussetzung für die Initialisierung eines Objekts mit diesem syntaktischen Zucker ist, dass der Typ `System.Collections.IEnumerable` und die `Add` Methode implementiert. Obwohl wir es ein Auflistungsinitialisierer nennen, wird das Objekt muss *nicht* eine Sammlung.

Examples

Collection-Initialisierer

Initialisieren Sie einen Auflistungstyp mit Werten:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Collection-Initialisierer sind syntaktischer Zucker für `Add()` Aufrufe. Der obige Code entspricht:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Beachten Sie, dass die Initialisierung mit einer temporären Variablen atomar erfolgt, um Race-Bedingungen zu vermeiden.

Für Typen, die in ihrer `Add()` Methode mehrere Parameter enthalten, schließen Sie die durch Kommas getrennten Argumente in geschweifte Klammern ein:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

Das ist äquivalent zu:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

C # 6 Index-Initialisierer

Beginnend mit C # 6 können Sammlungen mit Indexern initialisiert werden, indem der zuzuweisende Index in eckigen Klammern angegeben wird, gefolgt von einem Gleichheitszeichen und dem zuzuweisenden Wert.

Wörterbuch-Initialisierung

Ein Beispiel für diese Syntax mit einem Dictionary:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

Das ist äquivalent zu:

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

Die Syntax des Collection-Initialisierers vor C # 6 lautete:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Was würde entsprechen:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

Es gibt also einen signifikanten Unterschied in der Funktionalität, da die neue Syntax den *Indexer* des initialisierten Objekts verwendet, um Werte zuzuweisen, statt die `Add()` Methode zu verwenden. Dies bedeutet, dass für die neue Syntax nur ein öffentlich verfügbarer Indexer erforderlich ist und für jedes Objekt gilt, das über einen verfügt.

```
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
}
```

Dies würde ausgeben:

```
10 was assigned to index 0
20 was assigned to index 1
```

Auflistungsinitialisierer in benutzerdefinierten Klassen

Um eine Klassenunterstützungsauflistungsinitialisierer zu `IEnumerable` , muss sie die `IEnumerable` Schnittstelle implementieren und über mindestens eine `Add` Methode verfügen. Seit C # 6 kann jede Sammlung, die `IEnumerable` implementiert, um benutzerdefinierte `Add` Methoden erweitert werden, die Erweiterungsmethoden verwenden.

```
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());
}
```

Collection-Initialisierer mit Parameter-Arrays

Sie können normale Parameter und Parameter-Arrays mischen:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

Diese Syntax ist jetzt möglich:

```
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}
}
```

Sammlungsinitialisierer innerhalb des Objektinitialisierers verwenden

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` ist eine Eigenschaft der Sammlung. Wenn das `Tag` Objekt mit der Objektinitialisiersyntax erstellt wird, können `Synonyms` auch mit der Erfassungsinitialisiersyntax initialisiert werden:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

Die Auflistungseigenschaft kann nur gelesen werden und unterstützt weiterhin die Syntax der Auflistungsinitialisierer. Betrachten Sie dieses modifizierte Beispiel (die Eigenschaft " `Synonyms` jetzt einen privaten Setzer):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }

    public IList<string> Synonyms { get; private set; }
}
```

Ein neues `Tag` Objekt kann folgendermaßen erstellt werden:

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

Dies funktioniert, da Collection-Initialisierer nur Syntaxzuweisungen bei Aufrufen von `Add()` . Hier wird keine neue Liste erstellt. Der Compiler generiert lediglich Aufrufe von `Add()` für das vorhandene Objekt.

Collection-Initialisierer online lesen: https://riptutorial.com/de/csharp/topic/21/collection-initialisierer

Kapitel 38: Datei- und Stream-E / A

Einführung

Verwaltet Dateien.

Syntax

- `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)`

Parameter

Parameter	Einzelheiten
Pfad	Der Speicherort der Datei.
anhängen	Wenn die Datei vorhanden ist, fügt true dem Ende der Datei Daten hinzu (Anfügen), überschreibt die Datei die Datei.
Text	Text, der geschrieben oder gespeichert werden soll.
Inhalt	Eine Sammlung von zu schreibenden Strings.
Quelle	Der Speicherort der Datei, die Sie verwenden möchten.
dest	Der Speicherort, an den eine Datei gehen soll.

Bemerkungen

- Stellen Sie immer sicher, dass `Stream` Objekte geschlossen werden. Dies kann mit einem `using` Block wie oben gezeigt geschehen oder durch manuelles Aufrufen von `myStream.Close()` .

- Stellen Sie sicher, dass der aktuelle Benutzer über die erforderlichen Berechtigungen für den Pfad verfügt, den Sie zum Erstellen der Datei versuchen.
- Verbatim-Zeichenfolgen sollten verwendet werden, wenn eine `@"C:\MyFolder\MyFile.txt"` Schrägstrichen deklariert wird: `@"C:\MyFolder\MyFile.txt"`

Examples

Lesen aus einer Datei mithilfe der System.IO.File-Klasse

Sie können die Funktion **System.IO.File.ReadAllText** verwenden, um den gesamten Inhalt einer Datei in eine Zeichenfolge zu lesen.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

Mit der Funktion **System.IO.File.ReadAllLines** können Sie eine Datei auch als **Zeilenarray lesen** :

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Schreiben von Zeilen in eine Datei mithilfe der System.IO.StreamWriter-Klasse

Die **System.IO.StreamWriter-** Klasse:

> Implementiert einen TextWriter zum Schreiben von Zeichen in einen Stream in einer bestimmten Kodierung.

Mit der `WriteLine` Methode können Sie Inhalt Zeile für Zeile in eine Datei schreiben.

Beachten Sie die Verwendung des Schlüsselworts `using` , das dafür sorgt, dass das StreamWriter-Objekt gelöscht wird, sobald es den Gültigkeitsbereich verlässt und die Datei geschlossen wird.

```
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);
    }
}
```

Beachten Sie, dass der Stream einen zweiten empfangen kann `bool` Parameter in seinem Konstruktor zu ermöglichen `Append` in eine Datei statt die Datei überschreibt:

```
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");
}
```

Schreiben in eine Datei mithilfe der System.IO.File-Klasse

Sie können die Funktion **System.IO.File.WriteAllText** verwenden, um eine Zeichenfolge in eine Datei zu schreiben.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

Sie können auch die Funktion **System.IO.File.WriteAllLines** verwenden, die als zweiten Parameter eine `IEnumerable<String>` empfängt (im Gegensatz zu einer einzelnen Zeichenfolge im vorherigen Beispiel). Dadurch können Sie Inhalte aus einem Zeilenfeld schreiben.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

Faules Lesen einer Datei zeilenweise über ein IEnumerable

Wenn Sie mit großen Dateien arbeiten, können Sie die `System.IO.File.ReadLines` Methode verwenden, um alle Zeilen aus einer Datei in einen `IEnumerable<string>` zu lesen. Dies ist ähnlich wie `System.IO.File.ReadAllLines` , nur dass die gesamte Datei nicht gleichzeitig in den Arbeitsspeicher geladen wird, was die Arbeit mit großen Dateien effizienter macht.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

Der zweite Parameter von File.ReadLines ist optional. Sie können es verwenden, wenn Sie die Kodierung angeben müssen.

Es ist wichtig zu `ToArray` , dass beim Aufruf von `ToArray` , `ToList` oder einer ähnlichen Funktion alle Zeilen gleichzeitig geladen werden müssen, was bedeutet, dass der Nutzen der Verwendung von `ReadLines` aufgehoben wird. Bei Verwendung dieser Methode `IEnumerable` , über `IEnumerable` mithilfe einer `foreach` Schleife oder LINQ aufzählen.

Erstelle Datei

Statische Dateiklasse

Mithilfe der `Create` Methode der statischen Klasse `File` können wir Dateien erstellen. Die Methode erstellt die Datei unter dem angegebenen Pfad. Gleichzeitig öffnet sie die Datei und gibt uns den `FileStream` der Datei. Stellen Sie sicher, dass Sie die Datei schließen, nachdem Sie damit fertig sind.

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-Klasse

Es gibt viele Überladungen dieses Klassenkonstruktors, die hier eigentlich gut dokumentiert sind . Das folgende Beispiel ist für dasjenige, das die am häufigsten verwendeten Funktionalitäten dieser Klasse abdeckt.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

Über diese Links können Sie die Enums für FileMode , FileAccess und FileShare überprüfen. Was sie im Grunde bedeuten, ist wie folgt:

FileMode: Antworten "Soll Datei erstellt werden? Geöffnet? Anlegen, wenn nicht vorhanden, dann geöffnet?" irgendwie Fragen.

FileAccess: Antworten "Soll ich die Datei lesen, in die Datei schreiben oder beides?" irgendwie Fragen.

FileShare: Antworten "Sollten andere Benutzer die Datei lesen, schreiben usw. können, während ich sie gleichzeitig verwende?" irgendwie Fragen.

Datei kopieren

Statische Dateiklasse

`File` statische Dateiklasse kann für diesen Zweck problemlos verwendet werden.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Anmerkung: Bei dieser Methode wird die Datei kopiert, dh sie wird aus der Quelle gelesen und dann in den Zielpfad geschrieben. Dies ist ein Ressourcenverbrauch, der relativ viel Zeit in Bezug auf die Dateigröße benötigt. Wenn Sie keine Threads verwenden, kann das Programm einfrieren.

Datei bewegen

Statische Dateiklasse

Die statische Dateiklasse kann leicht für diesen Zweck verwendet werden.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Anmerkung1: Ändert nur den Index der Datei (wenn die Datei auf demselben Volume verschoben wird). Dieser Vorgang benötigt keine relative Zeit zur Dateigröße.

Anmerkung2: Eine vorhandene Datei kann nicht im **Zielpfad** überschrieben werden.

Datei löschen

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

Während `Delete` keine Ausnahme `Delete` , wenn die Datei nicht vorhanden ist, wird eine Ausnahme ausgelöst, z. B. wenn der angegebene Pfad ungültig ist oder der Aufrufer nicht über die erforderlichen Berechtigungen verfügt. Sie sollten Aufrufe innerhalb von try-catch-Block immer in `Delete` einschließen und alle erwarteten Ausnahmen behandeln. Im Falle möglicher Wettkampfbedingungen, umschließen Sie die Logik in der Lock-Anweisung .

Dateien und Verzeichnisse

Holen Sie sich alle Dateien im Verzeichnis

```
 var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Gibt ein Array von `FileInfo` , das alle Dateien im angegebenen Verzeichnis darstellt.

Holen Sie sich Dateien mit bestimmten Erweiterungen

```
 var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Gibt ein Array von `FileInfo` , das alle Dateien im angegebenen Verzeichnis mit der angegebenen Erweiterung darstellt.

Async schreibt mit StreamWriter Text in eine Datei

```
// 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);
}
```

Datei- und Stream-E / A online lesen: https://riptutorial.com/de/csharp/topic/4266/datei--und-stream-e---a

Kapitel 39: Datenanmerkung

Examples

DisplayNameAttribute (Anzeigeattribut)

`DisplayName` legt den Anzeigenamen für eine Eigenschafts-, Ereignis- oder öffentliche Void-Methode mit null (0) Argumenten fest.

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

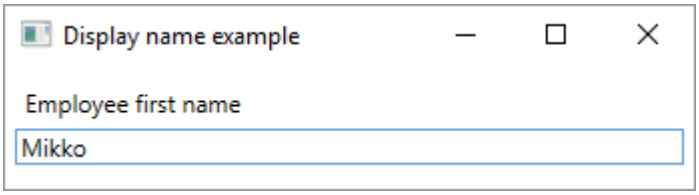
Einfaches Anwendungsbeispiel in der XAML-Anwendung

```
<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 (Datenmodellierungsattribut)

`EditableAttribute` legt fest, ob Benutzer den Wert der Klasseneigenschaft ändern können sollen.

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

Einfaches Anwendungsbeispiel in der XAML-Anwendung

```
<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();
        }
    }
}
```

Validierungsattribute

Validierungsattribute werden verwendet, um verschiedene Validierungsregeln deklarativ für Klassen oder Klassenmitglieder durchzusetzen. Alle Validierungsattribute werden von der ValidationAttribute- Basisklasse abgeleitet.

Beispiel: RequiredAttribute

Bei der Validierung durch die `ValidationAttribute.Validate` Methode gibt dieses Attribut einen Fehler zurück, wenn die `Name` Eigenschaft null ist oder nur Leerzeichen enthält.

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

Beispiel: StringLengthAttribute

Das `StringLengthAttribute` , ob eine Zeichenfolge kleiner als die maximale Länge einer Zeichenfolge ist. Optional kann eine Mindestlänge angegeben werden. Beide Werte sind inklusive.

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

Beispiel: RangeAttribute

Das `RangeAttribute` gibt den maximalen und minimalen Wert für ein numerisches Feld an.

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

Beispiel: CustomValidationAttribute

Mit der `CustomValidationAttribute` Klasse kann eine benutzerdefinierte `static` Methode zur Überprüfung aufgerufen werden. Die benutzerdefinierte Methode muss `static ValidationResult [MethodName] (object input)` .

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

Methodendeklaration:

```
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;
    }
}
```

Angepasstes Validierungsattribut erstellen

Benutzerdefinierte Validierungsattribute können erstellt werden, indem sie von der `ValidationAttribute` Basisklasse abgeleitet und anschließend bei Bedarf `virtual` Methoden überschrieben werden.

```
[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;
    }
}
```

Dieses Attribut kann dann wie folgt verwendet werden:

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

Grundlagen der Datenanmerkungen

Datenanmerkungen sind eine Möglichkeit, Klassen oder Member einer Klasse mit mehr Kontextinformationen zu versehen. Es gibt drei Hauptkategorien von Anmerkungen:

- Validierungsattribute: Hinzufügen von Validierungskriterien zu Daten
- Attribute anzeigen: Geben Sie an, wie die Daten dem Benutzer angezeigt werden sollen
- Modellierungsattribute: Fügen Sie Informationen zur Verwendung und Beziehung zu anderen Klassen hinzu

Verwendungszweck

Hier ist ein Beispiel, bei dem zwei `ValidationAttribute` und ein `DisplayAttribute` verwendet werden:

```
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; }
}
```

Datenanmerkungen werden meist in Frameworks wie ASP.NET verwendet. Wenn zum Beispiel in `ASP.NET MVC` ein Modell von einer Controller-Methode empfangen wird, kann mit `ModelState.IsValid()` festgestellt werden, ob das empfangene Modell alle seine `ValidationAttribute` `ModelState.IsValid()` . `DisplayAttribute` wird auch in `ASP.NET MVC` , um zu bestimmen, wie Werte auf einer Webseite angezeigt werden.

Validierungsattribute manuell ausführen

In den meisten Fällen werden Validierungsattribute in Frameworks (z. B. ASP.NET) verwendet. Diese Frameworks sorgen für die Ausführung der Validierungsattribute. Was aber, wenn Sie Validierungsattribute manuell ausführen möchten? Verwenden Sie einfach die `Validator` Klasse

(keine Reflektion erforderlich).

Validierungskontext

Jede Validierung benötigt einen Kontext, um Informationen darüber zu erhalten, was geprüft wird. Dazu können verschiedene Informationen gehören, z. B. das zu überprüfende Objekt, einige Eigenschaften, der Name, der in der Fehlermeldung angezeigt werden soll usw.

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

Nachdem der Kontext erstellt wurde, gibt es mehrere Möglichkeiten, die Validierung durchzuführen.

Überprüfen Sie ein Objekt und alle seine Eigenschaften

```
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
```

Bestätigen Sie eine Eigenschaft eines Objekts

```
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
```

Und mehr

Weitere Informationen zur manuellen Validierung finden Sie unter:

- ValidationContext-Klassendokumentation
- Validator-Klassendokumentation

Datenanmerkung online lesen: https://riptutorial.com/de/csharp/topic/4942/datenanmerkung

Kapitel 40: Datenflusskonstruktionen für Task Parallel Library (TPL)

Examples

JoinBlock

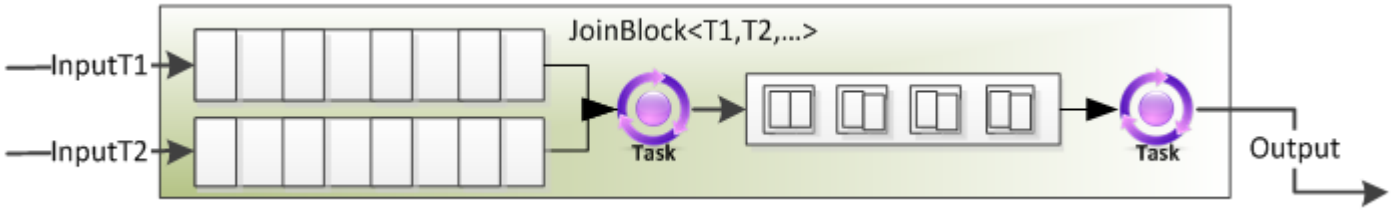
(Sammelt 2-3 Eingaben und kombiniert sie zu einem Tupel)

Wie bei BatchBlock können mit JoinBlock <T1, T2,…> Daten aus mehreren Datenquellen gruppiert werden. Tatsächlich ist dies der Hauptzweck von JoinBlock <T1, T2,…>.

Ein JoinBlock <string, double, int> ist beispielsweise ein ISourceBlock <Tuple <string, double, int >>.

Wie bei BatchBlock kann JoinBlock <T1, T2,…> sowohl im gierigen als auch im nicht gierigen Modus arbeiten.

- Im standardmäßigen Greedy-Modus werden alle den Zielen angebotenen Daten akzeptiert, auch wenn das andere Ziel nicht über die erforderlichen Daten verfügt, um ein Tupel zu bilden.
- Im nicht-gierigen Modus verschieben die Ziele des Blocks die Daten, bis allen Zielen die erforderlichen Daten zum Erstellen eines Tupels zur Verfügung gestellt wurden. Zu diesem Zeitpunkt verwendet der Block ein zweiphasiges Festschreibungsprotokoll, um alle erforderlichen Elemente aus den Quellen atomar abzurufen. Diese Verschiebung ermöglicht es einer anderen Entität, die Daten zwischenzeitlich zu verbrauchen, um dem Gesamtsystem einen Fortschritt zu ermöglichen.



Anfragen mit einer begrenzten Anzahl gepoolter Objekte bearbeiten

```
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);
```

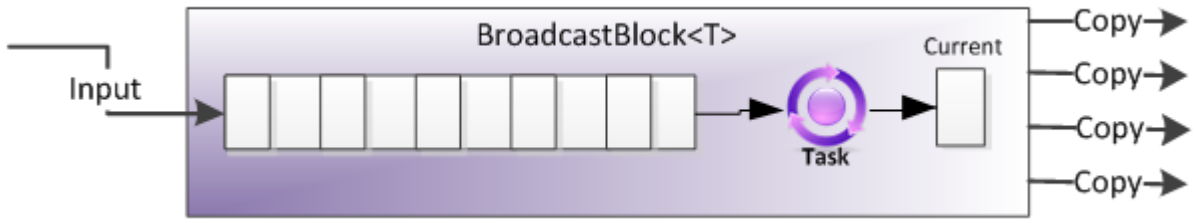
Einführung in TPL Dataflow von Stephen Toub

BroadcastBlock

(Kopieren Sie ein Objekt und senden Sie die Kopien an jeden Block, mit dem es verlinkt ist.)

Im Gegensatz zu BufferBlock besteht das Ziel von BroadcastBlock darin, allen mit dem Block verknüpften Zielen zu ermöglichen, eine Kopie jedes veröffentlichten Elements zu erhalten, wobei der "aktuelle" Wert ständig mit den an ihn weitergegebenen Objekten überschrieben wird.

Im Gegensatz zu BufferBlock hält BroadcastBlock die Daten nicht unnötig an. Nachdem ein bestimmtes Datum für alle Ziele angeboten wurde, wird dieses Element durch das nächste Datenelement überschrieben (wie bei allen Datenflussblöcken werden Nachrichten in der FIFO-Reihenfolge behandelt). Dieses Element wird allen Zielen angeboten und so weiter.



Asynchroner Producer / Consumer mit einem gedrosselten Producer

```
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);
```

Anzeigen des Status eines Agenten

```
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");
    }
}
```

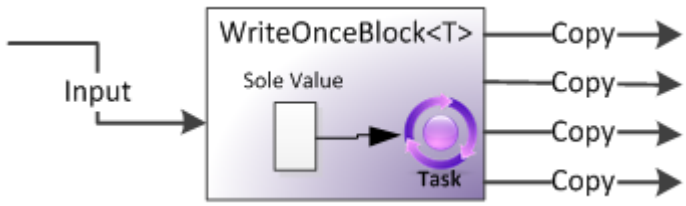
Einführung in TPL Dataflow von Stephen Toub

WriteOnceBlock

(Readonly-Variable: Speichert das erste Datenelement und gibt Kopien davon als Ausgabe aus. Ignoriert alle anderen Datenelemente.)

Wenn BufferBlock der grundlegendste Block in TPL Dataflow ist, ist WriteOnceBlock der einfachste.
Es speichert höchstens einen Wert. Sobald dieser Wert festgelegt wurde, wird er niemals ersetzt oder überschrieben.

Sie können sich WriteOnceBlock wie eine readonly-Membervariable in C # vorstellen, mit der Ausnahme, dass Sie nicht nur in einem Konstruktor einstellbar und dann unveränderlich sind, sondern nur einmal einstellbar und dann unveränderlich sind.



Potentielle Ausgaben einer Aufgabe aufteilen

```
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)
```

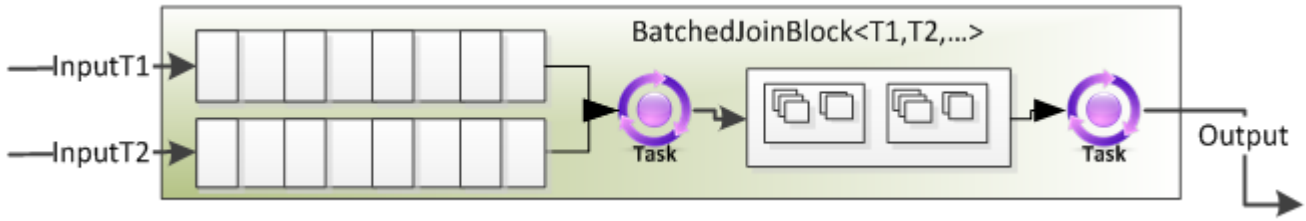
```
    {
        exception.Post(ex);
    }
}
```

Einführung in TPL Dataflow von Stephen Toub

BatchedJoinBlock

(Sammelt eine bestimmte Anzahl von Gesamtelementen aus 2-3 Eingaben und gruppiert sie in einem Tuple von Sammlungen von Datenelementen.)

BatchedJoinBlock <T1, T2, ...> ist gewissermaßen eine Kombination aus BatchBlock und JoinBlock <T1, T2, ...>.
Während JoinBlock <T1, T2,…> verwendet wird, um eine Eingabe von jedem Ziel zu einem Tupel zu aggregieren, und BatchBlock, um N Eingaben in einer Sammlung zu aggregieren, wird BatchedJoinBlock <T1, T2,…> verwendet, um N Eingaben von across zu sammeln alle Ziele in Tupel von Sammlungen.



Streuung / Sammeln

Stellen Sie sich ein Scatter / Gather-Problem vor, bei dem N-Operationen gestartet werden, von denen einige erfolgreich sein können und String-Ausgaben erzeugen können, und andere, die möglicherweise fehlschlagen und Ausnahmen erzeugen.

```
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);
}
```

Einführung in TPL Dataflow von Stephen Toub

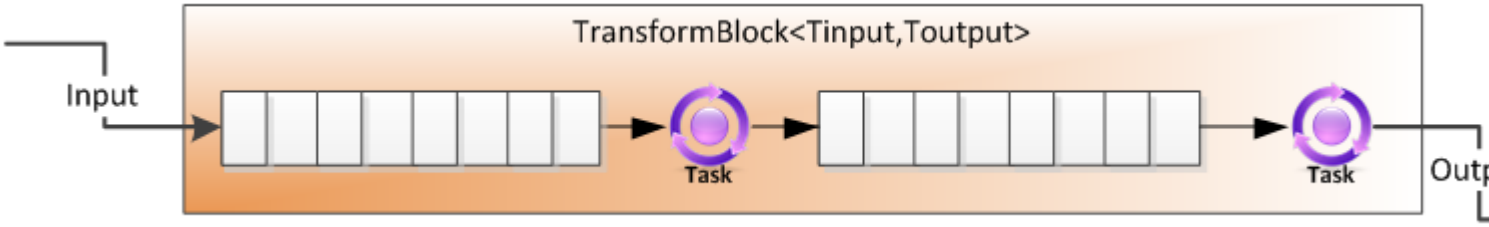
TransformBlock

(Wählen Sie eins zu eins aus)

Wie bei ActionBlock ermöglicht TransformBlock <TInput, TOutput> die Ausführung eines Delegaten, um eine Aktion für jedes Eingabedatum auszuführen. **Im Gegensatz zu ActionBlock hat diese Verarbeitung eine Ausgabe.** Dieser Delegat kann ein Func <TInput, TOutput> sein. In diesem Fall wird die Verarbeitung dieses Elements als abgeschlossen betrachtet, wenn der Delegat zurückgegeben wird, oder er kann ein Func <TInput, Task> sein. In diesem Fall wird die Verarbeitung dieses Elements als nicht abgeschlossen betrachtet wenn der Delegat zurückkehrt, aber wenn die zurückgegebene Aufgabe abgeschlossen ist. Für diejenigen, die mit LINQ vertraut sind, ist es Select () ein wenig ähnlich, dass es eine Eingabe übernimmt, diese Eingabe in eine bestimmte Weise transformiert und dann eine Ausgabe erzeugt.

Standardmäßig verarbeitet TransformBlock <TInput, TOutput> seine Daten sequentiell mit einem MaxDegreeOfParallelism-Wert von 1. Zusätzlich zum Empfang von gepufferten Eingaben und deren Verarbeitung nimmt dieser Block seine gesamte verarbeitete Ausgabe auf und puffert diese (auch nicht vorhandene Daten) verarbeitet und Daten, die verarbeitet wurden).

Es hat zwei Aufgaben: eine für die Verarbeitung der Daten und eine für die Weitergabe der Daten an den nächsten Block.



Eine gleichzeitige Pipeline

```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

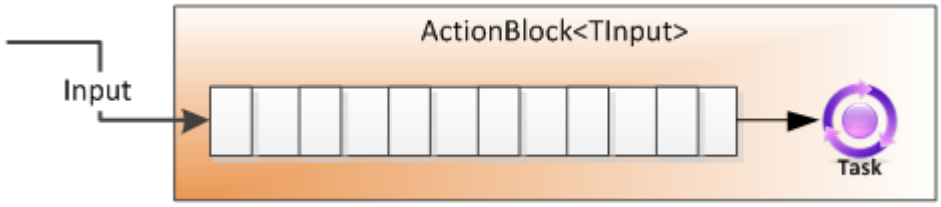
Einführung in TPL Dataflow von Stephen Toub

ActionBlock

(für jeden)

Diese Klasse kann logisch als Puffer für zu verarbeitende Daten mit Aufgaben zur Verarbeitung dieser Daten betrachtet werden, wobei der „Datenflussblock“ beide verwaltet. In der einfachsten Verwendung können wir einen ActionBlock instanziieren und Daten darauf "buchen". Der bei der ActionBlock-Konstruktion bereitgestellte Delegat wird asynchron für alle gesendeten Daten

ausgeführt.



Synchrone Berechnung

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Drosseln asynchroner Downloads auf maximal 5 gleichzeitig

```
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");
```

Einführung in TPL Dataflow von Stephen Toub

TransformManyBlock

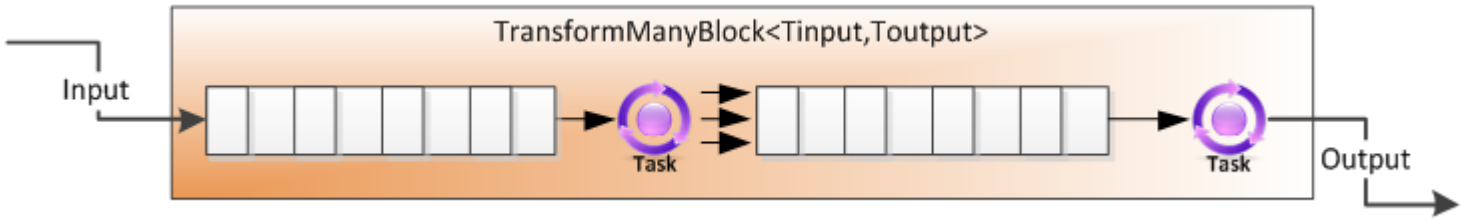
(SelectMany, 1-m: Die Ergebnisse dieser Zuordnung werden wie bei SelectMany von LINQ "abgeflacht".)

TransformManyBlock <TInput, TOutput> ist TransformBlock <TInput, TOutput> sehr ähnlich. Der Hauptunterschied besteht darin, dass ein TransformBlock <TInput, TOutput> für jeden Eingang nur einen Ausgang erzeugt, während TransformManyBlock <TInput, TOutput> für jeden Eingang eine beliebige Anzahl (null oder mehr) Ausgänge erzeugt. Wie bei ActionBlock und TransformBlock <TInput, TOutput> kann diese Verarbeitung mithilfe von Delegaten angegeben werden, und zwar sowohl für die synchrone als auch für die asynchrone Verarbeitung.

Ein Func <TInput, IEnumerable> wird für synchron verwendet, und ein Func <TInput, Task <IEnumerable> wird für asynchron verwendet. Wie bei ActionBlock und TransformBlock <TInput wird bei TOutput>, TransformManyBlock <TInput, TOutput> standardmäßig die sequentielle Verarbeitung verwendet, sie kann jedoch auch anders konfiguriert werden.

Der Zuordnungsdelegierte führt eine Sammlung von Elementen erneut aus, die einzeln in den

Ausgabepuffer eingefügt werden.



Asynchroner Web Crawler

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Ausdehnung eines Aufzählers in seine Bestandteile

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Filtern durch Wechseln von 1 zu 0 oder 1 Elementen

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Einführung in TPL Dataflow von Stephen Toub

BatchBlock

(Gruppiert eine bestimmte Anzahl von sequentiellen Datenelementen in Sammlungen von Datenelementen.)

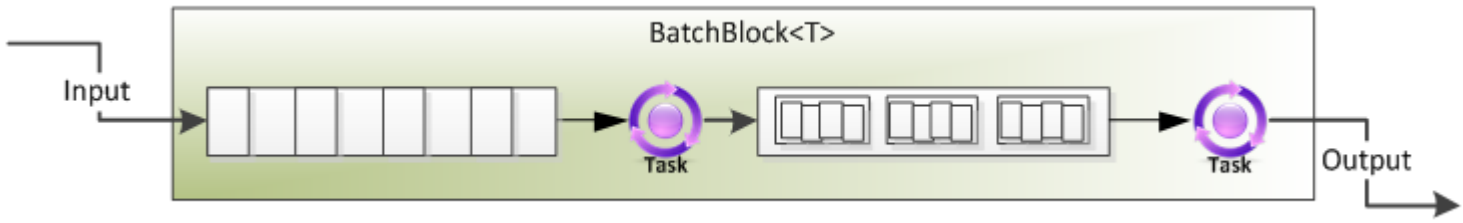
BatchBlock kombiniert N einzelne Elemente in einem Stapelelement, das als Array von Elementen dargestellt wird. Eine Instanz wird mit einer bestimmten Stapelgröße erstellt. Der Block erstellt dann einen Stapel, sobald er diese Anzahl von Elementen erhalten hat, und gibt den Stapel asynchron in den Ausgabepuffer aus.

BatchBlock kann sowohl im gierigen als auch im nicht gierigen Modus ausgeführt werden.

- Im voreingestellten Greedy-Modus werden alle Meldungen, die aus einer beliebigen Anzahl von Quellen zum Block angeboten werden, akzeptiert und zwischengespeichert, um in

Stapel umgewandelt zu werden.

- Im nicht-gierigen Modus werden alle Nachrichten aus Quellen verschoben, bis genügend Quellen dem Block Angebote zum Erstellen eines Stapels angeboten haben. Somit kann ein BatchBlock verwendet werden, um 1 Element von jeder der N Quellen, N Elemente von einer Quelle und eine Vielzahl von Optionen dazwischen zu empfangen.



Batch-Anforderungen in Gruppen von 100 zum Übermitteln an eine Datenbank

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Einmal pro Sekunde einen Stapel erstellen

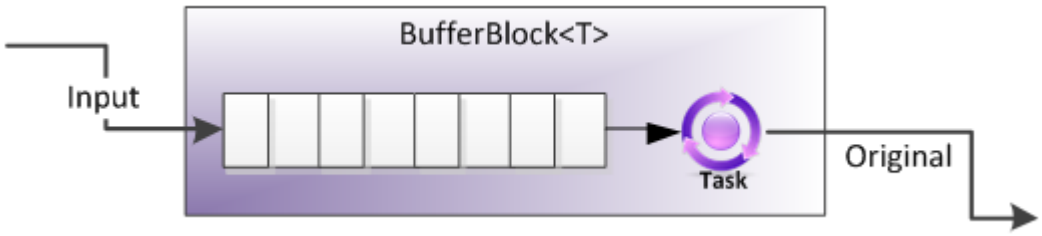
```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Einführung in TPL Dataflow von Stephen Toub

BufferBlock

(FIFO-Warteschlange: Die eingehenden Daten sind die Daten, die ausgehen.)

Kurz gesagt, BufferBlock stellt einen unbegrenzten oder beschränkten Puffer zum Speichern von Instanzen von T bereit.
Sie können Instanzen von T in den Block "buchen", wodurch die gesendeten Daten in einer FIFO-Reihenfolge (First-In-First-Out) des Blocks gespeichert werden.
Sie können von dem Block aus "empfangen", wodurch Sie synchron oder asynchron Instanzen von T erhalten können, die zuvor gespeichert wurden oder in der Zukunft verfügbar sind (wiederum FIFO).



Asynchroner Producer / Consumer mit einem gedrosselten Producer

```
// 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());
}
```

Einführung in TPL Dataflow von Stephen Toub

Datenflusskonstruktionen für Task Parallel Library (TPL) online lesen:
https://riptutorial.com/de/csharp/topic/3110/datenflusskonstruktionen-fur-task-parallel-library--tpl-

Kapitel 41: DateTime-Methoden

Examples

DateTime.Add (TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

DateTime.AddDays (Double)

Fügen Sie Tage in ein DateTime-Objekt ein.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

Sie können auch Tage abziehen, die einen negativen Wert überschreiten:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

DateTime.AddHours (Double)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

DateTime.AddMilliseconds (doppelt)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
```

```
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

DateTime.Compare (DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

DateTime.DaysInMonth (Int32, Int32)

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

DateTime.AddYears (Int32)

Fügen Sie dem dateTime-Objekt Jahre hinzu:

```
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));
```

Reine Funktionswarnung beim Umgang mit DateTime

Wikipedia definiert derzeit eine reine Funktion wie folgt:

1. Die Funktion wertet immer denselben Ergebniswert aus, wenn dieselben Argumentwerte verwendet werden. Der Funktionsergebniswert kann nicht von versteckten Informationen oder Zuständen abhängen, die sich während der Programmausführung oder zwischen verschiedenen Programmausführungen ändern können, und kann auch nicht von externen Eingaben von E / A-Geräten abhängen.
2. Die Auswertung des Ergebnisses verursacht keine semantisch beobachtbaren Nebeneffekte oder Ausgaben wie Mutationen von veränderlichen Objekten oder Ausgaben an E / A-Geräte

Als Entwickler müssen Sie sich der reinen Methoden bewusst sein, und Sie werden in vielen Bereichen auf diese stoßen. Eine, die ich gesehen habe, die viele junge Entwickler beißt, arbeitet mit DateTime-Klassenmethoden. Viele davon sind rein und wenn Sie sich dessen nicht bewusst sind, können Sie sich überraschen lassen. Ein Beispiel:

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

In Anbetracht des obigen Beispiels kann man davon ausgehen, dass das auf der Konsole ausgedruckte Ergebnis '26 / 12/2016 'ist, aber in Wirklichkeit endet das gleiche Datum. Dies liegt daran, dass AddDays eine reine Methode ist und das ursprüngliche Datum nicht beeinflusst. Um die erwartete Ausgabe zu erhalten, müssen Sie den Aufruf von AddDays folgendermaßen ändern:

```
        sample = sample.AddDays(1);
```

DateTime.Parse (String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

DateTime.TryParse (String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent
and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};
```

```
foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

Parse und TryParse mit Kulturinformationen

Möglicherweise möchten Sie es verwenden, wenn DateTimes aus verschiedenen Kulturen (Sprachen) analysiert werden. Beispiel: Niederländisches Datum wird analysiert.

```
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}
```

Beispiel für Parse:

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

DateTime als Initialisierer in der for-Schleife

```
// 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());
}
```

Das Iterieren auf einem `TimeSpan` funktioniert genauso.

DateTime ToString, ToShortDateString, ToLongDateString und ToString formatiert

```
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
    }
}
```

Aktuelles Datum

Um das aktuelle Datum `DateTime.Today` Sie die `DateTime.Today` Eigenschaft. Dies gibt ein `DateTime` Objekt mit dem heutigen Datum zurück. Wenn diese dann konvertiert wird `.ToString()` wird dies standardmäßig in der `.ToString()` Ihres Systems durchgeführt.

Zum Beispiel:

```
Console.WriteLine(DateTime.Today);
```

Schreibt das heutige Datum in Ihrem lokalen Format in die Konsole.

DateTime-Formatierung

Standard DateTime Formatierung

DateTimeFormatInfo gibt einen Satz von Bezeichnern für die einfache Datums- und Zeitformatierung an. Jeder Bezeichner entspricht einem bestimmten DateTimeFormatInfo-Formatmuster.

```
//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
```

Benutzerdefinierte DateTime-Formatierung

Es gibt folgende benutzerdefinierte Formatbezeichner:

- `y` (Jahr)
- `M` (Monat)
- `d` (Tag)
- `h` (stunde 12)
- `H` (Stunde 24)
- `m` (Minute)
- `s` (Sekunde)
- `f` (zweite Fraktion)
- `F` (zweiter Bruch, nachfolgende Nullen werden getrimmt)
- `t` (PM oder AM)
- `z` (Zeitzone).

```
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
```

Sie können auch Datumstrennzeichen `/` (Schrägstrich) und Zeitseparator verwenden `:` (Doppelpunkt).

Für Codebeispiel

Weitere Informationen MSDN .

DateTime.ParseExact (String, String, IFormatProvider)

Konvertiert die angegebene Zeichenfolgendarstellung eines Datums und einer Uhrzeit in das entsprechende DateTime-Format unter Verwendung des angegebenen Formats und kulturspezifischer Formatinformationen. Das Format der Zeichenfolgendarstellung muss genau mit dem angegebenen Format übereinstimmen.

Konvertieren Sie eine bestimmte Formatzeichenfolge in die entsprechende DateTime

Nehmen wir an, wir haben einen kulturspezifischen DateTime-String `08-07-2016 11:30:12 PM` als `MM-dd-yyyy hh:mm:ss tt` Format, und wir möchten, dass er in ein entsprechendes `DateTime` Objekt `DateTime` wird

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

Konvertieren Sie eine Datumszeitzeichenfolge in ein entsprechendes `DateTime` Objekt ohne ein bestimmtes Kulturformat

Nehmen wir an, wir haben einen DateTime-String im Format `dd-MM-yy hh:mm:ss tt` und möchten, dass er ohne entsprechende `DateTime` in ein entsprechendes `DateTime` Objekt konvertiert wird

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

Konvertieren Sie eine Datumszeitzeichenfolge in ein entsprechendes DateTime-Objekt ohne ein bestimmtes Kulturformat mit einem anderen Format

Nehmen wir an, wir haben eine Date-Zeichenfolge, ein Beispiel wie '23 -12-2016 'oder '12 / 23/2016', und wir möchten, dass es in ein entsprechendes `DateTime` Objekt ohne spezifische Kulturinformationen konvertiert wird

```
   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);
```

`System.Globalization` : Für CultureInfo-Klasse muss `System.Globalization` hinzugefügt werden

DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime)

Konvertiert die angegebene Zeichenfolgendarstellung eines Datums und einer Uhrzeit in das entsprechende DateTime-Format, wobei das angegebene Format, die kulturspezifischen Formatinformationen und der Stil verwendet werden. Das Format der Zeichenfolgendarstellung muss genau mit dem angegebenen Format übereinstimmen. Die Methode gibt einen Wert zurück, der angibt, ob die Konvertierung erfolgreich war.

Zum Beispiel

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

Analysieren Sie Datum ohne Stilkennzeichen.

```
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);
}
```

Verwenden Sie benutzerdefinierte Formate mit M und MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Eine Zeichenfolge mit Zeitzoneninformationen parsen.

```
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);
```

```
}
```

Analysiert eine Zeichenfolge, die UTC darstellt.

```
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);
}
```

Ausgänge

```
' 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).
```

DateTime-Methoden online lesen: https://riptutorial.com/de/csharp/topic/1587/datetime-methoden

Kapitel 42: Delegierte

Bemerkungen

Zusammenfassung

Ein **Delegattyp** ist ein Typ, der eine bestimmte Methodensignatur darstellt. Eine Instanz dieses Typs bezieht sich auf eine bestimmte Methode mit einer übereinstimmenden Signatur. Methodenparameter können Delegat-Typen haben, so dass dieser einen Methode ein Verweis auf eine andere Methode übergeben wird, die dann aufgerufen werden kann

`Func<...,TResult>` : `Action<...>` , `Predicate<T>` und `Func<...,TResult>`

Der `System` Namespace enthält die Delegaten `Action<...>` , `Predicate<T>` und `Func<...,TResult>` , wobei "..." generische Typparameter darstellt (für 0-Parameter ist `Action` generisch).

`Func` repräsentiert Methoden mit einem Rückgabetyp, der mit `TResult` übereinstimmt, und `Action` stellt Methoden ohne Rückgabewert dar (void). In beiden Fällen stimmen die zusätzlichen generischen Typparameter der Reihenfolge nach mit den Methodenparametern überein.

`Predicate` für Methode mit booleschem Rückgabetyp, T ist Eingabeparameter.

Benutzerdefinierte Delegatentypen

Benannte Delegattypen können mit dem `delegate` Schlüsselwort deklariert werden.

Delegierte aufrufen

Delegaten können mit derselben Syntax wie Methoden aufgerufen werden: Der Name der Delegateninstanz, gefolgt von Klammern, die Parameter enthalten.

Delegierten zuweisen

Delegierten können auf folgende Weise zugewiesen werden:

- Eine benannte Methode zuweisen
- Zuweisen einer anonymen Methode mit einem Lambda
- Zuweisen einer benannten Methode mit dem `delegate` Schlüsselwort

Delegierte kombinieren

Mit dem Operator `+` können mehrere Delegatenobjekte einer Delegateninstanz zugewiesen werden. Mit dem Operator `-` kann ein Komponentendelegat aus einem anderen Delegat entfernt werden.

Examples

Basiswerte der benannten Methodendelegaten

Wenn Sie Delegaten benannte Methoden zuweisen, beziehen sie sich auf dasselbe zugrunde liegende Objekt, wenn

- Sie sind die gleiche Instanzmethode in derselben Instanz einer Klasse
- Sie sind dieselbe statische Methode für eine Klasse

```
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
```

Delegatentyp deklarieren

Die folgende Syntax erstellt einen `delegate` mit dem Namen `NumberInOutDelegate` , ein Verfahren darstellt , welches einen nimmt `int` und gibt eine `int` .

```
public delegate int NumberInOutDelegate(int input);
```

Dies kann wie folgt verwendet werden:

```
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;
    }
}
```

Die `example` Delegateninstanz wird auf dieselbe Weise wie die `Square` Methode ausgeführt. Eine Delegateninstanz fungiert buchstäblich als Delegat für den Aufrufer: Der Aufrufer ruft den Delegaten auf, und der Delegat ruft die Zielmethode auf. Diese Umleitung entkoppelt den Aufrufer von der Zielmethode.

Sie können einen **generischen** Delegattyp deklarieren. In diesem Fall können Sie in einigen der Typargumente angeben, dass der Typ kovariant (`out`) oder contravariant (`in`) ist. Zum Beispiel:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Wie andere generische Typen können auch generische `where TFrom : struct, IConvertible where TTo : new()` Einschränkungen aufweisen, z. B. `where TFrom : struct, IConvertible where TTo : new()` .

Vermeiden Sie Ko- und Gegensätze für Delegatetypen, die für Multicast-Delegaten verwendet werden sollen, wie z. B. Ereignishandlertypen. Dies liegt daran, dass die Verkettung (`+`) fehlschlagen kann, wenn sich der Laufzeittyp aufgrund der Abweichung vom Kompilierzeittyp unterscheidet. Vermeiden Sie zum Beispiel:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Verwenden Sie stattdessen einen generischen invarianten Typ:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Unterstützt werden auch Delegierte, bei denen einige Parameter durch `ref` oder `out` geändert `out` , wie in:

```
public delegate bool TryParser<T>(string input, out T result);
```

(Beispiel use `TryParser<decimal> example = decimal.TryParse;`) oder Delegaten, bei denen der letzte Parameter den Parameter `params` . Delegatetypen können optionale Parameter haben (Standardwerte für die Bereitstellung). Stellvertretertypen können Zeigertypen wie `int*` oder `char*` in ihren Signaturen oder Rückgabetypen verwenden (verwenden Sie ein `unsafe` Schlüsselwort). Ein Delegattyp und seine Parameter können benutzerdefinierte Attribute enthalten.

Die Func , Aktion und Prädikat Delegiertypen

Der System-Namespace enthält `Func<..., TResult>` mit generischen Parametern zwischen 0 und 15, wobei der Typ `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);
}
```

Der System-Namespace enthält außerdem `Action<...>` Delegattypen mit einer unterschiedlichen Anzahl generischer Parameter (von 0 bis 16). Es ist ähnlich zu `Func<T1, .., Tn>` , gibt aber immer `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>` ist auch eine Form von `Func` , aber es wird immer wieder zurückkehren `bool` . Mit einem Prädikat können Sie benutzerdefinierte Kriterien angeben. Abhängig vom Wert der Eingabe und der im Prädikat definierten Logik gibt es entweder " `true` oder " `false` . `Predicate<T>` verhält

sich daher wie `Func<T, bool>` und beide können initialisiert und auf dieselbe Weise verwendet werden.

```
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");
```

Die Wahl, ob `Predicate<T>` oder `Func<T, bool>` ist wirklich eine `Func<T, bool>` . `Predicate<T>` `Func<T, bool>` die Absicht des Autors wohl mehr zum Ausdruck, während `Func<T, bool>` wahrscheinlich einem größeren Teil der C # `Func<T, bool>` ist.

Darüber hinaus gibt es einige Fälle, in denen nur eine der Optionen verfügbar ist, insbesondere bei der Interaktion mit einer anderen API. Beispielsweise verwenden `List<T>` und `Array<T>` Allgemeinen `Predicate<T>` für ihre Methoden, während die meisten LINQ-Erweiterungen nur `Func<T, bool>` akzeptieren.

Zuweisen einer benannten Methode zu einem Delegaten

Benannte Methoden können Delegierten mit übereinstimmenden Signaturen zugewiesen werden:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne` nimmt ein `int` und gibt ein `int` , dessen Signatur dem Delegaten `Func<int,int>` . `Example.AddOne` kann direkt zugeordnet werden `addOne` weil sie passende Signaturen haben.

Gleichstellung delegieren

Das Aufrufen von `.Equals()` für einen Delegaten vergleicht anhand der Referenzgleichheit:

```
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
```

Diese Regeln gelten auch für `+=` oder `-=` für einen Multicast-Delegierten, z. B. für das Abonnieren und Abbestellen von Ereignissen.

Zuweisung eines Delegierten durch Lambda

Mit Lambdas können anonyme Methoden erstellt werden, die einem Delegierten zugewiesen werden können:

```
Func<int,int> addOne = x => x+1;
```

Beachten Sie, dass die explizite Typdeklaration erforderlich ist, wenn Sie eine Variable auf diese Weise erstellen:

```
var addOne = x => x+1; // Does not work
```

Delegaten als Parameter übergeben

Delegierte können als typisierte Funktionszeiger verwendet werden:

```
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;
      }
    }
  }
}
```

Delegierte kombinieren (Multicast-Delegierte)

Addition `+` und Subtraktion `-` Operationen können verwendet werden, um Delegierungsinstanzen zu kombinieren. Der Delegat enthält eine Liste der zugewiesenen Delegierten.

```
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);
        }
    }
}
```

In diesem Beispiel ist `d3` eine Kombination aus `d1` und `d2` Delegaten. Wenn das Programm aufgerufen wird, gibt es `System.Int32` Zeichenfolgen `1` und `System.Int32` .

Delegaten mit **nicht ungültigen** Rückgabetypen kombinieren:

Wenn ein Multicastdelegat einen Rückgabetyp ohne `nonvoid` , erhält der Aufrufer den Rückgabewert von der zuletzt `nonvoid` Methode. Die vorhergehenden Methoden werden weiterhin aufgerufen, ihre Rückgabewerte werden jedoch verworfen.

```
    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)` ruft zuerst `Square` und dann `Cube` . Der Rückgabewert von Square wird verworfen und der Rückgabewert der letzten Methode, dh `Cube` wird beibehalten.

Sichern Sie den Multicast-Delegaten

Wollten Sie schon immer einen Multicast-Delegaten anrufen, möchten Sie jedoch, dass die gesamte Anrufliste auch dann aufgerufen wird, wenn in der Kette eine Ausnahme auftritt. Dann haben Sie Glück, ich habe eine Erweiterungsmethode erstellt, die genau das tut und eine `AggregateException` nur nach Ausführung der gesamten Liste auslöst:

```
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();
    }
}
```

Dies gibt aus:

```
Target 2 executed
Target 1 executed
```

`SaveInvoke` Aufrufen ohne `SaveInvoke` würde nur Ziel 2 ausführen.

Schließung in einem Delegierten

Verschlüsse sind Inline - anonyme Methoden, die die Fähigkeit zu verwenden , haben `Parent` Methode Variablen und andere anonyme Methoden , die in der übergeordneten Umfang definiert sind.

> Im Wesentlichen ist ein Abschluss ein Codeblock, der zu einem späteren Zeitpunkt ausgeführt werden kann, jedoch die Umgebung beibehält, in der er erstellt wurde, dh er kann die lokalen Variablen usw. der Methode, die ihn erstellt hat, auch danach noch verwenden Methode wurde beendet. **- Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
```

```
    };
    int bar = myClosure();

}
```

Beispiel aus Closures in .NET .

Transformationen in Funktionen einkapseln

```
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();
        }
    }
}
```

Im Sinne einer sauberen Codierung kann das Einkapseln von Überprüfungen und Transformationen wie der oben genannten als Func Ihren Code leichter lesbar und verständlicher machen. Das obige Beispiel ist sehr einfach. Was wäre, wenn es mehrere DateTime-Eigenschaften mit jeweils unterschiedlichen Validierungsregeln gäbe und wir verschiedene Kombinationen prüfen wollten? Einfache, einzeilige Funcs, die jeweils über eine Rückgabelogik verfügen, können sowohl lesbar sein als auch die scheinbare Komplexität Ihres Codes reduzieren. Betrachten Sie die folgenden Func-Aufrufe und stellen Sie sich vor, wie viel mehr Code die Methode durcheinanderbringen würde:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Delegierte online lesen: https://riptutorial.com/de/csharp/topic/1194/delegierte

Kapitel 43: Diagnose

Examples

Debug.WriteLine

Schreibt an die Trace-Listener in der Listeners-Auflistung, wenn die Anwendung in der Debug-Konfiguration kompiliert wird.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

In Visual Studio oder Xamarin Studio wird dies im Anwendungsausgabefenster angezeigt. Dies ist auf das Vorhandensein des Standard-Trace-Listeners in der TraceListenerCollection zurückzuführen.

Protokollausgabe mit TraceListeners umleiten

Sie können die Debugausgabe in eine Textdatei umleiten, indem Sie der Sammlung Debug.Listeners einen TextWriterTraceListener hinzufügen.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

Sie können die Debug-Ausgabe mithilfe eines ConsoleTraceListener in den Ausgangsstrom einer Konsolenanwendung umleiten.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Diagnose online lesen: https://riptutorial.com/de/csharp/topic/2147/diagnose

Kapitel 44: Dynamischer Typ

Bemerkungen

Das `dynamic` Schlüsselwort deklariert eine Variable, deren Typ zur Kompilierzeit nicht bekannt ist. Eine `dynamic` Variable kann einen beliebigen Wert enthalten, und der Typ des Werts kann sich zur Laufzeit ändern.

Wie in dem Buch "Metaprogrammierung in .NET" erwähnt, hat C # keinen Hintergrundtyp für das `dynamic` Schlüsselwort:

> Die durch das Schlüsselwort `dynamic` aktivierte Funktionalität ist eine clevere Gruppe von Compileraktionen, die `CallSite` Objekte im Site-Container des lokalen Ausführungsbereichs ausgeben und verwenden. Der Compiler verwaltet, was Programmierer über diese `CallSite` Instanzen als dynamische Objektreferenzen `CallSite` . Die Parameter, Rückgabetypen, Felder und Eigenschaften, die zur Kompilierzeit dynamisch behandelt werden, sind möglicherweise mit einigen Metadaten gekennzeichnet, um anzuzeigen, dass sie für die dynamische Verwendung generiert wurden. Der zugrunde liegende Datentyp für sie ist immer `System.Object` .

Examples

Dynamische Variable erstellen

```
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
```

Dynamisch zurückkehren

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
```

```
}
```

Dynamisches Objekt mit Eigenschaften erstellen

```
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
```

Umgang mit bestimmten Typen, die zur Kompilierzeit unbekannt sind

Die folgenden Ergebnisse liefern gleichwertige Ergebnisse:

```
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())}";
    }
}
```

Der Vorteil der Dynamik besteht darin, einen neuen Typ zum Behandeln hinzuzufügen, wobei nur eine Überladung von DebugToStringInternal des neuen Typs erforderlich ist. Es entfällt auch die Notwendigkeit, den Typ manuell zu erstellen.

Dynamischer Typ online lesen: https://riptutorial.com/de/csharp/topic/762/dynamischer-typ

Kapitel 45: Eigene MessageBox in Windows Form Application erstellen

Einführung

Zuerst müssen wir wissen, was eine MessageBox ist ...

Das MessageBox-Steuerelement zeigt eine Nachricht mit dem angegebenen Text an und kann durch Angabe eines benutzerdefinierten Bilds, Titels und Schaltflächensets angepasst werden (Diese Tastensätze ermöglichen es dem Benutzer, mehr als eine einfache Ja / Nein-Antwort auszuwählen).

Durch Erstellen unserer eigenen MessageBox können Sie das MessageBox-Steuerelement in neuen Anwendungen wiederverwenden, indem Sie einfach die generierte DLL verwenden oder die Datei mit der Klasse kopieren.

Syntax

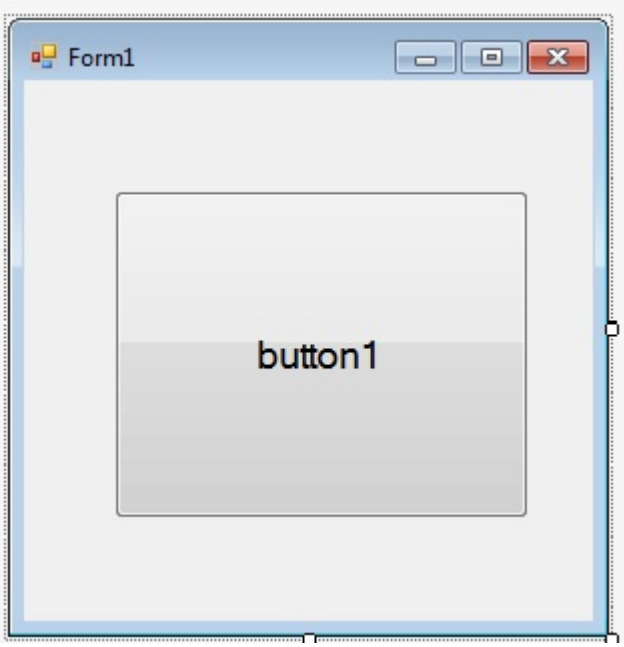
- 'statisches DialogResult Ergebnis = DialogResult.No; // DialogResult wird nach der Entlassung von Dialogen zurückgegeben. '

Examples

Eigenes MessageBox-Steuerelement erstellen.

Um unser eigenes MessageBox-Steuerelement zu erstellen, folgen Sie einfach der folgenden Anleitung ...

1. Öffnen Sie Ihre Instanz von Visual Studio (VS 2008/2010/2012/2015/2017).
2. Gehen Sie zur Symbolleiste oben und klicken Sie auf Datei -> Neues Projekt -> Windows Forms-Anwendung -> Geben Sie dem Projekt einen Namen und klicken Sie dann auf OK.
3. Ziehen Sie nach dem Laden ein Button-Steuerelement aus der Toolbox (links) auf das Formular (wie unten gezeigt).



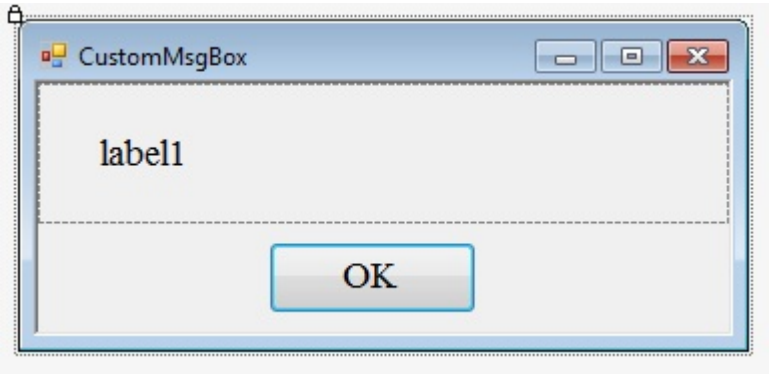
4. Doppelklicken Sie auf die Schaltfläche. Die Integrierte Entwicklungsumgebung generiert automatisch den Click-Ereignishandler für Sie.

5. Bearbeiten Sie den Code für das Formular, sodass es wie folgt aussieht (Sie können mit der rechten Maustaste auf das Formular klicken und auf Code bearbeiten klicken):

```
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. Projektmappen-Explorer -> Klicken Sie mit der rechten Maustaste auf Ihr Projekt -> Hinzufügen -> Windows Form, und legen Sie den Namen "CustomMsgBox.cs" fest.

7. Ziehen Sie ein Schaltflächen- und Beschriftungssteuerelement aus der Toolbox in das Formular (es wird ungefähr so aussehen wie das folgende Formular):



8. Schreiben Sie nun den folgenden Code in das neu erstellte Formular:

```
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. Starten Sie nun das Programm, indem Sie einfach die Taste F5 drücken. Glückwunsch, Sie haben eine wiederverwendbare Kontrolle vorgenommen.

Wie Verwenden eines eigenen MessageBox-Steuerelements in einer anderen Windows Form-Anwendung

Um Ihre vorhandenen CS-Dateien zu finden, klicken Sie mit der rechten Maustaste auf das Projekt in Ihrer Instanz von Visual Studio, und klicken Sie im Datei-Explorer auf Ordner öffnen.

1. Visual Studio -> Ihr aktuelles Projekt (Windows Form) -> Projektmappen-Explorer -> Projektname -> Rechtsklick -> Hinzufügen -> Vorhandenes Element -> Suchen Sie dann Ihre vorhandene CS-Datei.

2. Jetzt muss noch ein letztes Mal getan werden, um das Steuerelement zu verwenden. Fügen Sie Ihrem Code eine using-Anweisung hinzu, damit Ihre Assembly über ihre Abhängigkeiten Bescheid weiß.

   ```
   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. Um die Messagebox anzuzeigen, verwenden Sie einfach folgendes ...

   CustomMsgBox.Show ("Ihre Nachricht für das Meldungsfeld ...", "MSG", "OK");

Eigene MessageBox in Windows Form Application erstellen online lesen: https://riptutorial.com/de/csharp/topic/9788/eigene-messagebox-in-windows-form-application-erstellen

Kapitel 46: Eigenschaften

Bemerkungen

Eigenschaften kombinieren die Klassendatenspeicherung von Feldern mit der Zugänglichkeit von Methoden. Manchmal kann es schwierig sein, zu entscheiden, ob eine Eigenschaft verwendet wird, eine Eigenschaft, die auf ein Feld verweist, oder eine Methode, die auf ein Feld verweist. Als Faustregel gilt:

- Eigenschaften sollten ohne internes Feld verwendet werden, wenn sie nur Werte erhalten und / oder festlegen. ohne dass eine andere Logik auftritt. In solchen Fällen würde das Hinzufügen eines internen Felds das Hinzufügen von Code ohne Nutzen bedeuten.
- Eigenschaften sollten mit internen Feldern verwendet werden, wenn Sie die Daten bearbeiten oder validieren müssen. Ein Beispiel ist das Entfernen von führenden und nachgestellten Leerzeichen aus Zeichenfolgen oder das Sicherstellen, dass ein Datum nicht in der Vergangenheit liegt.

Im Hinblick auf die Methoden vs Eigenschaften, wo Sie beide (abrufen `get`) und update (`set`) einen Wert, ist eine Eigenschaft , die bessere Wahl. .Net bietet außerdem eine Vielzahl von Funktionen, die die Struktur einer Klasse nutzen. B. ein Raster zu einem Formular hinzufügen, listet .Net standardmäßig alle Eigenschaften der Klasse in diesem Formular auf; Um solche Konventionen bestmöglich zu nutzen, sollten Sie Eigenschaften verwenden, wenn dieses Verhalten normalerweise wünschenswert ist, und Methoden, bei denen Sie bevorzugen, dass die Typen nicht automatisch hinzugefügt werden.

Examples

Verschiedene Eigenschaften im Kontext

```
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;
    }
}
```

Öffentlich erhalten

Getter werden verwendet, um Werte aus Klassen anzuzeigen.

```
string name;
public string Name
{
    get { return this.name; }
}
```

Öffentliches Set

Setter werden verwendet, um den Eigenschaften Werte zuzuweisen.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

Auf Eigenschaften zugreifen

```
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)){}
}
```

Standardwerte für Eigenschaften

Das Festlegen eines Standardwerts kann mithilfe der Initialisierer (C # 6) erfolgen

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

Wenn es nur gelesen wird, können Sie Werte wie folgt zurückgeben:

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
  }
```

Automatisch implementierte Eigenschaften

Automatisch implementierte Eigenschaften wurden in C # 3 eingeführt.
Eine automatisch implementierte Eigenschaft wird mit einem leeren Getter und Setter (Accessor) deklariert:

```
public bool IsValid { get; set; }
```

Wenn eine automatisch implementierte Eigenschaft in Ihren Code geschrieben wird, erstellt der Compiler ein anonymes privates Feld, auf das nur über die Zugriffsmethoden der Eigenschaft zugegriffen werden kann.

Die obige automatisch implementierte Eigenschaftsanweisung entspricht dem Schreiben dieses langen Codes:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

Automatisch implementierte Eigenschaften können keine Logik in ihren Zugriffsmethoden haben, zum Beispiel:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

Eine automatisch implementierte Eigenschaft *kann* jedoch unterschiedliche Zugriffsmodifizierer für ihre Zugriffsmethoden haben:

```
public bool IsValid { get; private set; }
```

In C # 6 können automatisch implementierte Eigenschaften keinen Setter enthalten (dies macht sie unveränderlich, da ihr Wert nur innerhalb des Konstruktors festgelegt oder hart codiert werden kann):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

Weitere Informationen zum Initialisieren von automatisch implementierten Eigenschaften finden Sie in der Dokumentation der Auto-Property-Initialisierer .

Schreibgeschützte Eigenschaften

Erklärung

Ein häufiges Missverständnis, insbesondere für Anfänger, ist die schreibgeschützte Eigenschaft, die mit `readonly` Schlüsselwort `readonly` gekennzeichnet ist. Das ist nicht korrekt und in der Tat *ist ein Fehler bei der Kompilierung* :

```
public readonly string SomeProp { get; set; }
```

Eine Eigenschaft ist schreibgeschützt, wenn sie nur über einen Getter verfügt.

```
public string SomeProp { get; }
```

Verwenden von schreibgeschützten Eigenschaften zum Erstellen unveränderlicher Klassen

```
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;
    }
}
```

Eigenschaften online lesen: https://riptutorial.com/de/csharp/topic/49/eigenschaften

Kapitel 47: Eigenschaften initialisieren

Bemerkungen

Beginnen Sie bei der Entscheidung, wie eine Eigenschaft erstellt werden soll, mit einer automatisch implementierten Eigenschaft, um Einfachheit und Kürze zu erreichen.

Wechseln Sie zu einer Eigenschaft mit einem Hintergrundfeld nur dann, wenn die Umstände dies erfordern. Wenn Sie andere Manipulationen benötigen, die über ein einfaches Set hinausgehen, erhalten Sie möglicherweise ein Hintergrundfeld.

Examples

C # 6.0: Initialisieren Sie eine automatisch implementierte Eigenschaft

Erstellen Sie eine Eigenschaft mit Getter und / oder Setter und initialisieren Sie alles in einer Zeile:

```
public string Foobar { get; set; } = "xyz";
```

Eigenschaft mit einem Sicherungsfeld initialisieren

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

Eigenschaft im Konstruktor initialisieren

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

Eigenschaftsinitialisierung während der Objektinstanziierung

Eigenschaften können festgelegt werden, wenn ein Objekt instanziiert wird.

```
var redCar = new Car
{
    Wheels = 2,
```

```
    Year = 2016,
    Color = Color.Red
};
```

Eigenschaften initialisieren online lesen: https://riptutorial.com/de/csharp/topic/82/eigenschaften-initialisieren

Kapitel 48: Eine Übersicht über c # - Kollektionen

Examples

HashSet

Dies ist eine Sammlung von Unikaten mit O (1) -Suchfunktion.

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

Zum Vergleich führt die Verwendung von " `Contains` eine Liste" zu einer schlechteren Leistung:

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

`HashSet.Contains` verwendet eine Hashtabelle, sodass `HashSet.Contains` unabhängig von der Anzahl der Elemente in der Auflistung extrem schnell sind.

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 [] (Array von T)

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

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

```
}

// output:
// one
// two

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


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

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

Liste

`List<T>` ist eine Liste eines bestimmten Typs. Elemente können hinzugefügt, eingefügt, entfernt und über den Index adressiert werden.

```
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>` kann als Array betrachtet werden, dessen Größe Sie ändern können. Die Auflistung der Auflistung in der Reihenfolge ist schnell, ebenso wie der Zugriff auf einzelne Elemente über ihren Index. Um auf Elemente zuzugreifen, die auf einem bestimmten Aspekt ihres Wertes oder auf einem anderen Schlüssel basieren, ermöglicht ein `Dictionary<T>` eine schnellere Suche.

Wörterbuch

Wörterbuch <TKey, TValue> ist eine Karte. Für einen bestimmten Schlüssel kann es einen Wert im Wörterbuch geben.

```
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);
}
```

Doppelter Schlüssel bei der Sammlungsinitialisierung

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

Stapel

```
// 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
```

Beachten Sie, dass `LinkedList<T>` die *doppelt* verknüpfte Liste darstellt. Es handelt sich also einfach um eine Sammlung von Knoten, und jeder Knoten enthält ein Element vom Typ `T` Jeder Knoten ist mit dem vorhergehenden und dem folgenden Knoten verbunden.

Warteschlange

```
// 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
```

> Sichere Köpfe hochziehen! Verwenden Sie ConcurrentQueue in Umgebungen mit mehreren Threads.

Eine Übersicht über c # -Kollektionen online lesen:
https://riptutorial.com/de/csharp/topic/2344/eine-ubersicht-uber-c-sharp--kollektionen

Kapitel 49: Einen variablen Thread sicher machen

Examples

Steuern des Zugriffs auf eine Variable in einer Parallel.For-Schleife

```
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 );
    }
}
```

Es reicht nicht aus, nur `sum = sum + i` ohne die Sperre auszuführen, da die Lese-, Änderungs- und Schreiboperation nicht atomar ist. Ein Thread überschreibt alle externen Änderungen in `sum` , die auftreten, nachdem der aktuelle Wert von `sum` gelesen wurde. Der modifizierte Wert von `sum + i` wieder in `sum` .

Einen variablen Thread sicher machen online lesen:
https://riptutorial.com/de/csharp/topic/4140/einen-variablen-thread-sicher-machen

Kapitel 50: Einfädeln

Bemerkungen

Ein **Thread** ist ein Teil eines Programms, das unabhängig von anderen Teilen ausgeführt werden kann. Es kann Aufgaben gleichzeitig mit anderen Threads ausführen. **Multithreading** ist eine Funktion, mit der Programme eine gleichzeitige Verarbeitung durchführen können, sodass jeweils mehr als ein Vorgang ausgeführt werden kann.

Sie können zum Beispiel Threading verwenden, um einen Timer oder Zähler im Hintergrund zu aktualisieren, während Sie gleichzeitig andere Aufgaben im Vordergrund ausführen.

Multithread-Anwendungen reagieren besser auf Benutzereingaben und sind auch leicht skalierbar, da der Entwickler Threads hinzufügen kann, wenn die Arbeitslast steigt.

Standardmäßig verfügt ein C # -Programm über einen Thread - den Hauptprogramm-Thread. Sekundäre Threads können jedoch erstellt und verwendet werden, um Code parallel zum primären Thread auszuführen. Solche Threads werden Arbeitsthreads genannt.

Um den Betrieb eines Threads zu steuern, delegiert die CLR eine Funktion an das als Thread Scheduler bekannte Betriebssystem. Ein Thread-Scheduler stellt sicher, dass allen Threads die richtige Ausführungszeit zugewiesen wird. Außerdem wird überprüft, ob die blockierten oder gesperrten Threads nicht viel CPU-Zeit beanspruchen.

Der .NET Framework `System.Threading` Namespace `System.Threading` die Verwendung von Threads. System.Threading ermöglicht Multithreading durch Bereitstellung einer Reihe von Klassen und Schnittstellen. Neben dem Bereitstellen von Typen und Klassen für einen bestimmten Thread werden auch Typen definiert, in denen eine Sammlung von Threads, eine Timerklasse usw. gespeichert werden. Es bietet auch Unterstützung, indem es den synchronisierten Zugriff auf gemeinsam genutzte Daten ermöglicht.

`Thread` ist die Hauptklasse im `System.Threading` Namespace. Andere Klassen umfassen `AutoResetEvent` , `Interlocked` , `Monitor` , `Mutex` und `ThreadPool` .

Zu den Delegaten, die im `System.Threading` Namespace vorhanden sind, gehören `ThreadStart` , `TimerCallback` und `WaitCallback` .

Aufzählungen im `System.Threading` Namespace umfassen `ThreadPriority` , `ThreadState` und `EventResetMode` .

In .NET Framework 4 und `System.Threading.Tasks.Parallel` Versionen wird die Multithread-Programmierung durch die Klassen `System.Threading.Tasks.Parallel` und `System.Threading.Tasks.Task` , Parallel LINQ (PLINQ), neue parallele Auflistungsklassen in `System.Collections.Concurrent` einfacher und einfacher `System.Collections.Concurrent` Namespace und ein neues aufgabenbasiertes Programmiermodell.

Examples

Einfache vollständige Threading-Demo

```
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));
        }
    }
}
```

Einfache vollständige Threading-Demo mit Aufgaben

```
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));
        }
    }
}
```

Expliziter Aufgabenparallismus

```
    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();
    }
```

Implizite Aufgabenparallelität

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
            () => b.DoSomeOtherWork()
            );

      }
```

Einen zweiten Thread erstellen und starten

Wenn Sie mehrere lange Berechnungen durchführen, können Sie sie gleichzeitig in verschiedenen Threads auf Ihrem Computer ausführen. Dazu erstellen wir einen neuen **Thread** und weisen auf eine andere Methode hin.

```
using System.Threading;
```

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

Einen Thread mit Parametern starten

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);
    }
}
```

Einen Thread pro Prozessor erstellen

> `Environment.ProcessorCount` Ruft die Anzahl der **logischen** Prozessoren auf der aktuellen Maschine ab.

Die CLR plant dann jeden Thread zu einem logischen Prozessor. Dies könnte theoretisch jeden Thread auf einem anderen logischen Prozessor, alle Threads auf einem einzelnen logischen Prozessor oder eine andere Kombination bedeuten.

```
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);
    }
}
```

Gleichzeitiges Lesen und Schreiben von Daten vermeiden

Manchmal möchten Sie, dass Ihre Threads gleichzeitig Daten freigeben. In diesem Fall ist es wichtig, den Code zu kennen und alle Teile zu sperren, die möglicherweise schief gehen. Ein einfaches Beispiel für die Zählung von zwei Threads ist unten dargestellt.

Hier ist ein gefährlicher (falscher) Code:

```
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);
        }
    }
}
```

Sie werden bemerken, anstatt 1,2,3,4,5 zu zählen ... wir zählen 1,1,2,2,3 ...

Um dieses Problem zu beheben, müssen wir den Wert von count **sperren** , damit mehrere verschiedene Threads nicht gleichzeitig lesen und schreiben können. Durch das Hinzufügen einer Sperre und eines Schlüssels können wir verhindern, dass die Threads gleichzeitig auf die Daten zugreifen.

```
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-Schleife

Wenn Sie eine foreach-Schleife haben, die Sie beschleunigen möchten, und es Ihnen nichts ausmacht, in welcher Reihenfolge sich die Ausgabe befindet, können Sie sie in eine parallele foreach-Schleife konvertieren.

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

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

Deadlocks (zwei Threads warten aufeinander)

Ein Deadlock tritt auf, wenn zwei oder mehr Threads darauf warten, dass sich eine Ressource fertigstellt oder freigibt, sodass sie für immer warten.

Ein typisches Szenario, in dem zwei Threads auf den Abschluss warten, ist, wenn ein Windows Forms-GUI-Thread auf einen Arbeitsthread wartet und der Arbeitsthread versucht, ein vom GUI-Thread verwaltetes Objekt aufzurufen. Beachten Sie, dass bei diesem Code-Beispiel das Klicken auf button1 dazu führt, dass das Programm hängen bleibt.

```
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()` ist ein Aufruf, der den aufrufenden Thread blockiert, bis workerthread abgeschlossen ist. `textBox1.Invoke(invoke_delegate)` ist ein Aufruf, der den aufrufenden Thread blockiert, bis der GUI-Thread invoke_delegate verarbeitet hat. Dieser Aufruf verursacht jedoch Deadlocks, wenn der GUI-Thread bereits auf den Abschluss des aufrufenden Threads wartet.

Um dies zu umgehen, können Sie stattdessen eine nicht blockierende Methode zum Aufrufen des Textfelds verwenden:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

Dies führt jedoch zu Problemen, wenn Sie Code ausführen müssen, der vom Textfeld abhängt, das zuerst aktualisiert wird. In diesem Fall führen Sie das als Teil des Aufrufs aus. Beachten Sie jedoch, dass dies auf dem GUI-Thread ausgeführt wird.

```
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
```

```
}
```

Starten Sie alternativ einen ganzen neuen Thread, und warten Sie auf den GUI-Thread, damit der Arbeitsthread möglicherweise abgeschlossen wird.

```
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
}
```

Vermeiden Sie, wenn möglich, Zirkelverweise zwischen Threads, um das Risiko eines gegenseitigen Wartens zu minimieren. Eine Hierarchie von Threads, in denen untergeordnete Threads nur Nachrichten für übergeordnete Threads hinterlassen und nicht auf sie warten, wird nicht zu diesem Problem führen. Es ist jedoch immer noch anfällig für Deadlocks, die auf dem Sperren von Ressourcen basieren.

Deadlocks (Ressource halten und warten)

Ein Deadlock tritt auf, wenn zwei oder mehr Threads darauf warten, dass sich eine Ressource fertigstellt oder freigibt, sodass sie für immer warten.

Wenn Thread1 eine Sperre für Ressource A hält und auf die Freigabe der Ressource B wartet, während Thread2 die Ressource B enthält und auf die Freigabe der Ressource A wartet, sind sie blockiert.

Durch Klicken auf button1 für den folgenden Beispielcode wird Ihre Anwendung in den oben genannten Deadlock-Zustand versetzt und hängt

```
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#";
            }
        }
    }
}
```

Um auf diese Weise nicht blockiert zu werden, können Sie Monitor.TryEnter (lock_object, timeout_in_milliseconds) verwenden, um zu prüfen, ob bereits ein Objekt gesperrt ist. Wenn es Monitor.TryEnter nicht gelingt, vor timeout_in_milliseconds eine Sperre für lock_object zu erlangen, gibt es false zurück, wodurch der Thread die Möglichkeit erhält, andere angehaltene Ressourcen freizugeben und nachgibt, wodurch andere Threads die Möglichkeit erhalten, wie in dieser leicht modifizierten Version des obigen Beispiels abzuschließen :

```
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#";
            }
        }
    }
}
```

Beachten Sie, dass diese Problemumgehung darauf beruht, dass Thread2 hartnäckig ist, weil seine Sperren und Thread1 nachgeben wollen, so dass Thread2 immer Vorrang hat. Beachten Sie auch, dass Thread1 die Arbeit, die er nach dem Sperren der Ressource A ausgeführt hat, wiederholen muss, wenn er nachgibt. Seien Sie daher vorsichtig, wenn Sie diesen Ansatz mit mehr als einem nachgiebigen Thread implementieren, da Sie Gefahr laufen, in einen sogenannten

Livelock einzutreten - ein Zustand, der eintreten würde, wenn zwei Threads den ersten Teil ihrer Arbeit erledigen und sich dann gegenseitig nachgeben , wiederholt wiederholt.

Einfädeln online lesen: https://riptutorial.com/de/csharp/topic/51/einfadeln

Kapitel 51: Eingebaute Typen

Examples

Unveränderlicher Referenztyp - Zeichenfolge

```
// 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);
    }
}
```

Werttyp - Zeichen

```
// 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';
```

Werttyp - short, int, long (vorzeichenbehaftete 16-Bit-, 32-Bit-, 64-Bit-Ganzzahlen)

```
// 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;
```

Es ist auch möglich, diese Typen auf Null zu setzen, was bedeutet, dass zusätzlich zu den üblichen Werten auch Null zugewiesen werden kann. Wenn eine Variable mit einem nullwertfähigen Typ nicht initialisiert wird, ist sie null statt 0. Nullwertfähige Typen werden markiert, indem ein Fragezeichen (?) Nach dem Typ hinzugefügt wird.

```
int a; //This is now 0.
int? b; //This is now null.
```

Werttyp - ushort, uint, ulong (vorzeichenlose 16-Bit-, 32-Bit-, 64-Bit-Ganzzahlen)

```
// 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;
```

Es ist auch möglich, diese Typen auf Null zu setzen, was bedeutet, dass zusätzlich zu den üblichen Werten auch Null zugewiesen werden kann. Wenn eine Variable mit einem nullwertfähigen Typ nicht initialisiert wird, ist sie null statt 0. Nullwertfähige Typen werden markiert, indem ein Fragezeichen (?) Nach dem Typ hinzugefügt wird.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Werttyp - 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");
```

```
}
```

Das Schlüsselwort bool ist ein Alias von System.Boolean. Es wird verwendet, um Variablen zu deklarieren, um die booleschen Werte `true` und `false` zu speichern.

Vergleiche mit geschachtelten Werttypen

Wenn Werttypen Variablen des Typs `object` zugewiesen werden, werden sie in *Kästchen gesetzt* - der Wert wird in einer Instanz eines `System.Object` gespeichert. Dies kann zu unbeabsichtigten Folgen beim Vergleich von Werten mit `==` , zB:

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

Dies kann durch die Verwendung der überladenen `Equals` Methode vermieden werden, die das erwartete Ergebnis liefert.

```
var comparison2 = left.Equals(right); // true
```

Alternativ können Sie auch die `left` und `right` Variable entpacken, um die `int` Werte zu vergleichen:

```
var comparison3 = (int)left == (int)right; // true
```

Konvertierung von geschachtelten Werttypen

Boxed- Value-Typen können nur in ihrem ursprünglichen `Type` entpackt werden, selbst wenn eine Konvertierung der beiden `Type` Werte gültig ist, z.

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

Dies kann vermieden werden, indem Sie zunächst den ursprünglichen `Type` entpacken, z. B .:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Eingebaute Typen online lesen: https://riptutorial.com/de/csharp/topic/42/eingebaute-typen

Kapitel 52: Einschließlich Font-Ressourcen

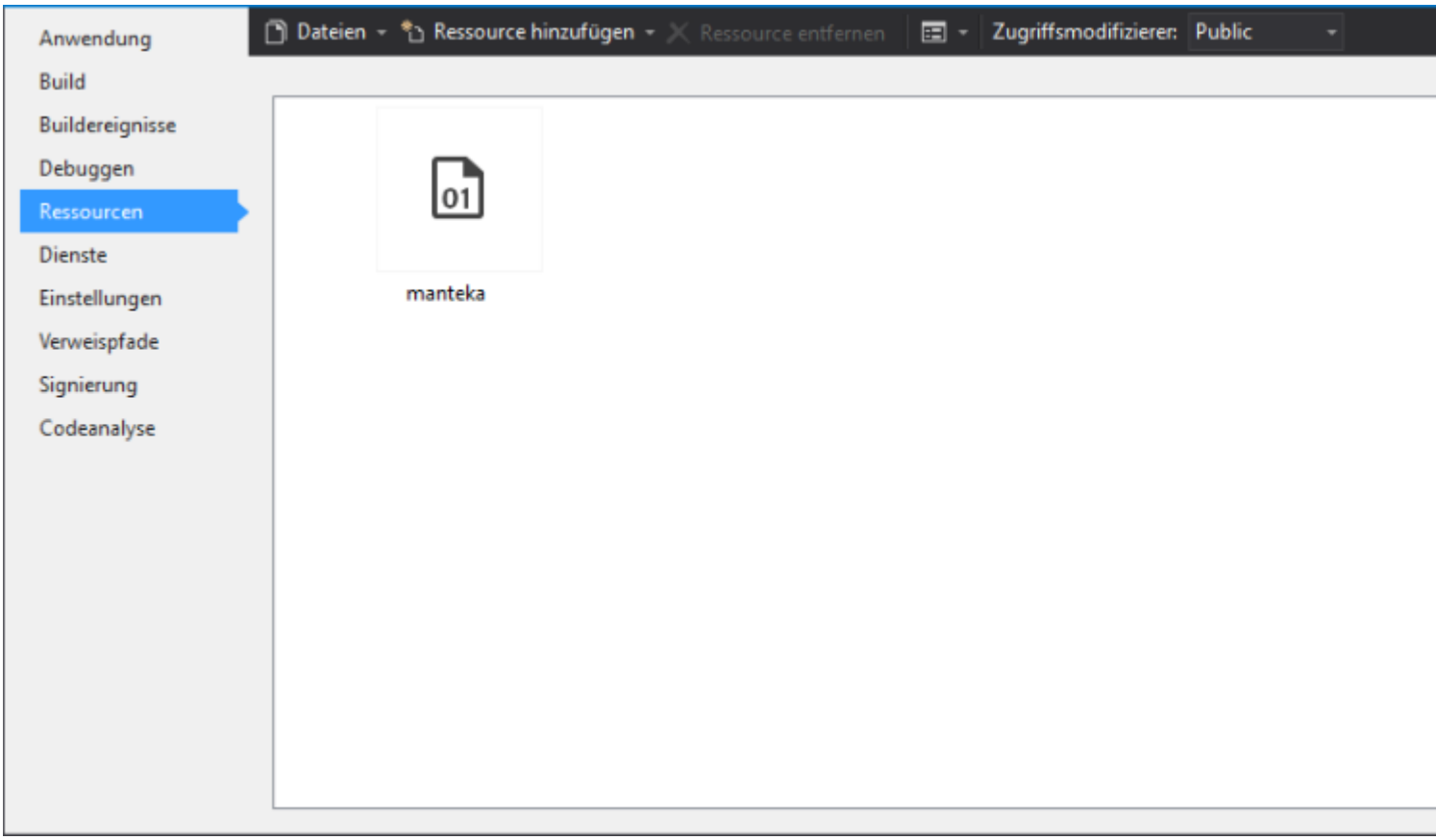
Parameter

Parameter	Einzelheiten
fontbytes	Byte-Array aus der binären .ttf

Examples

Instantiate 'Fontfamily' von Resources

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Integrationsmethode

```
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];
}
```

Verwendung mit einem 'Button'

```
    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);
        }
    }
```

Einschließlich Font-Ressourcen online lesen: https://riptutorial.com/de/csharp/topic/9789/einschlie-lich-font-ressourcen

Kapitel 53: Enum

Einführung

Eine Enumeration kann von einer der folgenden Arten abgeleitet werden: Byte, Byte, Kurz, Ushort, Int, Uint, Long, Ulong. Der Standardwert ist int und kann durch Angabe des Typs in der Enumendefinition geändert werden:

public enum Wochentag: Byte {Montag = 1, Dienstag = 2, Mittwoch = 3, Donnerstag = 4, Freitag = 5}

Dies ist nützlich, wenn Sie P / Invoke zu nativem Code aufrufen, Datenquellen zuordnen und ähnliche Umstände. Im Allgemeinen sollte der Standardwert int verwendet werden, da die meisten Entwickler davon ausgehen, dass es sich bei der Aufzählung um ein int handelt.

Syntax

- Aufzählungsfarben {Red, Green, Blue} // Enum-Deklaration
- enum Farben: Byte {Rot, Grün, Blau} // Deklaration mit spezifischem Typ
- Aufzählungsfarben {Rot = 23, Grün = 45, Blau = 12} // Deklaration mit definierten Werten
- Colors.Red // Greifen Sie auf ein Enum-Element zu
- int value = (int) Colors.Red // Liefert den Int-Wert eines Enumerationselements
- Colors color = (Colors) intValue // Holen Sie ein Enumerationselement aus int

Bemerkungen

Eine Aufzählung (Abkürzung für "Aufzählungstyp") ist ein Typ, der aus einer Menge benannter Konstanten besteht, die durch einen typspezifischen Bezeichner dargestellt werden.

Aufzählungen sind am nützlichsten, um Konzepte darzustellen, die eine (normalerweise kleine) Anzahl möglicher diskreter Werte haben. Sie können beispielsweise verwendet werden, um einen Wochentag oder einen Monat des Jahres darzustellen. Sie können auch als Flags verwendet werden, die mit bitweisen Operationen kombiniert oder geprüft werden können.

Examples

Holen Sie sich alle Mitgliederwerte einer Aufzählung

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
```

```
    Console.WriteLine(e);
```

Dies wird drucken:

```
One
Two
Three
```

Aufzählung als Flaggen

Das `FlagsAttribute` kann auf ein Enum angewendet werden, das das Verhalten von `ToString()` , um es an die Art des `ToString()` :

```
[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);
```

Da `FlagsAttribute` die Aufzählungskonstanten Potenzen von zwei (oder deren Kombinationen) sind und die Aufzählungswerte letztendlich numerische Werte sind, sind Sie durch die Größe des zugrunde liegenden numerischen Typs begrenzt. Der größte verfügbare numerische Typ, den Sie verwenden können, ist `UInt64` , mit dem Sie 64 verschiedene (nicht kombinierte) Flag- `UInt64` angeben können. Das `enum` Schlüsselwort verwendet standardmäßig den zugrunde liegenden Typ `int` (`Int32` . Der Compiler erlaubt die Deklaration von Werten, die breiter als 32 Bit sind. Diese werden ohne Vorwarnung umgebrochen und führen zu zwei oder mehr Enumerationsmitgliedern mit demselben Wert. Wenn eine Aufzählung ein Bitset mit mehr als 32 Flags enthalten soll, müssen Sie daher explizit einen größeren Typ angeben:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Obwohl Flags oft nur ein einzelnes Bit sind, können sie zur einfacheren Verwendung in benannten "Sets" kombiniert werden.

```
[Flags]
enum FlagsEnum
{
    None = 0,
```

```
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Um zu vermeiden, dass die Dezimalwerte von Zweierpotenzen ausgegeben werden, kann der Operator für die linke Umschalttaste (<<) auch verwendet werden, um dieselbe Aufzählung zu deklarieren

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Ab C # 7.0 können auch binäre Literale verwendet werden.

Um zu überprüfen, ob für den Wert der Aufzählungsvariablen ein bestimmtes Flag gesetzt ist, kann die `HasFlag` Methode verwendet werden. Sagen wir, wir haben

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

Und einen `value`

```
var value = MyEnum.One | MyEnum.Two;
```

Mit `HasFlag` wir überprüfen, ob eines der Flags gesetzt ist

```
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");
```

Außerdem können wir alle Werte von enum durchlaufen, um alle gesetzten Flags zu erhalten

```
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);
}
```

Oder

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

Alle drei Beispiele werden gedruckt:

```
Enum has One
Enum has Two
```

Testen Sie Aufzählungswerte im Flags-Stil mit bitweiser Logik

Ein Aufzählungswert im Flags-Stil muss mit bitweiser Logik getestet werden, da er möglicherweise keinem einzelnen Wert entspricht.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

Der `Default` ist eigentlich eine Kombination aus zwei anderen, *die* mit einem bitweisen ODER verknüpft sind. Um das Vorhandensein einer Flagge zu testen, müssen wir ein bitweises UND verwenden.

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

Aufzählung zu String und zurück

```
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));
```

Standardwert für Aufzählung == NULL

Der Standardwert für eine Aufzählung ist Null . Wenn eine Aufzählung keinen Artikel mit dem Wert Null definiert, ist der Standardwert Null.

```
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");
    }
}
```

Beispiel: https://dotnetfiddle.net/l5Rwie

Enum-Grundlagen

Von MSDN :

> Ein Aufzählungstyp (auch als Aufzählung oder Aufzählung bezeichnet) bietet eine effiziente Methode zum Definieren einer Menge von benannten **Integralkonstanten** , die **einer Variablen zugewiesen werden können** .

Im Wesentlichen ist eine Aufzählung ein Typ, der nur einen Satz endlicher Optionen zulässt, und jede Option entspricht einer Zahl. Standardmäßig steigen diese Zahlen in der Reihenfolge, in der die Werte deklariert werden, beginnend mit Null. Zum Beispiel könnte man eine Aufzählung für die Wochentage festlegen:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Dieses Enum könnte folgendermaßen verwendet werden:

```
// 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;
```

Standardmäßig ist der zugrunde liegende Typ jedes Elements in der `enum` `int` , aber auch `byte` , `sbyte` , `short` , `ushort` , `uint` , `long` und `ulong` können verwendet werden. Wenn Sie einen anderen Typ als `int` , müssen Sie den Typ mit einem Doppelpunkt nach dem Aufzählungsnamen angeben:

```
public enum Day : byte
{
```

```
    // same as before
}
```

Die Zahlen hinter dem Namen sind jetzt Bytes anstelle von ganzen Zahlen. Sie können den zugrunde liegenden Typ der Aufzählung folgendermaßen erhalten:

```
Enum.GetUnderlyingType(typeof(Days));
```

Ausgabe:

```
System.Byte
```

Demo: .NET Geige

Bitweise Manipulation über Enummen

Das FlagsAttribute sollte immer dann verwendet werden, wenn das Aufzählungszeichen eine Sammlung von Flags und nicht einen einzelnen Wert darstellt. Der numerische Wert, der jedem Aufzählungswert zugewiesen wird, erleichtert die Bearbeitung von Aufzählungen mit bitweisen Operatoren.

Beispiel 1: Mit [Flags]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

druckt Rot, Blau

Beispiel 2: Ohne [Flags]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

druckt 3

Die << -Notation für Flags verwenden

Der Linksverschiebungsoperator (`<<`) kann in Flag-Enumendeklarationen verwendet werden, um sicherzustellen, dass jedes Flag in binärer Darstellung genau eine `1` , wie es Flags sollten.

Dies hilft auch, die Lesbarkeit großer Aufzählungen mit vielen Flags zu verbessern.

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

Es ist offensichtlich, dass `MyEnum` nur richtige Flaggen enthält und keine unordentlichen `Flag30 = 1073741822` wie `Flag30 = 1073741822` (oder 11111111111111111111111111110 in binär), was ungeeignet ist.

Hinzufügen zusätzlicher Beschreibungsinformationen zu einem Aufzählungswert

In einigen Fällen möchten Sie möglicherweise eine zusätzliche Beschreibung zu einem Aufzählungswert hinzufügen, beispielsweise wenn der Aufzählungswert selbst weniger lesbar ist als der, den Sie dem Benutzer anzeigen möchten. In solchen Fällen können Sie die `System.ComponentModel.DescriptionAttribute` Klasse verwenden.

Zum Beispiel:

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

Wenn Sie nun die Beschreibung eines bestimmten Aufzählungswerts zurückgeben möchten, können Sie Folgendes tun:

```
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"
}
```

Dies kann auch leicht in eine Erweiterungsmethode für alle Enums umgewandelt werden:

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())
typeof(DescriptionAttribute))).Description;
    }
}
```

Und dann einfach wie `Console.WriteLine(result.GetDescription());` verwendet:
`Console.WriteLine(result.GetDescription());`

Hinzufügen und Entfernen von Werten zur markierten Aufzählung

Dieser Code dient zum Hinzufügen und Entfernen eines Wertes aus einer gekennzeichneten Enum-Instanz:

```
[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
```

Aufzählungen können unerwartete Werte enthalten

Da eine Enumeration in und von ihrem zugrunde liegenden Integraltyp umgewandelt werden kann, kann der Wert außerhalb des Wertebereichs liegen, der in der Definition des Enummentyps angegeben ist.

Obwohl unter dem Aufzählungstyp `DaysOfWeek` nur 7 definierte Werte vorhanden sind, kann er trotzdem einen beliebigen `int` Wert enthalten.

```
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
```

Derzeit gibt es keine Möglichkeit, eine Aufzählung zu definieren, die dieses Verhalten nicht aufweist.

`Enum.IsDefined` Aufzählungswerte können jedoch mithilfe der Methode `Enum.IsDefined` . Zum Beispiel,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

Enum online lesen: https://riptutorial.com/de/csharp/topic/931/enum

Kapitel 54: Equals und GetHashCode

Bemerkungen

Jede Implementierung von `Equals` muss die folgenden Anforderungen erfüllen:

- **Reflexiv** : Ein Objekt muss sich selbst gleich sein.
  `x.Equals(x)` gibt `true` .
- **Symmetrisch** : Es gibt keinen Unterschied, wenn ich x mit y oder y mit x vergleiche - das Ergebnis ist das gleiche.
  `x.Equals(y)` liefert den gleichen Wert wie `y.Equals(x)` .
- **Transitiv** : Wenn ein Objekt einem anderen Objekt entspricht und dieses Objekt einem dritten Objekt entspricht, muss das erste Objekt dem dritten Objekt entsprechen.
  wenn `(x.Equals(y) && y.Equals(z)) true` zurückgibt, gibt `x.Equals(z) true` .
- **Konsistent** : Wenn Sie ein Objekt mehrmals mit einem anderen Objekt vergleichen, ist das Ergebnis immer dasselbe.
  Aufeinanderfolgende `x.Equals(y)` von `x.Equals(y)` denselben Wert, solange die von x und y referenzierten Objekte nicht geändert werden.
- **Vergleich mit null** : Kein Objekt ist gleich `null` .
  `x.Equals(null)` gibt `false` .

Implementierungen von `GetHashCode` :

- **Kompatibel mit `Equals`** : Wenn zwei Objekte gleich sind (dh `Equals` gibt true zurück), **muss** `GetHashCode` für jedes `GetHashCode` denselben Wert zurückgeben.
- **Große Reichweite** : Wenn zwei Objekte nicht gleich sind (`Equals` sagt falsch), sollten die Hash-Codes mit **hoher Wahrscheinlichkeit** unterschiedlich sein. *Perfektes* Hashing ist oft nicht möglich, da eine begrenzte Anzahl von Werten zur Auswahl steht.
- **Günstig** : Die Berechnung des Hash-Codes sollte in allen Fällen kostengünstig sein.

Siehe: Richtlinien zum Überladen von Equals () und Operator ==

Examples

Standard Gleiches Verhalten.

`Equals` wird in der `Object` Klasse selbst deklariert.

```
public virtual bool Equals(Object obj);
```

Standardmäßig hat `Equals` das folgende Verhalten:

- Wenn es sich bei der Instanz um einen Referenztyp handelt, gibt `Equals` nur dann true zurück, wenn die Referenzen identisch sind.
- Wenn es sich bei der Instanz um einen Werttyp handelt, gibt `Equals` nur dann true zurück, wenn Typ und Wert gleich sind.
- `string` ist ein Sonderfall. Es verhält sich wie ein Werttyp.

```
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;
        }
    }
}
```

Eine gute GetHashCode-Überschreibung schreiben

`GetHashCode` hat erhebliche Auswirkungen auf die Leistung von Dictionary <> und HashTable.

Gute `GetHashCode` Methoden

- sollte eine gleichmäßige Verteilung haben
  - Jede ganze Zahl sollte eine ungefähr gleiche Chance haben, für eine zufällige Instanz zurückzukehren
  - Wenn Ihre Methode für jede Instanz dieselbe Ganzzahl zurückgibt (z. B. die Konstante

'999'), haben Sie eine schlechte Leistung
- sollte schnell sein
  - Hierbei handelt es sich NICHT um kryptographische Hashes, bei denen Langsamkeit eine Funktion ist
  - Je langsamer Ihre Hash-Funktion ist, desto langsamer ist Ihr Wörterbuch
- muss denselben HashCode in zwei Instanzen zurückgeben, die `Equals` als true auswertet
  - Wenn dies nicht der `GetHashCode` (z. B. weil `GetHashCode` eine Zufallszahl zurückgibt), werden Elemente möglicherweise nicht in einer `List` , einem `Dictionary` oder ähnlichem gefunden.

Eine gute Methode zum Implementieren von `GetHashCode` ist die Verwendung einer Primzahl als Startwert und das Hinzufügen der Hashcodes der Felder des Typs multipliziert mit anderen Primzahlen.

```
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;
    }
}
```

Nur die Felder, die in der `Equals` Methode verwendet werden, sollten für die Hash-Funktion verwendet werden.

Wenn Sie denselben Typ für Dictionary / HashTables auf unterschiedliche Weise behandeln müssen, können Sie IEqualityComparer verwenden.

Überschreiben Sie Equals und GetHashCode für benutzerdefinierte Typen

Für eine Klasse `Person` wie:

```
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
```

Definieren Sie jedoch `Equals` und `GetHashCode` wie folgt:

```
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
```

`GetHashCode` LINQ verwenden, um verschiedene Abfragen zu Personen `GetHashCode` werden sowohl `Equals` als auch `GetHashCode` :

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

Equals und GetHashCode in IEqualityComparator

Für den angegebenen Typ `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
```

Definieren von `Equals` und `GetHashCode` in einem `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when
comparing two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Beachten Sie, dass für diese Abfrage zwei Objekte als gleich betrachtet wurden, wenn sowohl `Equals` als auch `GetHashCode` denselben Hashcode für die beiden Personen zurückgegeben haben.

Equals und GetHashCode online lesen: https://riptutorial.com/de/csharp/topic/3429/equals-und-gethashcode

Kapitel 55: Erbe

Syntax

- Klasse DerivedClass: BaseClass
- Klasse DerivedClass: BaseClass, IExampleInterface
- Klasse DerivedClass: BaseClass, IExampleInterface, IAnotherInterface

Bemerkungen

Klassen können direkt von nur einer Klasse erben, können jedoch (stattdessen oder gleichzeitig) eine oder mehrere Schnittstellen implementieren.

Strukturen können Schnittstellen implementieren, können jedoch nicht explizit von jedem Typ erben. Sie erben implizit von `System.ValueType` , das wiederum direkt von `System.Object` erbt.

Statische Klassen können keine Schnittstellen implementieren.

Examples

Vererbung von einer Basisklasse

Um das Duplizieren von Code zu vermeiden, definieren Sie allgemeine Methoden und Attribute in einer allgemeinen Klasse als Basis:

```
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()
    {
        // ...
    }
}
```

Da Sie nun eine Klasse haben, die `Animal` im Allgemeinen darstellt, können Sie eine Klasse definieren, die die Besonderheiten bestimmter Tiere beschreibt:

```
public class Cat : Animal
{
    public Cat()
```

```
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}
```

Die Cat-Klasse erhält nicht nur Zugriff auf die explizit beschriebenen Methoden, sondern auch auf alle Methoden, die in der allgemeinen Basisklasse `Animal` sind. Jedes Tier (ob es eine Katze war oder nicht) konnte essen, starren oder rollen. Ein Tier kann jedoch nicht kratzen, es sei denn, es ist auch eine Katze. Sie können dann andere Klassen definieren, die andere Tiere beschreiben. (Wie Gopher mit einer Methode zur Zerstörung von Blumengärten und Faultier ohne zusätzliche Methoden.)

Von einer Klasse erben und eine Schnittstelle implementieren

```
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";
    }
}
```

Von einer Klasse erben und mehrere Schnittstellen implementieren

```
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";
    }
}
```

Vererbung testen und navigieren

```
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
```

Eine abstrakte Basisklasse erweitern

Im Gegensatz zu Schnittstellen, die als Implementierungsverträge bezeichnet werden können, fungieren abstrakte Klassen als Erweiterungsverträge.

Eine abstrakte Klasse kann nicht instanziiert werden, sie muss erweitert werden und die resultierende Klasse (oder abgeleitete Klasse) kann dann instanziiert werden.

Abstrakte Klassen werden verwendet, um generische Implementierungen bereitzustellen

```
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.
}
```

Das obige Beispiel zeigt, wie jede Klasse, die ein Auto erweitert, automatisch die HonkHorn-Methode mit der Implementierung erhält. Dies bedeutet, dass sich jeder Entwickler, der ein neues Auto erstellt, keine Gedanken darüber machen muss, wie es sein Horn hupen wird.

Konstruktoren in einer Unterklasse

Wenn Sie eine Unterklasse einer Basisklasse erstellen, können Sie die Basisklasse mithilfe von `: base` nach den Parametern des Unterklassenkonstruktors erstellen.

```
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;
    }
}
```

Erbe. Reihenfolge der Konstrukteure

Stellen Sie sich vor, wir haben eine Klasse `Animal` die eine Kindklasse `Dog`

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

Standardmäßig erbt jede Klasse implizit die `Object` Klasse.

Dies ist derselbe Code wie oben.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

Beim Erstellen einer Instanz der `Dog` Klasse wird der **Standardkonstruktor** der **Basisklassen (ohne Parameter) aufgerufen, wenn kein expliziter Aufruf eines anderen Konstruktors in der übergeordneten Klasse erfolgt** . In unserem Fall wird zuerst `Object's` Konstruktor, dann `Animal's` und am Ende `Dog's` Constructor genannt.

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

Ausgabe wird sein

> In Animal's Konstruktor
> In Dogs Konstruktor

Demo anzeigen

Rufen Sie den Konstruktor der Eltern explizit auf.

In den obigen Beispielen ruft unser `Dog` Klassenkonstruktor den **Standardkonstruktor** der `Animal` Klasse auf. Wenn Sie möchten, können Sie angeben, welcher Konstruktor aufgerufen werden soll: Es ist möglich, einen beliebigen Konstruktor aufzurufen, der in der übergeordneten Klasse definiert ist.

Betrachten wir diese zwei Klassen.

```
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);
    }
}
```

Was ist hier los?

Wir haben 2 Konstruktoren in jeder Klasse.

Was bedeutet `base` ?

`base` ist eine Referenz auf die übergeordnete Klasse. In unserem Fall, wenn wir eine Instanz der `Dog` Klasse wie folgt erstellen

```
Dog dog = new Dog();
```

Die Laufzeit ruft zuerst `Dog()` , den parameterlosen Konstruktor. Aber sein Körper funktioniert nicht sofort. Nach den Klammern des Konstruktors haben wir einen solchen Aufruf: `base()` , das heißt,

wenn wir den Standard- `Dog` Konstruktor aufrufen, wird dieser wiederum den **Standardkonstruktor** des übergeordneten **Objekts** aufrufen. Nachdem der Konstruktor des übergeordneten Elements ausgeführt wurde, wird der `Dog()` Konstruktorkörper zurückgegeben und anschließend ausgeführt.

So wird die Ausgabe so sein:

> Der Standardkonstruktor von Animal
> Standardkonstruktor des Hundes

Demo anzeigen

Was ist nun, wenn wir den Konstruktor des `Dog's` mit einem Parameter aufrufen?

```
Dog dog = new Dog("Rex");
```

Sie wissen , dass die Mitglieder in der übergeordneten Klasse , die nicht privat sind , werden von der untergeordneten Klasse geerbt, was bedeutet , dass `Dog` auch die haben `name` Feld.
In diesem Fall haben wir unserem Konstruktor ein Argument übergeben. Er seinerseits geht das Argument der übergeordneten Klasse **Konstruktor mit einem Parameter,** der den initialisiert `name` Feld.

Ausgabe wird sein

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

Zusammenfassung:

Jede Objekterstellung beginnt bei der Basisklasse. In der Vererbung werden die Klassen, die sich in der Hierarchie befinden, verkettet. Da alle Klassen von `Object` , ist der erste Konstruktor, der beim Erstellen eines Objekts aufgerufen wird, der `Object` Klassenkonstruktor. Dann wird der nächste Konstruktor in der Kette aufgerufen und erst nachdem alle aufgerufen werden, wird das Objekt erstellt

Basis-Schlüsselwort

1. Das Basisschlüsselwort wird verwendet, um innerhalb einer abgeleiteten Klasse auf Mitglieder der Basisklasse zuzugreifen:
2. Rufen Sie eine Methode für die Basisklasse auf, die von einer anderen Methode überschrieben wurde. Geben Sie an, welcher Basisklassenkonstruktor beim Erstellen von Instanzen der abgeleiteten Klasse aufgerufen werden soll.

Vererbung von Methoden

Es gibt mehrere Möglichkeiten, Methoden zu vererben

```
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
    }
}
```

Vererbung Anti-Muster

Nicht ordnungsgemäße Vererbung

Nehmen wir an, es gibt 2 Klassen `Foo` und `Bar` . `Foo` hat zwei Funktionen `Do1` und `Do2` . `Bar` muss `Do1` von `Foo` , aber es braucht kein `Do2` oder eine Funktion, die `Do2` entspricht, aber etwas völlig anderes macht.

Schlechter Weg : Machen Sie `Do2()` auf `Foo` virtuell, überschreiben Sie es in `Bar` oder `throw Exception` einfach `throw Exception` in `Bar` für `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

Gute Möglichkeit

Nehmen Sie `Do1()` von `Foo` und legen Sie es in die neue Klasse `Baz` Dann erben Sie `Foo` und `Bar` von `Baz` und implementieren Sie `Do2()` separat

```
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
    }
}
```

Nun , warum erstes Beispiel ist schlecht und das zweite ist gut: Wenn Entwickler NR2 eine Änderung zu tun hat `Foo` , stehen die Chancen er Implementierung brechen `Bar` , weil `Bar` jetzt untrennbar ist `Foo` . Bei letzterem Beispiel wurde `Foo` and `Bar` commonalty zu `Baz` und sie beeinflussen sich nicht (wie das sollte).

Basisklasse mit rekursiver Typangabe

Einmalige Definition einer generischen Basisklasse mit rekursivem Typbezeichner. Jeder Knoten hat ein Elternteil und mehrere Kinder.

```
/// <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; } }
}
```

Das Obige kann jedes Mal wiederverwendet werden, wenn eine Baumhierarchie von Objekten definiert werden muss. Das Knotenobjekt in der Baumstruktur muss von der Basisklasse mit erben

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

Jede Knotenklasse weiß, wo sie sich in der Hierarchie befindet, was das übergeordnete Objekt ist und was die untergeordneten Objekte sind. Mehrere eingebaute Typen verwenden eine Baumstruktur wie `Control` oder `XmlElement` Der oben angegebene `Tree<T>` kann als Basisklasse eines *beliebigen* Typs in Ihrem Code verwendet werden.

Um beispielsweise eine Hierarchie von Teilen zu erstellen, bei denen das Gesamtgewicht aus dem Gewicht *aller* untergeordneten Elemente berechnet wird, gehen Sie wie folgt vor:

```
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); }
}
}
```

verwendet werden als

```
// [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;
```

Ein anderes Beispiel wäre die Definition relativer Koordinatenrahmen. In diesem Fall hängt die wahre Position des Koordinatenrahmens von den Positionen *aller* übergeordneten Koordinatenrahmen ab.

```
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));
    }
}
```

verwendet werden als

```
// 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;
```

Erbe online lesen: https://riptutorial.com/de/csharp/topic/29/erbe

Kapitel 56: Ergebnis Keyword

Einführung

Wenn Sie das Ertragsschlüsselwort in einer Anweisung verwenden, geben Sie an, dass die Methode, der Operator oder der Accessor, in dem es angezeigt wird, ein Iterator sind. Durch die Verwendung von Yield zur Definition eines Iterators wird keine explizite zusätzliche Klasse benötigt (die Klasse, die den Status für eine Aufzählung enthält), wenn Sie das IEnumerable- und das IEnumerator-Muster für einen benutzerdefinierten Auflistungstyp implementieren.

Syntax

- Rendite Rendite [TYP]
- Rendite brechen

Bemerkungen

Setzt man die `yield` Schlüsselwort in einem Verfahren mit dem Rückgabetyp `IEnumerable` , `IEnumerable<T>` , `IEnumerator` oder `IEnumerator<T>` weist den Compiler eine Implementierung des Rückgabetypen (zu generieren `IEnumerable` oder `IEnumerator`) , die, wenn geschlungen über läuft die Methode bis zu jeder "Ausbeute", um jedes Ergebnis zu erhalten.

Die `yield` Schlüsselwort ist nützlich , wenn Sie „das nächste“ Element einer theoretisch unbegrenzten Sequenz zurückkehren wollen, so die gesamte Sequenz Berechnung vorher unmöglich wären, oder wenn die vollständige Sequenz von Werten vor der Rückkehr in der Berechnung für den Benutzer zu einer unerwünschten Pause führen würden .

`yield break` kann auch verwendet werden, um die Sequenz jederzeit zu beenden.

Da für das `yield` Schlüsselwort ein Iterator-Schnittstellentyp als Rückgabetyp erforderlich ist, z. B. `IEnumerable<T>` , können Sie dies nicht in einer asynchronen Methode verwenden, da dies ein `Task<IEnumerable<T>>` -Objekt `Task<IEnumerable<T>>` .

Lesen Sie weiter

- https://msdn.microsoft.com/de-de/library/9k7k7cf0.aspx

Examples

Einfache Verwendung

Mit dem Schlüsselwort `yield` wird eine Funktion definiert, die einen `IEnumerable` oder `IEnumerator` (sowie die abgeleiteten generischen Varianten) `IEnumerator` deren Werte `IEnumerator` generiert werden, wenn ein Aufrufer die zurückgegebene Auflistung `IEnumerator` . Lesen Sie mehr über den Zweck in den Anmerkungen .

Das folgende Beispiel enthält eine Anweisung für die Rendite, die sich in einer `for` Schleife befindet.

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Dann kannst du es nennen:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Konsolenausgabe

```
4
5
6
...
14
```

Live-Demo zu .NET-Geige

Bei jeder Iteration des `foreach` Anweisungstextes wird ein Aufruf der `Count` Iterator-Funktion erstellt. Jeder Aufruf der Iterator-Funktion geht zur nächsten Ausführung der `yield return` Anweisung über, die während der nächsten Iteration der `for` Schleife auftritt.

Weitere sachdienliche Verwendung

```
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);
        }
    }
}
```

Es gibt auch andere Möglichkeiten, einen immer `IEnumerable<User>` aus einer SQL - Datenbank, natürlich - dies zeigt nur , dass Sie verwenden können , `yield` etwas zu verwandeln , die „Folge von Elementen“ Semantik in eine hat `IEnumerable<T>` , dass jemand kann über iterieren .

Vorzeitige Beendigung

Sie können die Funktionalität vorhandener `yield` Methoden erweitern, indem Sie einen oder mehrere Werte oder Elemente übergeben, die möglicherweise eine Beendigungsbedingung innerhalb der Funktion definieren, indem Sie einen `yield break` aufrufen, um die Ausführung der inneren Schleife zu stoppen.

```
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++;
    }
}
```

Das oben genannte Verfahren von einer gegebenen iterieren würde `start` , bis einem der Werte innerhalb des `earlyTerminationSet` angetroffen wurde.

```
// 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);
}
```

Ausgabe:

1
2
3
4
5
6

Live-Demo zu .NET-Geige

Argumente richtig prüfen

Eine Iterator-Methode wird erst ausgeführt, wenn der Rückgabewert aufgezählt ist. Es ist daher vorteilhaft, Vorbedingungen außerhalb des Iterators zu behaupten.

```
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;
    }
}
```

Calling Side Code (Verwendung):

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

Ausgabe:

1
2
3
4
5
6
7
8
9
10

Live-Demo zu .NET-Geige

Wenn eine Methode `yield` , um ein Aufzählungszeichen zu generieren, erstellt der Compiler eine Zustandsmaschine, die bei Iteration Code bis zu einem `yield` . Es gibt dann das Ergebnis zurück und speichert seinen Status.

Dies bedeutet, dass Sie beim ersten Aufruf der Methode (da die Statusmaschine erstellt wird) nicht über ungültige Argumente informiert werden (das Übergeben von `null` usw.), nur wenn Sie versuchen, auf das erste Element zuzugreifen (da nur dann der Code innerhalb der Methode wird von der Zustandsmaschine ausgeführt). Durch das Einbetten einer normalen Methode, die zuerst Argumente prüft, können Sie sie beim Aufruf der Methode überprüfen. Dies ist ein Beispiel für ein schnelles Versagen.

Bei Verwendung von C # 7+ kann die `CountCore` Funktion bequem als *lokale Funktion* in der `Count` Funktion ausgeblendet werden. Siehe Beispiel hier .

Gibt ein anderes Enumerable innerhalb einer Methode zurück, die Enumerable zurückgibt

```
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 };
}
```

Faule Bewertung

Erst wenn die `foreach` Anweisung zum nächsten Element wechselt, wird der Iteratorblock bis zur nächsten `yield` ausgewertet.

Betrachten Sie das folgende Beispiel:

```
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);
```

```
    }
}
```

Dies wird ausgegeben:

```
Iteration starten
Innerer Iterator: 0
Innerhalb von foreach: 0
Innerer Iterator: 1
Innenbereich: 1
Innerer Iterator: 2
Innenbereich: 2
```

Demo anzeigen

Als Konsequenz:

- "Iteration starten" wird zuerst gedruckt, obwohl die Iteratormethode vor dem Drucken der Zeile aufgerufen wurde, da die Zeile `Integers().Take(3);` startet die Iteration nicht wirklich (kein Aufruf von `IEnumerator.MoveNext()` wurde durchgeführt)
- Die Zeilen, die auf die Konsole gedruckt werden, wechseln zwischen der Zeile innerhalb der Iterator-Methode und der Zeile innerhalb des `foreach` , anstatt alle Zeilen innerhalb der Iterator-Methode, die zuerst bewertet werden
- Dieses Programm wird aufgrund der `.Take()` -Methode beendet, obwohl die Iterator-Methode eine `while true` der sie niemals ausbricht.

Versuchen Sie es ... endlich

Wenn eine Iterator-Methode innerhalb eines `try...finally` `IEnumerator` eine Rendite hat, führt der zurückgegebene `IEnumerator` die `finally` Anweisung aus, wenn `Dispose` für ihn aufgerufen wird, solange sich der aktuelle Bewertungspunkt im `try` Block befindet.

Angesichts der Funktion:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

Wenn Sie anrufen:

```
private void DisposeOutsideTry()
```

```
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Dann druckt es:

1

Demo anzeigen

Wenn Sie anrufen:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Dann druckt es:

1
2
Endlich ausgeführt

Demo anzeigen

Verwenden von Yield, um einen IEnumerator zu erstellen bei der Implementierung von IEnumerable

Die `IEnumerable<T>` -Schnittstelle hat eine einzige Methode, `GetEnumerator()` , die einen `IEnumerator<T>` .

Während die `yield` Schlüsselwort verwendet werden kann , um direkt eine erstellen `IEnumerable<T>` , kann es *auch* auf genau die gleiche Art und Weise verwendet werden , um eine erstellen `IEnumerator<T>` . Das einzige, was sich ändert, ist der Rückgabetyp der Methode.

Dies kann nützlich sein, wenn wir eine eigene Klasse erstellen möchten, die `IEnumerable<T>` implementiert:

```
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();
    }
}
```

(Beachten Sie, dass dieses spezielle Beispiel nur zur Veranschaulichung dient und mit einer einzigen Iterator-Methode, die ein `IEnumerable<T>` sauberer implementiert werden könnte.)

Eifrige Bewertung

Das `yield` ermöglicht eine faule Bewertung der Sammlung. Das zwangsweise Laden der gesamten Sammlung in den Speicher wird als **eifrige Auswertung bezeichnet** .

Der folgende Code zeigt dies:

```
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();
```

Beim Aufruf von `ToList` , `ToDictionary` oder `ToArray` wird die sofortige Bewertung der Aufzählung `ToList` , und alle Elemente werden in eine Auflistung abgerufen.

Faules Bewertungsbeispiel: Fibonacci-Zahlen

```
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));
        }
    }
}
```

So funktioniert es unter der Haube (ich empfehle die resultierende .exe-Datei in IL Disaambler-Tool zu dekompilieren):

1. C # -Compiler generiert eine Klasse, die `IEnumerable<BigInteger>` und `IEnumerator<BigInteger>` (`<Fibonacci>d__0` in ildasm) implementiert.
2. Diese Klasse implementiert eine Zustandsmaschine. Der Zustand besteht aus der aktuellen Position in der Methode und den Werten der lokalen Variablen.
3. Der interessanteste Code befindet sich in der `bool IEnumerator.MoveNext()` Methode `bool IEnumerator.MoveNext()` . Grundsätzlich was `MoveNext()` macht:
    - Stellt den aktuellen Status wieder her. Variablen wie `prev` und `current` werden zu Feldern in unserer Klasse (`<current>5__2` und `<prev>5__1` in ildasm). In unserer Methode haben wir zwei Positionen (`<>1__state`): zuerst an der öffnenden geschweiften Klammer, an zweiter Stelle bei `yield return` .
    - Führt Code bis zum nächsten `yield return` oder `yield break` / `}` .
    - Für die `yield return` resultierende Wert gespeichert, sodass die `Current` Eigenschaft ihn zurückgeben kann. `true` wird zurückgegeben. An diesem Punkt wird der aktuelle Status für den nächsten `MoveNext` Aufruf erneut `MoveNext` .
    - Für die `yield break` / `}` -Methode wird nur `false` was bedeutet, dass die Iteration durchgeführt wird.

Beachten Sie auch, dass die 10001. Nummer 468 Byte lang ist. State Machine speichert nur `current` und `prev` Variablen als Felder. Wenn wir jedoch alle Zahlen in der Reihenfolge von der ersten bis zur 10000sten speichern möchten, beträgt die Speichergröße mehr als 4 Megabyte. So kann eine faule Auswertung, wenn sie richtig verwendet wird, in einigen Fällen den Speicherbedarf reduzieren.

Der Unterschied zwischen Break und Yield Break

Die Verwendung von `yield break` im Gegensatz zu `break` möglicherweise nicht so offensichtlich, wie man vielleicht denkt. Es gibt viele schlechte Beispiele im Internet, bei denen die Verwendung der beiden austauschbar ist und den Unterschied nicht wirklich veranschaulicht.

Der verwirrende Teil ist, dass beide Schlüsselwörter (oder Schlüsselphrasen) nur innerhalb von Schleifen Sinn machen (`foreach` , `while` ...). Wann also eines für das andere?

Es ist wichtig zu wissen, dass die Methode in einen Iterator umgewandelt wird , sobald Sie das `yield` Schlüsselwort in einer Methode verwenden. Der einzige Zweck einer solchen Methode besteht darin, eine endliche oder unendliche Sammlung und Ausbeute (Ausgabe) ihrer Elemente zu durchlaufen. Sobald der Zweck erfüllt ist, gibt es keinen Grund, die Ausführung der Methode fortzusetzen. Manchmal passiert das natürlich mit der letzten schließenden Klammer der Methode `}` . Aber manchmal möchten Sie die Methode vorzeitig beenden. In einer normalen (nicht iterierenden) Methode würden Sie das Schlüsselwort `return` . Aber man kann nicht verwenden `return` in einem Iterator, haben Sie verwenden `yield break` . Mit anderen Worten, `yield break` für einen Iterator ist die gleiche wie `return` für eine Standardmethode. Während die `break` Anweisung nur die nächste Schleife beendet.

Sehen wir uns einige Beispiele an:

```
/// <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;
    }
```

Ergebnis Keyword online lesen: https://riptutorial.com/de/csharp/topic/61/ergebnis-keyword

Kapitel 57: Erste Schritte: Json mit C

Einführung

Im folgenden Thema wird eine Möglichkeit zum Arbeiten mit Json unter Verwendung der C # -Sprache und Konzepte der Serialisierung und Deserialisierung beschrieben.

Examples

Einfaches Json-Beispiel

```
{
    "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
             }]
}
```

Wenn Sie neu bei Json sind, finden Sie hier ein beispielhaftes Tutorial .

Das Wichtigste zuerst: Bibliothek, um mit Json zu arbeiten

Um mit Json unter Verwendung von C # arbeiten zu können, müssen Sie Newtonsoft (.net-Bibliothek) verwenden. Diese Bibliothek bietet Methoden, mit denen der Programmierer Objekte und mehr Objekte serialisieren und deserialisieren kann. Es gibt ein Tutorial, wenn Sie Details zu den Methoden und Verwendungen erfahren möchten.

Wenn Sie Visual Studio verwenden, *wechseln* Sie zu *Tools / Nuget Package Manager / Package to Solution /,* und geben Sie "Newtonsoft" in die Suchleiste ein. Installieren Sie das Paket. Wenn Sie nicht über NuGet verfügen, kann Ihnen dieses ausführliche Tutorial helfen.

C # -Implementierung

Bevor Sie Code lesen, ist es wichtig, die wichtigsten Konzepte zu verstehen, die bei der Programmierung von Anwendungen mit json hilfreich sind.

> **Serialisierung** : Konvertieren eines Objekts in einen Bytestrom, der durch

Anwendungen gesendet werden kann. Der folgende Code kann serialisiert und in den vorherigen Json konvertiert werden.

Deserialisierung : Prozess der Umwandlung eines Json / Streams von Bytes in ein Objekt. Es ist genau das Gegenteil der Serialisierung. Der vorherige Json kann in ein C # -Objekt deserialisiert werden, wie in den folgenden Beispielen gezeigt.

Um dies zu klären, ist es wichtig, die json-Struktur in Klassen umzuwandeln, um die bereits beschriebenen Prozesse zu verwenden. Wenn Sie Visual Studio verwenden, können Sie einen Json automatisch in eine Klasse umwandeln, indem Sie *"Bearbeiten / Einfügen Spezial / JSON als Klassen* einfügen *"* auswählen und die Json-Struktur einfügen.

```
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;
}
```

Serialisierung

```
 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
    }
```

Die Methode „.SerializeObject“ erhält als Parameter einen *Typ Objekt,* so dass man alles hineingesteckt werden.

Deserialisierung

Sie können einen Json von überall, von einer Datei oder sogar von einem Server empfangen, sodass er im folgenden Code nicht enthalten ist.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

Die Methode ".DeserializeObject" deserialisiert " *jsonExample* " in ein " *Author* " -Objekt. Aus diesem Grund ist es wichtig, die Json-Variablen in der Klassendefinition zu setzen, damit die Methode darauf zugreifen kann, um sie zu füllen.

Serialization & Des-Serialization Allgemeine Dienstprogramme

In diesem Beispiel wurde die allgemeine Funktion für die Serialisierung und Deserialisierung von Objekten aller Art verwendet.

```
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;
        }
}
}
```

Erste Schritte: Json mit C # online lesen: https://riptutorial.com/de/csharp/topic/9910/erste-schritte--json-mit-c-sharp

Kapitel 58: Erweiterungsmethoden

Syntax

- public static ReturnType MyExtensionMethod (dieses TargetType-Ziel)
- public static ReturnType MyExtensionMethod (dieses TargetType-Ziel, TArg1 arg1, ...)

Parameter

Parameter	Einzelheiten
diese	Dem ersten Parameter einer Erweiterungsmethode sollte immer das Schlüsselwort `this` vorangestellt werden, gefolgt von der Kennung, mit der auf die "aktuelle" Instanz des zu erweiternden Objekts verwiesen wird

Bemerkungen

Erweiterungsmethoden sind syntaktischer Zucker, mit dem statische Methoden für Objektinstanzen aufgerufen werden können, als wären sie ein Mitglied des Typs.

Erweiterungsmethoden erfordern ein explizites Zielobjekt. Sie müssen das Schlüsselwort `this` um auf die Methode innerhalb des erweiterten Typs selbst zuzugreifen.

Erweiterungsmethoden müssen als statisch deklariert werden und in einer statischen Klasse leben.

Welcher Namensraum?

Die Wahl des Namespaces für Ihre Erweiterungsmethodenklasse ist ein Kompromiss zwischen Sichtbarkeit und Erkennbarkeit.

Die am häufigsten genannte Option ist ein benutzerdefinierter Namespace für Ihre Erweiterungsmethoden. Dies erfordert jedoch einen Kommunikationsaufwand, sodass die Benutzer Ihres Codes wissen, dass die Erweiterungsmethoden vorhanden sind, und wo sie zu finden sind.

Alternativ können Sie einen Namespace auswählen, sodass Entwickler Ihre Erweiterungsmethoden über Intellisense ermitteln können. Wenn Sie also die `Foo` Klasse erweitern möchten, ist es logisch, die Erweiterungsmethoden in den gleichen Namespace wie `Foo` .

Es ist wichtig zu `IEnumerable` , dass **nichts Sie daran hindert, den Namespace "einer anderen Person" zu verwenden** : Wenn Sie `IEnumerable` erweitern `IEnumerable` , können Sie Ihre Erweiterungsmethode im `System.Linq` Namespace `System.Linq` .

Dies ist nicht *immer* eine gute Idee. In einem bestimmten Fall möchten Sie beispielsweise einen

allgemeinen Typ erweitern (z. B. `bool IsApproxEqualTo(this double value, double other)`), ohne jedoch das gesamte `System` "verschmutzen". In diesem Fall sollte ein lokaler, spezifischer Namespace ausgewählt werden.

Schließlich ist es auch möglich, die Erweiterungsmethoden in *keinen Namensraum zu setzen* !

Eine gute Referenzfrage: Wie verwalten Sie die Namespaces Ihrer Erweiterungsmethoden?

Anwendbarkeit

Beim Erstellen von Erweiterungsmethoden ist darauf zu achten, dass sie für alle möglichen Eingaben geeignet sind und nicht nur für bestimmte Situationen relevant sind. Zum Beispiel ist es möglich, Systemklassen wie `string` , wodurch der neue Code für **jeden** String verfügbar ist. Wenn Ihr Code domänenspezifische Logik für ein domänenspezifisches Zeichenfolgenformat ausführen muss, ist eine Erweiterungsmethode nicht geeignet, da das Vorhandensein von Aufrufern die Verwendung anderer Zeichenfolgen im System verwechseln würde.

Die folgende Liste enthält grundlegende Funktionen und Eigenschaften von Erweiterungsmethoden

1. Es muss eine statische Methode sein.
2. Es muss sich in einer statischen Klasse befinden.
3. Es verwendet das Schlüsselwort "this" als ersten Parameter mit einem Typ in .NET. Diese Methode wird von einer bestimmten Typinstanz auf der Clientseite aufgerufen.
4. Es wird auch von VS Intellisense gezeigt. Wenn wir den Punkt drücken . Nach einer Typinstanz kommt es dann in VS Intellisense.
5. Eine Erweiterungsmethode sollte sich in demselben Namespace befinden wie sie verwendet wird, oder Sie müssen den Namespace der Klasse mithilfe einer using-Anweisung importieren.
6. Sie können einen beliebigen Namen für die Klasse angeben, die über eine Erweiterungsmethode verfügt, die Klasse sollte jedoch statisch sein.
7. Wenn Sie einem Typ neue Methoden hinzufügen möchten und nicht über den Quellcode dafür verfügen, können Sie Erweiterungsmethoden dieses Typs verwenden und implementieren.
8. Wenn Sie Erweiterungsmethoden erstellen, die dieselbe Signaturmethode wie der Typ haben, den Sie erweitern, werden die Erweiterungsmethoden niemals aufgerufen.

Examples

Erweiterungsmethoden - Übersicht

Erweiterungsmethoden wurden in C # 3.0 eingeführt. Erweiterungsmethoden erweitern vorhandene Verhaltenstypen und fügen sie hinzu, ohne dass ein neuer abgeleiteter Typ erstellt, neu kompiliert oder der ursprüngliche Typ anderweitig geändert wird. *Sie sind besonders hilfreich, wenn Sie die Quelle eines Typs, den Sie verbessern möchten, nicht ändern können.* Erweiterungsmethoden können für Systemtypen, von Dritten definierte Typen und von Ihnen selbst definierte Typen erstellt werden. Die Erweiterungsmethode kann aufgerufen werden, als

wäre sie eine Member-Methode des ursprünglichen Typs. Dies ermöglicht, dass **Method Chaining** zur Implementierung einer **Fluent-Schnittstelle verwendet wird** .

Eine Erweiterungsmethode wird erstellt, indem einer **statischen Klasse** eine **statische Methode** hinzugefügt **wird** , die sich vom ursprünglichen Typ unterscheidet, der erweitert wird. Die statische Klasse, die die Erweiterungsmethode enthält, wird häufig nur zum Zweck des Haltens von Erweiterungsmethoden erstellt.

Erweiterungsmethoden benötigen einen speziellen ersten Parameter, der den ursprünglichen Typ angibt, der erweitert wird. Dieser erste Parameter wird mit dem Schlüsselwort verziert `this` (die eine besondere und unterschiedliche Verwendung stellt `this` in C # -en sollte so verschieden von der Verwendung zu verstehen `this` , die an den Mitglieder der aktuellen Objektinstanz ermöglicht Bezug).

Im folgenden Beispiel wird der ursprüngliche Typ erweitert die Klasse `string` . `String` wurde um die Methode `Shorten()` , die die zusätzliche Funktion der Verkürzung bietet. Die statische Klasse `StringExtensions` wurde erstellt, um die Erweiterungsmethode aufzunehmen. Die Erweiterungsmethode `Shorten()` zeigt an, dass es sich um eine Erweiterung von `string` über den speziell markierten ersten Parameter handelt. Um zu zeigen, dass die `Shorten()` -Methode den `string` , wird der erste Parameter mit `this` markiert. Daher ist die vollständige Signatur des ersten Parameters `this string text` , wobei `string` der ursprüngliche Typ ist, der erweitert wird, und `text` der gewählte Parametername ist.

```
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);
    }
}
```

Live-Demo zu .NET-Geige

Das Objekt, das als *erstes Argument einer Erweiterungsmethode* (mit dem Schlüsselwort `this` begleitet) übergeben wird, ist die Instanz, für die die Erweiterungsmethode aufgerufen wird.

Zum Beispiel, wenn dieser Code ausgeführt wird:

```
"some string".Shorten(5);
```

Die Werte der Argumente lauten wie folgt:

```
text: "some string"
length: 5
```

Beachten Sie, dass Erweiterungsmethoden nur verwendet werden können, wenn sie sich im selben Namespace befinden wie ihre Definition, wenn der Namespace explizit durch den Code mithilfe der Erweiterungsmethode importiert wird oder wenn die Erweiterungsklasse ohne Namespace ist. In den .NET Framework-Richtlinien wird empfohlen, Erweiterungsklassen in einem eigenen Namespace abzulegen. Dies kann jedoch zu Ermittlungsproblemen führen.

Dies führt zu keinen Konflikten zwischen den Erweiterungsmethoden und den verwendeten Bibliotheken, es sei denn, Namespaces, die in Konflikt stehen könnten, werden explizit eingezogen. Zum Beispiel LINQ-Erweiterungen :

```
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);
    }
}
```

Live-Demo zu .NET-Geige

Seit C # 6.0 ist es auch möglich, eine `using static` Direktive in die *Klasse zu setzen,* die die Erweiterungsmethoden enthält. `using static System.Linq.Enumerable;` Sie beispielsweise `using static System.Linq.Enumerable;` . Dadurch werden Erweiterungsmethoden dieser bestimmten Klasse verfügbar, ohne dass andere Typen aus demselben Namespace in den Gültigkeitsbereich aufgenommen werden.

Wenn eine Klassenmethode mit derselben Signatur verfügbar ist, priorisiert der Compiler diese dem Aufruf der Erweiterungsmethode. Zum Beispiel:

```
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"
    }
}
```

Live-Demo zu .NET Fiddle

Wenn zwei Erweiterungsfunktionen mit derselben Signatur vorhanden sind und sich eine davon im selben Namespace befindet, wird diese mit Priorität versehen. Wenn jedoch auf beide über `using` zugegriffen wird, wird ein Fehler bei der Kompilierung mit der Nachricht angezeigt:

Der Aufruf ist zwischen den folgenden Methoden oder Eigenschaften mehrdeutig

---

Beachten Sie, dass der syntaktische Komfort des Aufrufs einer Erweiterungsmethode über `originalTypeInstance.ExtensionMethod()` ein optionaler Komfort ist. Die Methode kann auch auf herkömmliche Weise aufgerufen werden, so dass der spezielle erste Parameter als Parameter für die Methode verwendet wird.

Dh beide der folgenden Arbeiten:

```
//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);
```

Explizite Verwendung einer Erweiterungsmethode

Erweiterungsmethoden können auch wie gewöhnliche statische Klassenmethoden verwendet werden. Diese Methode zum Aufrufen einer Erweiterungsmethode ist ausführlicher, jedoch in einigen Fällen erforderlich.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
```

```
}
```

Verwendungszweck:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

Wann werden Erweiterungsmethoden als statische Methoden aufgerufen?

Es gibt immer noch Szenarien, in denen Sie eine Erweiterungsmethode als statische Methode verwenden müssen:

- Konfliktlösung mit einer Member-Methode Dies kann passieren, wenn eine neue Version einer Bibliothek eine neue Member-Methode mit derselben Signatur einführt. In diesem Fall wird die Member-Methode vom Compiler bevorzugt.
- Lösen von Konflikten mit einer anderen Erweiterungsmethode mit derselben Signatur. Dies kann passieren, wenn zwei Bibliotheken ähnliche Erweiterungsmethoden enthalten und Namespaces beider Klassen mit Erweiterungsmethoden in derselben Datei verwendet werden.
- Übergabe der Erweiterungsmethode als Methodengruppe an den Delegatenparameter.
- Machen Sie Ihre eigene Bindung durch `Reflection` .
- Verwenden der Erweiterungsmethode im Direktfenster in Visual Studio.

Statisch verwenden

Wenn eine `using static` Direktive verwendet wird, um statische Member einer statischen Klasse in den globalen Bereich zu bringen, werden Erweiterungsmethoden übersprungen. Beispiel:

```
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);
```

Wenn Sie den Modifikator `this` aus dem ersten Argument der `Shorten` Methode entfernen, wird die letzte Zeile kompiliert.

Nullprüfung

Erweiterungsmethoden sind statische Methoden, die sich wie Instanzmethoden verhalten. Doch im Gegensatz zu, was passiert , wenn eine Instanz - Methode auf einer Aufruf `null` Referenz, wenn eine Erweiterungsmethode mit einer genannt wird `null` Referenz, macht es keinen werfen

`NullReferenceException` . Dies kann in einigen Szenarien sehr nützlich sein.

Betrachten Sie beispielsweise die folgende statische Klasse:

```
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
```

Live-Demo zu .NET-Geige

Erweiterungsmethoden können nur öffentliche (oder interne) Mitglieder der erweiterten Klasse anzeigen

```
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
    }
}
```

Erweiterungsmethoden sind nur ein syntaktischer Zucker und eigentlich keine Mitglieder der Klasse, die sie erweitern. Das bedeutet, dass sie die Kapselung nicht brechen können. Sie haben nur Zugriff auf `public` (oder wenn sie in derselben Assembly, `internal`) Feldern, Eigenschaften

und Methoden implementiert sind.

Generische Erweiterungsmethoden

Erweiterungsmethoden können wie andere Methoden Generika verwenden. Zum Beispiel:

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

und es wäre so zu nennen:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

Demo anzeigen

Ebenso für mehrere Typargumente:

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

Anrufen wäre wie folgt:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

Demo anzeigen

Sie können auch Erweiterungsmethoden für teilweise gebundene Typen in mehreren generischen Typen erstellen:

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```

Anrufen wäre wie folgt:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

Demo anzeigen

Sie können auch Typeinschränkungen angeben, wobei `where` :

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Aufrufcode:

```
int number = 5;
var IsDefault = number.IsDefault();
```

Demo anzeigen

Versenden von Erweiterungsmethoden basierend auf statischem Typ

Der statische Typ (Kompilierzeit) wird anstelle des dynamischen Typs (Laufzeittyp) verwendet, um die Parameter abzugleichen.

```
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"
    }
}
```

Live-Demo zu .NET-Geige

Der Versand, der auf einem statischen Typ basiert, erlaubt es nicht, eine Erweiterungsmethode für ein `dynamic` Objekt aufzurufen:

```
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
```

Erweiterungsmethoden werden von dynamischem Code nicht unterstützt.

```
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");
    }
}
```

> Der Grund [der Aufruf von Erweiterungsmethoden aus dynamischem Code] funktioniert nicht, weil in regulären, nicht dynamischen Code-Erweiterungsmethoden eine vollständige Suche in allen dem Compiler bekannten Klassen nach einer statischen Klasse mit einer entsprechenden Erweiterungsmethode durchgeführt wird . Die Suche wird basierend auf der Namespace-Verschachtelung in der Reihenfolge ausgeführt und ist `using` Direktiven in jedem Namespace verfügbar.
>
> Das bedeutet , dass , um einen dynamischen Erweiterung Methodenaufruf korrekt aufgelöst zu bekommen, irgendwie das DLR *zur Laufzeit* wissen hat , was alle Namespace Verschachtelungen und `using` Direktiven *im Quellcode* waren. Wir verfügen nicht über einen praktischen Mechanismus, um all diese Informationen in die Anrufseite zu kodieren. Wir haben überlegt, einen solchen Mechanismus zu erfinden, entschieden uns jedoch für zu hohe Kosten und ein zu großes Zeitplanrisiko, um es sich zu lohnen.

Quelle

Erweiterungsmethoden als stark typisierte Wrapper

Erweiterungsmethoden können zum Schreiben stark typisierter Wrapper für wörterbuchartige Objekte verwendet werden. Zum Beispiel einen Cache, `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;
}
```

Dieser Ansatz beseitigt die Notwendigkeit, String-Literale als Schlüssel in der gesamten Codebase zu verwenden, und es ist auch nicht erforderlich, während des Lesevorgangs auf den erforderlichen Typ zu gießen. Insgesamt wird eine sicherere, stark typisierte Art der Interaktion mit derart lose typisierten Objekten wie Wörterbüchern geschaffen.

Erweiterungsmethoden für die Verkettung

Wenn eine Erweiterungsmethode einen Wert zurückgibt, der denselben Typ wie `this` Argument hat, kann sie verwendet werden, um einen oder mehrere Methodenaufrufe mit einer kompatiblen Signatur zu "verketten". Dies kann für versiegelte und / oder primitive Typen nützlich sein und

ermöglicht die Erstellung sogenannter "fließender" APIs, wenn die Methodennamen wie natürliche menschliche Sprache gelesen werden.

```
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;
    }
}
```

Oder so

```
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();
    }
}
```

Erweiterungsmethoden in Kombination mit Schnittstellen

Die Verwendung von Erweiterungsmethoden mit Schnittstellen ist sehr praktisch, da die Implementierung außerhalb der Klasse gespeichert werden kann. Alles, was zur Erweiterung der Klasse erforderlich ist, ist das Dekorieren der Klasse mit Schnittstelle.

```
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();
   }
}
```

Verwenden Sie wie:

```
 var classy = new Classy();
 classy.Do(); // will call the extension
 classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

IList Erweiterungsmethode - Beispiel: Vergleich von 2 Listen

Sie können die folgende Erweiterungsmethode verwenden, um den Inhalt von zwei IList <T> - Instanzen desselben Typs zu vergleichen.

Standardmäßig werden die Elemente basierend auf ihrer Reihenfolge in der Liste und den Elementen selbst verglichen. `isOrdered` false an den Parameter `isOrdered` , werden nur die Elemente selbst unabhängig von ihrer Reihenfolge verglichen.

Für diese Methode funktioniert, der generische Typ (`T` müssen beide außer Kraft setzen) `Equals` und `GetHashCode` Methoden.

Verwendungszweck:

```
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
```

Methode:

```
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;
    }
}
```

Erweiterungsmethoden mit Enumeration

Erweiterungsmethoden sind nützlich, um Aufzählungen Funktionalität hinzuzufügen.

Eine übliche Verwendung ist die Implementierung einer Konvertierungsmethode.

```
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;
    }
}
```

Jetzt können Sie Ihren Aufzählungswert schnell in einen anderen Typ konvertieren. In diesem Fall ein Bool.

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

Alternativ können Erweiterungsmethoden verwendet werden, um eigenschaftsähnliche Methoden

hinzuzufügen.

```
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();
```

Erweiterungen und Schnittstellen ermöglichen zusammen DRY-Code und Mixin-ähnliche Funktionalität

Erweiterungsmethoden ermöglichen es Ihnen, Ihre Schnittstellendefinitionen zu vereinfachen, indem Sie nur die erforderlichen Kernfunktionen in die Schnittstelle selbst aufnehmen und als Erweiterungsmethoden Komfortmethoden und Überladungen definieren. Schnittstellen mit weniger Methoden lassen sich in neuen Klassen leichter implementieren. Durch das Beibehalten von Überladungen als Erweiterungen, anstatt sie in die Benutzeroberfläche aufzunehmen, müssen Sie den Boilerplate-Code nicht direkt in jede Implementierung kopieren. Dies ist in der Tat dem Mischmuster ähnlich, das C # nicht unterstützt.

`System.Linq.Enumerable` ,s Erweiterungen `IEnumerable<T>` ist ein gutes Beispiel dafür. `IEnumerable<T>` muss die implementierende Klasse nur zwei Methoden implementieren: generische und nicht generische `GetEnumerator()` . `System.Linq.Enumerable` bietet jedoch unzählige nützliche Dienstprogramme als Erweiterungen, die eine übersichtliche und klare Verwendung von `IEnumerable<T>` .

Im Folgenden finden Sie eine sehr einfache Schnittstelle, die als Erweiterungen nützliche

Überlastungen bietet.

```
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)}");
    }
}
```

Erweiterungsmethoden für die Behandlung von Sonderfällen

Erweiterungsmethoden können verwendet werden, um die Verarbeitung ineleganter Geschäftsregeln "zu verbergen", die andernfalls eine aufrufende Funktion mit if / then-Anweisungen überladen würden. Dies ist ähnlich und vergleichbar mit dem Umgang mit Nullwerten mit Erweiterungsmethoden. Zum Beispiel,

```
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.
```

Verwenden von Erweiterungsmethoden mit statischen Methoden und Rückrufen

Erwägen Sie die Verwendung von Erweiterungsmethoden als Funktionen, die anderen Code umschließen. Hier ein gutes Beispiel, das sowohl eine statische als auch eine Erweiterungsmethode verwendet, um das Try Catch-Konstrukt zu umschließen. Machen Sie Ihren Code Bullet Proof ...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
```

```
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
            {
                //error is the already logged and handled exception
                //determine the base result
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

Erweiterungsmethoden für Interfaces

Eine nützliche Funktion von Erweiterungsmethoden ist, dass Sie gängige Methoden für eine Schnittstelle erstellen können. Normalerweise kann eine Schnittstelle keine gemeinsamen Implementierungen haben, mit Erweiterungsmethoden jedoch.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}
```

```
public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

In diesem Beispiel kann die Methode `FeetDriven` für jedes `IVehicle` `FeetDriven` verwendet werden. Diese Logik in dieser Methode würde für alle `IVehicle` gelten, so dass dies auf diese Weise geschehen kann, so dass es in der `IVehicle` Definition keinen `FeetDriven` `IVehicle` muss, der für alle Kinder gleich implementiert wäre.

Verwenden von Erweiterungsmethoden zum Erstellen schöner Mapper-Klassen

Wir können bessere Mapper-Klassen mit Erweiterungsmethoden erstellen. Angenommen, ich habe einige DTO-Klassen wie

```
public class UserDTO
{
       public AddressDTO Address { get; set; }
}

public class AddressDTO
{
       public string Name { get; set; }
}
```

und ich muss entsprechende Ansichtsmodellklassen zuordnen

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

Dann kann ich meine Mapper-Klasse wie folgt erstellen

```
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
                };
      }
}
```

Dann kann ich meinen Mapper wie folgt aufrufen

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
        };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

Das Schöne daran ist, dass alle Zuordnungsmethoden einen gemeinsamen Namen haben (ToViewModel), und wir können sie auf verschiedene Weise wiederverwenden

Verwenden von Erweiterungsmethoden zum Erstellen neuer Auflistungstypen (z. B. DictList)

Sie können Erweiterungsmethoden erstellen, um die Verwendbarkeit für verschachtelte Sammlungen wie ein `Dictionary` mit einem `List<T>` zu verbessern.

Berücksichtigen Sie die folgenden Erweiterungsmethoden:

```
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;
    }
}
```

Sie können die Erweiterungsmethoden wie folgt verwenden:

```
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
```

Demo anzeigen

Erweiterungsmethoden online lesen:
https://riptutorial.com/de/csharp/topic/20/erweiterungsmethoden

Kapitel 59: FileSystemWatcher

Syntax

- public FileSystemWatcher ()
- public FileSystemWatcher (String-Pfad)
- public FileSystemWatcher (String-Pfad, String-Filter)

Parameter

Pfad	Filter
Das zu überwachende Verzeichnis in Standard- oder UNC-Notation (Universal Naming Convention).	Die Art der Dateien, die angesehen werden sollen. Zum Beispiel überwacht "* .txt" alle Textdateien auf Änderungen.

Examples

Grundlegender FileWatcher

Im folgenden Beispiel wird ein `FileSystemWatcher` , um das zur Laufzeit angegebene Verzeichnis zu überwachen. Die Komponente ist so eingestellt, dass sie nach Änderungen in der Zeit von **LastWrite** und **LastAccess** , nach dem Erstellen, Löschen oder Umbenennen von Textdateien im Verzeichnis **sucht** . Wenn eine Datei geändert, erstellt oder gelöscht wird, wird der Pfad zur Datei an die Konsole gedruckt. Wenn eine Datei umbenannt wird, werden der alte und der neue Pfad auf der Konsole gedruckt.

Verwenden Sie für dieses Beispiel die Namespaces System.Diagnostics und System.IO.

```
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

Ein häufiger Fehler, den viele Leute mit FileSystemWatcher beginnen, berücksichtigt nicht, dass das FileWatcher-Ereignis ausgelöst wird, sobald die Datei erstellt wird. Es kann jedoch einige Zeit dauern, bis die Datei fertig ist.

Beispiel :

Nehmen Sie zum Beispiel eine Dateigröße von 1 GB. Die Datei wurde von einem anderen Programm erstellt (Explorer.exe kopiert sie von irgendwo), es dauert jedoch einige Minuten, bis der Vorgang abgeschlossen ist. Das Ereignis wird zu diesem Zeitpunkt ausgelöst, und Sie müssen warten, bis die Datei zum Kopieren bereit ist.

Dies ist eine Methode, um zu überprüfen, ob die Datei bereit ist.

```
 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;
    }
}
```

FileSystemWatcher online lesen: https://riptutorial.com/de/csharp/topic/5061/filesystemwatcher

Kapitel 60: Flyweight Design Pattern implementieren

Examples

Implementieren einer Karte im RPG-Spiel

Fliegengewicht ist eines der strukturellen Entwurfsmuster. Es wird verwendet, um die Menge des verwendeten Speichers zu reduzieren, indem möglichst viele Daten mit ähnlichen Objekten gemeinsam genutzt werden. In diesem Dokument erfahren Sie, wie Sie Flyweight DP richtig einsetzen.

Lassen Sie mich Ihnen die Idee an einem einfachen Beispiel erklären. Stellen Sie sich vor, Sie arbeiten an einem RPG-Spiel und müssen riesige Dateien laden, die einige Charaktere enthalten. Zum Beispiel:

- `#` ist Gras. Sie können darauf laufen.
- `$` ist Ausgangspunkt
- `@` ist Rock. Du kannst nicht darauf laufen.
- `%` ist Schatztruhe

Beispiel einer Karte:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@##############@@@@@#######@#$@@@

@##############@@@@@#######@####@@@

@########%#######@############@@@

@#############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Da diese Objekte ähnliche Merkmale aufweisen, müssen Sie nicht für jedes Kartenfeld ein separates Objekt erstellen. Ich werde dir zeigen, wie man Fliegengewicht benutzt.

Definieren wir eine Schnittstelle, die unsere Felder implementieren werden:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Jetzt können wir Klassen erstellen, die unsere Felder repräsentieren. Wir müssen sie auch irgendwie identifizieren (ich habe eine Aufzählung verwendet):

```
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; } }
}
```

Wie gesagt, wir müssen nicht für jedes Feld eine eigene Instanz erstellen. Wir müssen ein **Repository** von Feldern erstellen. Das Wesen von Flyweight DP besteht darin, dass wir ein Objekt nur dynamisch erstellen, wenn wir es brauchen und es in unserem Repo noch nicht vorhanden ist, oder es zurückgeben, wenn es bereits existiert. Schreiben wir eine einfache Klasse, die dies für uns erledigt:

```
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);
    }
}
```

Großartig! Jetzt können wir unseren Code testen:

```
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);
    }
}
```

Das Ergebnis in der Konsole sollte lauten:

> Erstellt eine neue Instanz von Grass
>
> Erstellt eine neue Instanz von Rock

Aber warum erscheint Gras nur einmal, wenn wir es zweimal haben wollten? Das liegt daran, dass das erste Mal, wenn wir die `GetField` aufrufen, nicht in unserem **Repository vorhanden** ist. Sie wird also erstellt. `GetField` wir jedoch das nächste Mal Gras benötigen, ist sie bereits vorhanden. Daher geben wir sie nur zurück.

Flyweight Design Pattern implementieren online lesen:
https://riptutorial.com/de/csharp/topic/4619/flyweight-design-pattern-implementieren

Kapitel 61: Func Delegierte

Syntax

- `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)`

Parameter

Parameter	Einzelheiten
`arg` oder `arg1`	der (erste) Parameter der Methode
`arg2`	der zweite Parameter der Methode
`arg3`	der dritte Parameter der Methode
`arg4`	der vierte Parameter der Methode
`T` oder `T1`	der Typ des (ersten) Parameters der Methode
`T2`	der Typ des zweiten Parameters der Methode
`T3`	der Typ des dritten Parameters der Methode
`T4`	der Typ des vierten Parameters der Methode
`TResult`	der Rückgabetyp der Methode

Examples

Ohne Parameter

Dieses Beispiel zeigt, wie ein Delegat erstellt wird, der die Methode kapselt, die die aktuelle Uhrzeit zurückgibt

```
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();
}
```

Mit mehreren Variablen

```
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);
}
```

Lambda & anonyme Methoden

Eine anonyme Methode kann überall dort zugewiesen werden, wo ein Delegat erwartet wird:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Lambda-Ausdrücke können verwendet werden, um dasselbe auszudrücken:

```
Func<int, int> square = x => x * x;
```

In beiden Fällen können wir nun die in `square` gespeicherte Methode wie folgt aufrufen:

```
var sq = square.Invoke(2);
```

Oder als Abkürzung:

```
var sq = square(2);
```

Beachten Sie, dass für die typsichere Zuweisung die Parametertypen und der Rückgabetyp der anonymen Methode mit denen des Delegattyps übereinstimmen müssen:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

Parameter für Covariante und Kontravariante Typen

`Func` unterstützt auch Covariant & Contravariant

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Func Delegierte online lesen: https://riptutorial.com/de/csharp/topic/2769/func-delegierte

Kapitel 62: Funktion mit mehreren Rückgabewerten

Bemerkungen

In C # gibt es keine inhärente Antwort auf dieses sogenannte Bedürfnis. Trotzdem gibt es Workarounds, um dieses Bedürfnis zu befriedigen.

Der Grund, warum ich das Bedürfnis als "so genannt" bezeichne, ist, dass wir nur Methoden mit 2 oder mehr als 2 Werten benötigen, um zurückzukehren, wenn wir gegen gute Programmiergrundsätze verstoßen. Insbesondere das Prinzip der Einzelverantwortung .

Daher wäre es besser, wenn wir auf Funktionen aufmerksam machen, die 2 oder mehr Werte zurückgeben und unser Design verbessern.

Examples

Lösung "anonymes Objekt" + "dynamisches Schlüsselwort"

Sie können ein anonymes Objekt aus Ihrer Funktion zurückgeben

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

Und weisen Sie das Ergebnis einem dynamischen Objekt zu und lesen Sie die Werte darin.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

Tuple Lösung

Sie können eine Instanz der `Tuple` Klasse aus Ihrer Funktion mit zwei Vorlagenparametern als `Tuple<string, MyClass>` :

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

Und lesen Sie die Werte wie folgt:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

Ref- und Out-Parameter

Das `ref` Schlüsselwort wird verwendet, um ein Argument als Referenz zu übergeben . `out` tut dasselbe wie `ref` , benötigt jedoch vor dem Aufruf der Funktion keinen vom Anrufer zugewiesenen Wert.

Ref-Parameter : -Wenn Sie eine Variable als ref-Parameter übergeben möchten, müssen Sie sie initialisieren, bevor Sie sie als ref-Parameter an method übergeben.

Out-Parameter: - Wenn Sie eine Variable als Out-Parameter übergeben möchten, müssen Sie sie nicht initialisieren, bevor Sie sie als Out-Parameter an die Methode übergeben.

```
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;
}
```

Funktion mit mehreren Rückgabewerten online lesen:
https://riptutorial.com/de/csharp/topic/3908/funktion-mit-mehreren-ruckgabewerten

Kapitel 63: Funktionale Programmierung

Examples

Funktion und Aktion

Func bietet eine Halterung für parametrisierte anonyme Funktionen. Die führenden Typen sind die Eingaben und der letzte Typ ist immer der Rückgabewert.

```
// 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))
```

Aktionsobjekte sind wie void-Methoden, daher haben sie nur einen Eingabetyp. Es wird kein Ergebnis auf dem Auswertungsstapel abgelegt.

```
// 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.
```

Unveränderlichkeit

Unveränderlichkeit ist bei der funktionalen Programmierung üblich und bei der objektorientierten Programmierung selten.

Legen Sie beispielsweise einen Adresstyp mit veränderbarem Status an:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
```

```
}
```

Jeder Code kann jede Eigenschaft im obigen Objekt ändern.

Erstellen Sie nun den unveränderlichen Adresstyp:

```
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;
    }
}
```

Denken Sie daran, dass schreibgeschützte Sammlungen die Unveränderlichkeit nicht respektieren. Zum Beispiel,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

ist nicht unveränderlich, da der Benutzer des Objekts die Sammlung ändern kann (Elemente hinzufügen oder daraus entfernen). Um es unveränderlich zu machen, muss man entweder eine Schnittstelle wie IEnumerable verwenden, die keine hinzuzufügenden Methoden verfügbar macht, oder es zu einer ReadOnlyCollection machen.

```
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());
```

Mit dem unveränderlichen Objekt haben wir folgende Vorteile:

- Es befindet sich in einem bekannten Zustand (anderer Code kann es nicht ändern).

- Es ist fadensicher.
- Der Konstruktor bietet eine einzige Stelle für die Validierung.
- Wenn Sie wissen, dass das Objekt nicht geändert werden kann, ist der Code verständlicher.

Vermeiden Sie Nullreferenzen

C # -Entwickler erhalten eine Menge von Nullreferenzausnahmen. F # -Entwickler tun dies nicht, weil sie den Option-Typ haben. Ein Option <> -Typ (einige bevorzugen vielleicht <> als Namen) enthalten den Rückgabetyp Some und None. Es macht explizit deutlich, dass eine Methode im Begriff ist, einen Null-Datensatz zurückzugeben.

Beispielsweise können Sie Folgendes nicht lesen und wissen, ob Sie mit einem Nullwert arbeiten müssen.

```
var user = _repository.GetUser(id);
```

Wenn Sie die mögliche Null kennen, können Sie einen Boilerplate-Code einführen, um damit umzugehen.

```
var username = user != null ? user.Name : string.Empty;
```

Was ist, wenn wir stattdessen eine Option <> zurückgegeben haben?

```
Option<User> maybeUser = _repository.GetUser(id);
```

Der Code macht jetzt explizit deutlich, dass möglicherweise ein None-Datensatz zurückgegeben wird, und der Code für die Plattenspeicherung, der auf Some oder None überprüft werden muss:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

Die folgende Methode zeigt, wie eine Option <> zurückgegeben wird

```
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>();
}
```

Hier ist eine minimale Implementierung von Option <>.

```
public struct Option<T>
{
    private readonly T _value;
```

```
    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

Zur Veranschaulichung der obigen Vermeidung kann Null.csx mit C # REPL ausgeführt werden.

Wie gesagt, ist dies eine minimale Implementierung. Bei der Suche nach "Vielleicht" NuGet-Paketen werden eine Reihe guter Bibliotheken angezeigt.

Funktionen höherer Ordnung

Eine Funktion höherer Ordnung ist eine Funktion, die eine andere Funktion als Argument akzeptiert oder eine Funktion (oder beide) zurückgibt.

Dies wird im Allgemeinen mit Lambdas durchgeführt, beispielsweise wenn ein Vergleichselement an eine LINQ-Where-Klausel übergeben wird:

```
var results = data.Where(p => p.Items == 0);
```

Die Where () - Klausel könnte viele verschiedene Prädikate erhalten, was ihr beträchtliche Flexibilität verleiht.

Das Übergeben einer Methode an eine andere Methode wird auch beim Implementieren des Strategieentwurfsmusters angezeigt. Zum Beispiel können verschiedene Sortiermethoden ausgewählt und an eine Sortiermethode für ein Objekt übergeben werden, abhängig von den Anforderungen zur Laufzeit.

Unveränderliche Sammlungen

Das NuGet-Paket " `System.Collections.Immutable` enthält unveränderliche Auflistungsklassen.

Artikel erstellen und hinzufügen

```
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
```

Erstellen mit dem Builder

Bestimmte unveränderliche Sammlungen verfügen über eine `Builder` Klasse, mit der große unveränderliche Instanzen kostengünstig erstellt werden können:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

Erstellen aus einem vorhandenen IEnumerable

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

Liste aller unveränderlichen Sammlungstypen:

- 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>

Funktionale Programmierung online lesen: https://riptutorial.com/de/csharp/topic/2564/funktionale-programmierung

Kapitel 64: Generics

Syntax

- `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; } }`

Parameter

Parameter	Beschreibung
FERNSEHER	Geben Sie Platzhalter für generische Deklarationen ein

Bemerkungen

Generics in C # werden bis zur Laufzeit vollständig unterstützt: Generische Typen, die mit C # erstellt wurden, behalten ihre generische Semantik selbst nach der Kompilierung in CIL .

Dies bedeutet effektiv, dass es in C # möglich ist, generische Typen zu reflektieren und sie so zu sehen, wie sie deklariert wurden, oder zu überprüfen, ob ein Objekt beispielsweise eine Instanz eines generischen Typs ist. Dies steht im Gegensatz zur Typlöschung , bei der generische Typinformationen während des Kompilierens entfernt werden. Dies steht auch im Gegensatz zum Vorlagenansatz für Generika, bei dem mehrere konkrete generische Typen zur Laufzeit zu mehreren nicht generischen Typen werden und alle Metadaten, die zur weiteren Instanziierung der ursprünglichen generischen Typdefinitionen erforderlich sind, verloren gehen.

Seien Sie jedoch vorsichtig, wenn Sie über generische Typen nachdenken: Die Namen generischer Typen werden beim Kompilieren geändert. Dabei werden die spitzen Klammern und die Namen der Typparameter durch ein Backtick gefolgt von der Anzahl der generischen Typparameter ersetzt. Daher wird ein `Dictionary<TKey, Tvalue>` in `Dictionary`2` übersetzt.

Examples

Typparameter (Klassen)

Erklärung:

```
class MyGenericClass<T1, T2, T3, ...>
{
```

```
    // Do something with the type parameters.
}
```

Initialisierung:

```
var x = new MyGenericClass<int, char, bool>();
```

Verwendung (als Typ eines Parameters):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Typparameter (Methoden)

Erklärung:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Aufruf:

Es gibt keine Notwendigkeit, Typ-Argumente für eine genric-Methode bereitzustellen, da der Compiler implizit auf den Typ schließen kann.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

Bei Unklarheiten müssen jedoch generische Methoden mit type arguemnts as aufgerufen werden

```
MyGenericMethod<int, int, string>(x,y,z);
```

Typparameter (Schnittstellen)

Erklärung:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Verwendung (in Vererbung):

```
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> { ... }
```

Verwendung (als Typ eines Parameters):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

Implizite Typinferenz (Methoden)

Bei der Übergabe formeller Argumente an eine generische Methode können relevante generische Typargumente normalerweise implizit abgeleitet werden. Wenn alle generischen Typen abgeleitet werden können, ist die Angabe in der Syntax optional.

Betrachten Sie die folgende generische Methode. Es hat einen formalen Parameter und einen generischen Typparameter. Es gibt eine sehr offensichtliche Beziehung zwischen ihnen - der Typ, der als Argument an den generischen Typparameter übergeben wird, muss dem Typ der Kompilierungszeit des an den Formalparameter übergebenen Arguments entsprechen.

```
void M<T>(T obj)
{
}
```

Diese beiden Aufrufe sind gleichwertig:

```
M<object>(new object());
M(new object());
```

Diese beiden Aufrufe sind auch gleichwertig:

```
M<string>("");
M("");
```

Und so sind diese drei Aufrufe:

```
M<object>("");
M((object) "");
M("" as object);
```

Beachten Sie, dass alle Argumente angegeben werden müssen, wenn mindestens ein Typargument nicht abgeleitet werden kann.

Betrachten Sie die folgende generische Methode. Das erste generische Typargument ist mit dem Typ des formalen Arguments identisch. Es gibt jedoch keine solche Beziehung für das zweite generische Typargument. Daher kann der Compiler bei einem Aufruf dieser Methode nicht auf das zweite generische Typargument schließen.

```
void X<T1, T2>(T1 obj)
{
}
```

Das funktioniert nicht mehr:

```
X("");
```

Dies funktioniert auch nicht, weil der Compiler nicht sicher ist, ob wir den ersten oder den zweiten generischen Parameter angeben (beide wären als `object` gültig):

```
X<object>("");
```

Wir müssen beide wie folgt eingeben:

```
X<string, object>("");
```

Typeinschränkungen (Klassen und Schnittstellen)

Typeinschränkungen können einen Typparameter zwingen, eine bestimmte Schnittstelle oder Klasse zu implementieren.

```
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());
```

Syntax für mehrere Einschränkungen:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

Typeinschränkungen funktionieren auf dieselbe Weise wie Vererbung, da es möglich ist, mehrere Schnittstellen als Einschränkungen für den generischen Typ anzugeben, jedoch nur eine Klasse:

```
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
{
}
```

Eine andere Regel ist, dass die Klasse als erste Einschränkung und dann die Schnittstellen hinzugefügt werden muss:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

Alle deklarierten Bedingungen müssen gleichzeitig erfüllt sein, damit eine bestimmte generische Instantiierung funktioniert. Es gibt keine Möglichkeit, zwei oder mehr alternative Sätze von Abhängigkeiten anzugeben.

Typeinschränkungen (Klasse und Struktur)

Es ist möglich, festzulegen , ob das Argument Typ unter Verwendung der jeweiligen Einschränkungen ein Referenztyp oder ein Werttyp sein sollte `class` oder `struct` . Wenn diese Einschränkungen verwendet werden, *müssen* sie definiert werden, *bevor alle* anderen Einschränkungen (z. B. ein übergeordneter Typ oder `new()`) aufgelistet werden können.

```
// 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
    {
    }
}
```

Typeinschränkungen (neues Schlüsselwort)

Mithilfe der `new()` Einschränkung können Sie Typparameter erzwingen, um einen leeren (Standard-) Konstruktor zu definieren.

```
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.
```

Der zweite Aufruf von `Create()` führt zu einer Kompilierzeit mit der folgenden Nachricht:

> 'Bar' muss ein nicht abstrakter Typ mit einem öffentlichen parameterlosen Konstruktor sein, um ihn als Parameter 'T' im generischen Typ oder der generischen Methode 'Factory' verwenden zu können.

Es gibt keine Einschränkung für einen Konstruktor mit Parametern. Es werden nur parameterlose Konstruktoren unterstützt.

Typschluss (Klassen)

Entwickler können durch die Tatsache herausgefunden werden, dass die Typinferenz für Konstruktoren *nicht funktioniert* :

```
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.
```

Die erste Möglichkeit, eine Instanz ohne explizite Angabe von Typparametern zu erstellen, führt zu einem Fehler bei der Kompilierung, der Folgendes besagt:

> Die Verwendung des generischen Typs 'Tuple <T1, T2>' erfordert zwei Typargumente

Eine häufige Problemumgehung ist das Hinzufügen einer Hilfsmethode in einer statischen Klasse:

```
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...
```

Reflektieren von Typparametern

Der Operator `typeof` arbeitet mit Typparametern.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Explizite Typparameter

Es gibt verschiedene Fälle, in denen Sie die Typparameter für eine generische Methode explizit angeben müssen. In den beiden folgenden Fällen kann der Compiler nicht alle Typparameter aus den angegebenen Methodenparametern ableiten.

Ein Fall ist, wenn es keine Parameter gibt:

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
}

SomeMethod(); // doesn't compile
```

```
SomeMethod<int, bool>(); // compiles
```

Der zweite Fall ist, wenn einer (oder mehrere) der Typparameter nicht Teil der Methodenparameter ist:

```
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.
```

Verwendung einer generischen Methode mit einer Schnittstelle als Einschränkungstyp.

Dies ist ein Beispiel für die Verwendung der generischen Methode TFood inside Eat für die Klasse Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
```

```
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

Sie können die Eat-Methode folgendermaßen aufrufen:

```
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
```

In diesem Fall, wenn Sie versuchen anzurufen:

```
sheep.Eat(lion);
```

Dies ist nicht möglich, da der Objektlöwe die Schnittstelle IFood nicht implementiert. Wenn Sie versuchen, den obigen Aufruf auszuführen, wird ein Compiler-Fehler generiert: "Der Typ 'Carnivore' kann nicht als Typparameter 'TFood' in der generischen Art oder Methode 'Animal.Eat (TFood)' verwendet werden. Es gibt keine implizite Referenzkonvertierung von ' Fleischfresser 'zu' IFood '. "

Kovarianz

Wann ist ein `IEnumerable<T>` ein Subtyp eines anderen `IEnumerable<T1>` ? Wenn `T` ein Untertyp von `T1` . `IEnumerable` ist in seinem `T` Parameter *kovariant* , was bedeutet, dass die Subtyp-Beziehung von `IEnumerable` in *dieselbe Richtung geht* wie die von `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>
```

Eine Instanz eines kovarianten generischen Typs mit einem bestimmten Typparameter ist implizit in denselben generischen Typ mit einem weniger abgeleiteten Typparameter konvertierbar.

Diese Beziehung hält , weil `IEnumerable` *produziert* `T` s sie aber nicht verbrauchen. Ein Objekt , das erzeugt `Dog` s kann verwendet werden , als ob es produziert `Animal` s.

Covariante Typparameter werden mit dem Schlüsselwort `out` deklariert, da der Parameter nur als

Ausgabe verwendet werden muss .

```
interface IEnumerable<out T> { /* ... */ }
```

Ein als kovariant deklarierter Typparameter wird möglicherweise nicht als Eingabe angezeigt.

```
interface Bad<out T>
{
    void SetT(T t);  // type error
}
```

Hier ist ein vollständiges Beispiel:

```
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
      }
   }
}
```

Verstöße

Wann ist ein `IComparer<T>` ein Subtyp eines anderen `IComparer<T1>` ? Wenn `T1` ein Subtyp von `T` . `IComparer` ist in seinem `T` Parameter *kontravariant* , was bedeutet, dass die Untertypenbeziehung von `IComparer` in die *entgegengesetzte Richtung geht* wie `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

Eine Instanz eines kontravarianten generischen Typs mit einem bestimmten Typparameter ist implizit in denselben generischen Typ mit einem stärker abgeleiteten Typparameter konvertierbar.

Diese Beziehung hält , weil `IComparer` *verbraucht* `T` s sie aber nicht produzieren. Ein Objekt, das zwei beliebige `Animal` miteinander vergleichen kann, kann zum Vergleichen zweier `Dog` .

Kontravariante Typparameter werden mit dem Schlüsselwort `in` deklariert, da der Parameter nur als *Eingabe verwendet werden muss* .

```
interface IComparer<in T> { /* ... */ }
```

Ein als widerspruchsfrei deklarierter Typparameter wird möglicherweise nicht als Ausgabe angezeigt.

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

Invarianz

`IList<T>` ist niemals ein Subtyp einer anderen `IList<T1>` . `IList` ist in ihrem Typparameter *invariant* .

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }
```

```
IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

Es gibt keine Untertypenbeziehung für Listen, da Sie Werte in eine Liste aufnehmen *und* Werte aus einer Liste entnehmen können.

Wenn `IList` kovariant war, können Sie der angegebenen Liste Elemente des *falschen Untertyps* hinzufügen.

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

Wenn `IList` , können Sie Werte aus einem bestimmten Untertyp aus einer bestimmten Liste extrahieren.

```
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!
```

Invariant Typ - Parameter werden durch Weglassen sowohl die erklärt `in` und `out` Schlüsselwörtern.

```
interface IList<T> { /* ... */ }
```

Variantenschnittstellen

Schnittstellen können Variantentyp-Parameter haben.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

Klassen und Strukturen dürfen jedoch nicht

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
}
```

auch keine generischen Methodendeklarationen

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
```

```
    {
        // ...
    }
}
```

Das folgende Beispiel zeigt mehrere Abweichungsdeklarationen auf derselben Schnittstelle

```
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>
```

Variantendelegierte

Delegierte können Variantentypparameter haben.

```
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
```

Dies folgt aus dem Liskov-Substitutionsprinzip , das unter anderem festlegt , dass eine Methode D als abgeleiteter angesehen werden kann als eine Methode B, wenn:

- D hat einen gleichen oder mehr abgeleiteten Rückgabetyp als B
- D hat gleiche oder allgemeinere entsprechende Parametertypen als B

Daher sind die folgenden Zuweisungen alle typsicher:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

Variantentypen als Parameter und Rückgabewerte

Wenn ein kovarianter Typ als Ausgabe angezeigt wird, ist der enthaltende Typ kovariant. Die Herstellung eines Produzenten von `T` ist wie das Herstellen von `T` .

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

Wenn ein kontravarianter Typ als Ausgabe angezeigt wird, ist der enthaltende Typ kontravariant. Einen Konsumenten von `T` ist wie der Konsum von `T` .

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

Wenn ein kovarianter Typ als Eingabe erscheint, ist der enthaltende Typ kontravariant. Der Konsum eines Produzenten von `T` ist wie der Konsum von `T` .

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

Wenn ein kontravarianter Typ als Eingabe erscheint, ist der enthaltende Typ kovariant. Ein Konsument von `T` konsumieren ist wie das Produzieren von `T` .

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Gleichheit der generischen Werte prüfen.

Wenn die Logik einer generischen Klasse oder Methode die Überprüfung der Gleichheit von Werten mit generischem Typ erfordert, verwenden Sie `EqualityComparer<TType>.Default` Eigenschaft :

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
}
```

Dieser Ansatz ist besser als der Aufruf der `Object.Equals()` -Methode, da die Standardvergleichsimplementierung prüft, ob der `TBar` Typ die `IEquatale<TBar>` Schnittstelle `IEquatale<TBar>` implementiert, und wenn ja, die `IEquatable<TBar>.Equals(TBar other)` -Methode. Dadurch kann das Boxen / Unboxing von Werttypen vermieden werden.

Generisches Gussteil

```
    /// <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));
        }



    }
```

Verbrauch:

```
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);
```

Konfigurationsleser mit generischem Typ Casting

```
    /// <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));
        }

    }
```

Verbrauch:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Generics online lesen: https://riptutorial.com/de/csharp/topic/27/generics

Kapitel 65: Generischer Lambda Query Builder

Bemerkungen

Die Klasse heißt `ExpressionBuilder` . Es hat drei Eigenschaften:

```
 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) });
```

Eine öffentliche Methode `GetExpression` , die den Lambda-Ausdruck zurückgibt, und drei private Methoden:

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

Alle Methoden werden in den Beispielen ausführlich erläutert.

Examples

QueryFilter-Klasse

Diese Klasse enthält Prädikatfilterwerte.

```
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;
    }
}
```

Aufzählung für die Operationswerte:

```
 public enum Operator
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

GetExpression-Methode

```
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 Private Überladung

Für einen Filter:

Hier wird die Abfrage erstellt, sie empfängt einen Ausdrucksparameter und einen Filter.

```
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;
}
```

Für zwei Filter:

Es gibt die BinaryExpresion-Instanz anstelle des einfachen Ausdrucks zurück.

```
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-Methode

`ConstantExpression` muss vom selben Typ sein wie `MemberExpression` . Der Wert in diesem Beispiel ist eine Zeichenfolge, die konvertiert wird, bevor die `ConstantExpression` Instanz erstellt wird.

```
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;
}
```

Verwendungszweck

Sammlungsfilter = neue Liste (); QueryFilter filter = new QueryFilter ("Name", "Burger", Operator.StartsWith); filters.Add (filter);

```
Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

In diesem Fall handelt es sich um eine Abfrage gegen die Entität Food, die alle Nahrungsmittel finden soll, die im Namen mit "Burger" beginnen.

Ausgabe:

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Generischer Lambda Query Builder online lesen:
https://riptutorial.com/de/csharp/topic/6721/generischer-lambda-query-builder

Kapitel 66: Geprüft und nicht geprüft

Syntax

- checked (a + b) // überprüfter Ausdruck
- nicht angehakt (a + b) // ungeprüfter Ausdruck
- geprüft {c = a + b; c + = 5; } // Block geprüft
- ungeprüft {c = a + b; c + = 5; } // ungeprüfter Block

Examples

Geprüft und nicht geprüft

C # -Anweisungen werden entweder im geprüften oder ungeprüften Kontext ausgeführt. In einem überprüften Kontext löst ein arithmetischer Überlauf eine Ausnahme aus. In einem ungeprüften Kontext wird der arithmetische Überlauf ignoriert und das Ergebnis wird abgeschnitten.

```
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
```

Wenn keine davon angegeben wird, hängt der Standardkontext von anderen Faktoren ab, z. B. von Compileroptionen.

Als Bereich markiert und nicht markiert

Die Schlüsselwörter können auch Bereiche erstellen, um mehrere Vorgänge (un) zu überprüfen.

```
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
}
```

Geprüft und nicht geprüft online lesen: https://riptutorial.com/de/csharp/topic/2394/gepruft-und-nicht-gepruft

Kapitel 67: Gleichheitsoperator

Examples

Gleichheitsarten in c # und Gleichheitsoperator

In C # gibt es zwei verschiedene Arten von Gleichheit: Referenzgleichheit und Wertgleichheit. Wertgleichheit ist die allgemein verstandene Bedeutung von Gleichheit: Dies bedeutet, dass zwei Objekte die gleichen Werte enthalten. Beispielsweise haben zwei Ganzzahlen mit dem Wert 2 die Wertgleichheit. Referenzgleichheit bedeutet, dass nicht zwei Objekte verglichen werden müssen. Stattdessen gibt es zwei Objektverweise, die beide auf dasselbe Objekt verweisen.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

Bei vordefinierten Werttypen gibt der Gleichheitsoperator (==) true zurück, wenn die Werte seiner Operanden gleich sind, andernfalls false. Bei anderen Referenztypen als string gibt == true zurück, wenn seine beiden Operanden auf dasselbe Objekt verweisen. Für den Zeichenfolgentyp vergleicht == die Werte der Zeichenfolgen.

```
// 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);
```

Gleichheitsoperator online lesen: https://riptutorial.com/de/csharp/topic/1491/gleichheitsoperator

Kapitel 68: Google-Kontakte importieren

Bemerkungen

Die Kontaktdaten der Benutzer werden im JSON-Format empfangen, wir extrahieren sie und schließlich durchlaufen wir diese Daten. Dadurch erhalten wir die Google-Kontakte.

Examples

Bedarf

Laden Sie zum Importieren von Google-Kontakten (Google Mail) in der ASP.NET-MVC-Anwendung zunächst "Google API-Setup" herunter. Dadurch werden die folgenden Referenzen gewährt:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Fügen Sie diese der entsprechenden Anwendung hinzu.

Quellcode in der Steuerung

```
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;

        }
    }
}
```

Quellcode in der Ansicht.

Die einzige Aktionsmethode, die Sie hinzufügen müssen, ist das Hinzufügen eines Aktionslinks unten

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Google-Kontakte importieren online lesen: https://riptutorial.com/de/csharp/topic/6744/google-kontakte-importieren

Kapitel 69: Guid

Einführung

GUID (oder UUID) ist eine Abkürzung für "Globally Unique Identifier" (oder "Universally Unique Identifier"). Es ist eine 128-Bit-Ganzzahl, die zur Identifizierung von Ressourcen verwendet wird.

Bemerkungen

`Guid` sind *global eindeutige Bezeichner* , auch als *UUIDs* (*Universally Unique Identifiers) bezeichnet* .

Sie sind 128-Bit-Pseudozufallswerte. Es gibt so viele gültige `Guid` (ungefähr 10 ^ 18 `Guid` für jede Zelle jedes Menschen auf der Erde), dass sie, wenn sie durch einen guten Pseudozufallsalgorithmus erzeugt werden, mit allen praktischen Mitteln als einzigartig im gesamten Universum betrachtet werden können.

`Guid` werden meistens als Primärschlüssel in Datenbanken verwendet. Ihr Vorteil ist, dass Sie die Datenbank nicht aufrufen müssen, um eine (fast) garantierte neue ID zu erhalten.

Examples

Die Zeichenfolgendarstellung einer Guid abrufen

Eine Zeichenfolgendarstellung einer Guid kann mithilfe der integrierten `ToString` Methode `ToString` werden

```
string myGuidString = myGuid.ToString();
```

Je nach Ihren Anforderungen können Sie die Guid auch formatieren, indem Sie dem `ToString` Aufruf ein `ToString` Argument `ToString` .

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None          "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens       "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces        "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses   "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex           "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

Guid erstellen

Dies sind die häufigsten Methoden zum Erstellen einer Guid-Instanz:

- Erstellen einer leeren Guid (`00000000-0000-0000-0000-000000000000`):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Erstellen einer neuen (pseudozufälligen) Guid:

```
Guid g = Guid.NewGuid();
```

- Hilfslinien mit einem bestimmten Wert erstellen:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Deklarieren einer nullfähigen GUID

Wie andere Werttypen verfügt auch GUID über einen nullwertfähigen Typ, der einen Nullwert annehmen kann.

Erklärung:

```
Guid? myGuidVar = null;
```

Dies ist besonders nützlich, wenn Daten aus der Datenbank abgerufen werden, wenn die Möglichkeit besteht, dass der Wert aus einer Tabelle NULL ist.

Guid online lesen: https://riptutorial.com/de/csharp/topic/1153/guid

Kapitel 70: Hash-Funktionen

Bemerkungen

MD5 und SHA1 sind unsicher und sollten vermieden werden. Die Beispiele existieren zu Lernzwecken und aufgrund der Tatsache, dass ältere Software diese Algorithmen möglicherweise noch verwendet.

Examples

MD5

Hash-Funktionen ordnen binäre Zeichenfolgen beliebiger Länge zu kleinen binären Zeichenfolgen fester Länge zu.

Der `MD5` Algorithmus ist eine weit verbreitete Hash-Funktion, die einen 128-Bit-Hash-Wert (16 Bytes, 32 Hexdecimal-Zeichen) erzeugt.

Die `ComputeHash` Methode der `System.Security.Cryptography.MD5` Klasse gibt den Hash als Array von 16 Byte zurück.

Beispiel:

```
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);
        }
    }
}
```

Ausgabe: Der MD5-Hash von Hello World! ist:
ED076287532E86365E841E92BFC50D8C

Sicherheitsprobleme:

Wie die meisten Hash-Funktionen ist MD5 weder verschlüsselt noch verschlüsselt. Sie kann durch Brute-Force-Angriffe rückgängig gemacht werden und leidet unter erheblichen Schwachstellen gegen Kollisions- und Präimageangriffe.

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);
            }
        }
    }
 }
```

Ausgabe:

Der SHA1-Hash von Hello Word! ist: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

SHA256

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
```

```
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Ausgabe:

Der SHA256 Hash von Hello World! ist:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

SHA384

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Ausgabe:

Der SHA384-Hash von Hello World! ist:
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);
            }
        }
    }
}
```

Ausgabe: Der SHA512-Hash von Hello World! ist:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

PBKDF2 für Passwort-Hashing

PBKDF2 ("Passwortbasierte Schlüsselableitungsfunktion 2") ist eine der empfohlenen Hash-Funktionen für das Passwort-Hashing. Es ist Teil von RFC-2898 .

Die `Rfc2898DeriveBytes` Klasse von .NET basiert auf HMACSHA1.

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2 erfordert ein Salt und die Anzahl der Iterationen.

Iterationen:

Eine hohe Anzahl von Iterationen verlangsamt den Algorithmus, was das Cracken von Passwörtern erheblich erschwert. Daher wird eine hohe Anzahl von Iterationen empfohlen. PBKDF2 ist beispielsweise um Größenordnungen langsamer als MD5.

Salz:

Ein Salt verhindert das Nachschlagen von Hashwerten in Regenbogentabellen. Es muss zusammen mit dem Passwort-Hash gespeichert werden. Es wird ein Salz pro Kennwort (nicht ein globales Salz) empfohlen.

Komplette Passwort-Hashing-Lösung mit Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
  {
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could
be persisted as bytes or as a string using the convenience methods in the next class to
convert from byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary
based attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
```

```
      }
    }
    /// <summary>
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
```

```
  public sealed class PasswordHashContainer
  {
    /// <summary>
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
```

```
* modification, are permitted provided that the following conditions are met:
*
* 1. Redistributions of source code must retain the above copyright notice,
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/
```

Weitere Informationen finden Sie in dieser hervorragenden Ressource Crackstation - Salted Password Hashing - Tuning Right . Ein Teil dieser Lösung (die Hash-Funktion) basierte auf dem Code dieser Site.

Hash-Funktionen online lesen: https://riptutorial.com/de/csharp/topic/2774/hash-funktionen

Kapitel 71: HTTP-Anfragen ausführen

Examples

HTTP-POST-Anforderung erstellen und senden

```
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();
    }
}
```

Erstellen und Senden einer HTTP-GET-Anforderung

```
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();
    }
}
```

Fehlerbehandlung bestimmter HTTP-Antwortcodes (z. B. 404 Not Found)

```
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.
        // ...
    }
}
```

Senden einer asynchronen HTTP-POST-Anforderung mit JSON-Body

```
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" });
```

Senden einer asynchronen HTTP-GET-Anforderung und Lesen der JSON-Anforderung

```
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>();
```

HTML für Webseite abrufen (einfach)

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

HTTP-Anfragen ausführen online lesen: https://riptutorial.com/de/csharp/topic/1971/http-anfragen-ausfuhren

Kapitel 72: ICloneable

Syntax

- object ICloneable.Clone () {return Clone (); } // Private Implementierung der Schnittstellenmethode, die unsere benutzerdefinierte öffentliche Clone () - Funktion verwendet.
- public Foo Clone () {return new Foo (this); } // Die öffentliche Klonmethode sollte die Kopierkonstruktorlogik verwenden.

Bemerkungen

Die `CLR` erfordert ein Methodendefinitionsobjekt `object Clone()` das nicht typsicher ist. Es ist üblich, dieses Verhalten zu überschreiben und eine typsichere Methode zu definieren, die eine Kopie der enthaltenden Klasse zurückgibt.

Es ist Sache des Autors, zu entscheiden, ob Klonen nur flache oder tiefe Kopie bedeutet. Für unveränderliche Strukturen mit Referenzen wird empfohlen, eine tiefe Kopie zu erstellen. Für Klassen, die selbst Referenzen sind, ist es wahrscheinlich in Ordnung, eine flache Kopie zu implementieren.

HINWEIS: In `C#` eine Schnittstellenmethode mit der oben gezeigten Syntax privat implementiert werden.

Examples

ICloneable in einer Klasse implementieren

Implementiere `ICloneable` in einer Klasse mit einem Twist. Machen Sie ein öffentliches sicheres `Clone()` und implementieren Sie das `object Clone()` privat.

```
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
}
```

Später wie folgt zu verwenden:

```
{
    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);
}
```

Beachten Sie, dass das Ändern des `bob` Alters das Alter von `bob_clone` nicht ändert. Dies liegt daran, dass das Design Klonen verwendet, anstatt (Referenz-) Variablen zuzuweisen.

ICloneable in einer Struktur implementieren

Die Implementierung von ICloneable für eine Struktur ist im Allgemeinen nicht erforderlich, da die Strukturen mit dem Zuweisungsoperator `=` eine mitgliederweise Kopie durchführen. Das Design erfordert jedoch möglicherweise die Implementierung einer anderen Schnittstelle, die von `ICloneable` .

Ein anderer Grund wäre, wenn die Struktur einen Referenztyp (oder ein Array) enthält, der ebenfalls kopiert werden muss.

```
// 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
}
```

Später wie folgt zu verwenden:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

ICloneable online lesen: https://riptutorial.com/de/csharp/topic/7917/icloneable

Kapitel 73: IDisposable-Schnittstelle

Bemerkungen

- Es ist `IDisposable` Clients der Klasse, die `IDisposable` implementiert, um sicherzustellen, dass sie die `Dispose` Methode aufrufen, wenn sie das Objekt `IDisposable` verwendet haben. Die CLR enthält nichts, das Objekte direkt nach einer `Dispose` Methode sucht, die `Dispose` werden soll.
- Es ist nicht erforderlich, einen Finalizer zu implementieren, wenn Ihr Objekt nur verwaltete Ressourcen enthält. Rufen Sie `Dispose` für alle Objekte auf, die Ihre Klasse verwendet, wenn Sie Ihre eigene `Dispose` Methode implementieren.
- Es wird empfohlen, die Klasse gegen mehrere Aufrufe von `Dispose` zu `Dispose` , obwohl sie idealerweise nur einmal aufgerufen werden sollte. Dies kann erreicht werden, indem Sie Ihrer Klasse eine `private bool` Variable hinzufügen und den Wert auf `true` wenn die `Dispose` Methode ausgeführt wird.

Examples

In einer Klasse, die nur verwaltete Ressourcen enthält

Verwaltete Ressourcen sind Ressourcen, die der Garbage Collector der Laufzeitumgebung kennt und unter deren Kontrolle steht. In der BCL sind viele Klassen verfügbar, beispielsweise eine `SqlConnection` , die eine Wrapper-Klasse für eine nicht verwaltete Ressource ist. Diese Klassen implementieren bereits die `IDisposable` Schnittstelle - es liegt an Ihrem Code, sie zu bereinigen, wenn Sie fertig sind.

Es ist nicht erforderlich, einen Finalizer zu implementieren, wenn Ihre Klasse nur verwaltete Ressourcen enthält.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

In einer Klasse mit verwalteten und nicht verwalteten Ressourcen

Es ist wichtig, dass die Finalisierung verwaltete Ressourcen ignoriert. Der Finalizer läuft in einem anderen Thread - es ist möglich, dass die verwalteten Objekte zum Zeitpunkt der Finalisierung nicht mehr vorhanden sind. Die Implementierung einer geschützten `Dispose(bool)` -Methode ist eine gängige Praxis, um sicherzustellen, dass für verwaltete Ressourcen keine `Dispose` Methode

von einem Finalizer aufgerufen wird.

```
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, entsorgen

.NET Framework definiert eine Schnittstelle für Typen, die eine Abreißmethode erfordern:

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()` wird hauptsächlich für die Bereinigung von Ressourcen verwendet, beispielsweise für nicht verwaltete Referenzen. Es kann jedoch auch nützlich sein, die Entsorgung anderer Ressourcen zu erzwingen, obwohl diese verwaltet werden. Anstatt darauf zu warten, dass der GC schließlich auch Ihre Datenbankverbindung bereinigt, können Sie sicherstellen, dass dies in Ihrer eigenen `Dispose()` Implementierung der `Dispose()` .

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
```

```
    {
        this.CurrentDatabaseConnection.Dispose();
        this.CurrentDatabaseConnection = null;
    }
}
```

Wenn Sie direkt auf nicht verwaltete Ressourcen wie nicht verwaltete Zeiger oder win32-Ressourcen zugreifen müssen, erstellen Sie eine Klasse, die von `SafeHandle` und verwenden Sie die Konventionen / Tools dieser Klasse.

In einer geerbten Klasse mit verwalteten Ressourcen

Es ist `IDisposable` üblich, dass Sie eine Klasse erstellen, die `IDisposable` implementiert, und dann Klassen ableiten, die auch verwaltete Ressourcen enthalten. Es wird empfohlen, die `Dispose` Methode mit dem `virtual` Schlüsselwort zu kennzeichnen, damit Clients alle Ressourcen bereinigen können, die sie besitzen.

```
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();
    }
}
```

Schlüsselwort verwenden

Wenn ein Objekt die `IDisposable` Schnittstelle implementiert, kann es innerhalb der `using` Syntax erstellt werden:

```
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");
    }
}
```

Demo anzeigen

`using` ist syntatic Zucker für einen `try/finally` Block; Die obige Verwendung würde grob bedeuten:

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

IDisposable-Schnittstelle online lesen: https://riptutorial.com/de/csharp/topic/1795/idisposable-schnittstelle

Kapitel 74: IEnumerable

Einführung

`IEnumerable` ist die Basisschnittstelle für alle nicht generischen Sammlungen wie ArrayList, die aufgezählt werden können. `IEnumerator<T>` ist die Basisschnittstelle für alle generischen Enumeratoren wie List <>.

`IEnumerable` ist eine Schnittstelle, die die Methode `GetEnumerator` . Die `GetEnumerator` Methode gibt einen `IEnumerator` der Optionen zum Durchlaufen der Auflistung wie foreach enthält.

Bemerkungen

IEnumerable ist die Basisschnittstelle für alle nicht generischen Sammlungen, die aufgelistet werden können

Examples

IEnumerable

In seiner einfachsten Form repräsentiert ein Objekt, das IEnumerable implementiert, eine Reihe von Objekten. Die fraglichen Objekte können mit dem Schlüsselwort c # `foreach` wiederholt werden.

Im folgenden Beispiel implementiert das Objekt `sequenceOfNumbers` IEnumerable. Es repräsentiert eine Reihe von ganzen Zahlen. Die `foreach` Schleife durchläuft nacheinander jede.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

IEnumerable mit benutzerdefiniertem Enumerator

Durch die Implementierung der IEnumerable-Schnittstelle können Klassen auf dieselbe Weise wie BCL-Sammlungen aufgelistet werden. Dies erfordert die Erweiterung der Enumerator-Klasse, die den Status der Aufzählung verfolgt.

Neben dem Durchlaufen einer Standardsammlung sind folgende Beispiele zu nennen:

- Verwenden von Zahlenbereichen basierend auf einer Funktion anstelle einer Sammlung von Objekten
- Implementierung verschiedener Iterationsalgorithmen über Sammlungen, wie DFS oder BFS

in einer Diagrammsammlung

```
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;
        }
    }
}
```

IEnumerable online lesen: https://riptutorial.com/de/csharp/topic/2220/ienumerable

Kapitel 75: ILGenerator

Examples

Erstellt eine DynamicAssembly, die eine UnixTimestamp-Hilfemethode enthält

In diesem Beispiel wird die Verwendung des ILGenerator veranschaulicht, indem Code generiert wird, der bereits vorhandene und neu erstellte Member sowie die grundlegende Behandlung von Ausnahmen verwendet. Der folgende Code gibt eine DynamicAssembly aus, die ein Äquivalent zu diesem C # -Code enthält:

```
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");
```

Methodenüberschreibung erstellen

Dieses Beispiel zeigt, wie die `ToString` Methode in der generierten Klasse überschrieben wird

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

ILGenerator online lesen: https://riptutorial.com/de/csharp/topic/667/ilgenerator

Kapitel 76: Implementierung des Decorator Design Pattern

Bemerkungen

Vorteile der Verwendung von Decorator:

- Sie können zur Laufzeit neue Funktionen in verschiedenen Konfigurationen hinzufügen
- gute alternative für vererbung
- Der Kunde kann die Konfiguration auswählen, die er verwenden möchte

Examples

Cafeteria simulieren

Dekorateur ist eines der strukturellen Entwurfsmuster. Es wird verwendet, um das Verhalten eines Objekts hinzuzufügen, zu entfernen oder zu ändern. In diesem Dokument erfahren Sie, wie Sie Decorator DP richtig verwenden.

Lassen Sie mich Ihnen die Idee an einem einfachen Beispiel erklären. Stellen Sie sich vor, Sie sind jetzt in Starbobs, einer berühmten Kaffeegesellschaft. Sie können einen beliebigen Kaffee bestellen - mit Sahne und Zucker, mit Sahne und Topping und viel mehr Kombinationen! Die Basis aller Getränke ist jedoch Kaffee - dunkles, bitteres Getränk, das Sie modifizieren können. Lassen Sie uns ein einfaches Programm schreiben, das eine Kaffeemaschine simuliert.

Zuerst müssen wir eine Klasse erstellen und abstrahieren, die unser Basisgetränk beschreibt:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Lassen Sie uns nun einige Extras wie Zucker, Milch und Topping schaffen. Die erstellten Klassen müssen `AbstractCoffee` implementieren - sie werden es dekorieren:

```
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";
    }
}
```

Jetzt können wir unseren Lieblingskaffee kreieren:

```
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());

    }
}
```

Beim Ausführen des Codes wird die folgende Ausgabe erzeugt:

> Kaffee mit Belag mit Zucker mit Milch

Implementierung des Decorator Design Pattern online lesen:
https://riptutorial.com/de/csharp/topic/4798/implementierung-des-decorator-design-pattern

Kapitel 77: Indexer

Syntax

- public ReturnType [IndexType-Index] {get {...} set {...}}

Bemerkungen

Mit dem Indexer kann mit einer Array-ähnlichen Syntax auf eine Eigenschaft eines Objekts mit einem Index zugegriffen werden.

- Kann für eine Klasse, Struktur oder Schnittstelle verwendet werden.
- Kann überladen werden.
- Kann mehrere Parameter verwenden.
- Kann verwendet werden, um auf Werte zuzugreifen und sie einzustellen.
- Kann einen beliebigen Typ für den Index verwenden.

Examples

Ein einfacher Indexer

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Verwendungszweck:

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

Demo anzeigen

Indexer mit 2 Argumenten und Schnittstelle

```
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;
        }
    }
}
```

Überladen des Indexers zum Erstellen eines SparseArray

Durch Überladen des Indexers können Sie eine Klasse erstellen, die wie ein Array aussieht und sich anfühlt, aber nicht ist. Es gibt O (1) get- und set-Methoden, kann auf ein Element im Index 100 zugreifen und hat dennoch die Größe der Elemente in sich. Die SparseArray-Klasse

```
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);
            }
        }
    }
```

Indexer online lesen: https://riptutorial.com/de/csharp/topic/1660/indexer

Kapitel 78: INotifyPropertyChanged-Schnittstelle

Bemerkungen

Die Schnittstelle `INotifyPropertyChanged` wird immer dann benötigt, wenn Ihre Klasse die Änderungen an ihren Eigenschaften melden soll. Die Schnittstelle definiert ein einzelnes Ereignis `PropertyChanged` .

Mit XAML Binding wird das `PropertyChanged` Ereignis automatisch verbunden, sodass Sie nur die INotifyPropertyChanged-Schnittstelle in Ihrem Ansichtsmodell oder in den Datenkontextklassen implementieren müssen, um mit XAML Binding arbeiten zu können.

Examples

Implementieren von INotifyPropertyChanged in C # 6

Die Implementierung von `INotifyPropertyChange` kann fehleranfällig sein, da für die Schnittstelle der Eigenschaftsname als String angegeben werden muss. Um die Implementierung robuster zu gestalten, kann ein Attribut `CallerMemberName` verwendet werden.

```
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;
}
```

Wenn Sie über mehrere Klassen `INotifyPropertyChanged` , die `INotifyPropertyChanged`

implementieren, kann es hilfreich sein, die Schnittstellenimplementierung und die `INotifyPropertyChanged` auf die allgemeine Basisklasse `INotifyPropertyChanged` :

```
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 mit der Methode "Generic Set"

Die `NotifyPropertyChangedBase` Klasse definiert eine generische Set-Methode, die von einem beliebigen abgeleiteten Typ aufgerufen werden kann.

```
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;
    }
}
```

Um diese generische Set-Methode verwenden zu können, müssen Sie einfach eine Klasse erstellen, die von NotifyPropertyChangedBase abgeleitet ist.

```
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); }
    }
}
```

Wie oben gezeigt, können Sie `Set(ref _fieldName, value);` im Setter einer Eigenschaft, und es wird automatisch ein PropertyChanged-Ereignis ausgelöst, wenn es erforderlich ist.

Sie können sich dann beim PropertyChanged-Ereignis von einer anderen Klasse registrieren, die Eigenschaftsänderungen behandeln muss.

```
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;

}
```

INotifyPropertyChanged-Schnittstelle online lesen:
https://riptutorial.com/de/csharp/topic/2990/inotifypropertychanged-schnittstelle

Kapitel 79: Interoperabilität

Bemerkungen

Mit Win32-API mit C # arbeiten

Windows bietet viele Funktionen in Form der Win32-API. Mithilfe dieser API können Sie eine direkte Operation in Windows ausführen, wodurch die Leistung Ihrer Anwendung erhöht wird. Quelle Klicken Sie hier

Windows bietet eine breite Palette von APIs. Um Informationen zu verschiedenen APIs zu erhalten, können Sie Websites wie Pinvoke überprüfen.

Examples

Importieren Sie die Funktion aus einer nicht verwalteten C ++ - DLL

Im Folgenden finden Sie ein Beispiel zum Importieren einer Funktion, die in einer nicht verwalteten C ++ - DLL definiert ist. Im C ++ - Quellcode für "myDLL.dll" ist die Funktion `add` definiert:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Dann kann es wie folgt in ein C # -Programm aufgenommen werden:

```
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));
    }
}
```

Erläuterungen dazu, warum `extern "C"` und `__stdcall` erforderlich sind, finden Sie unter Aufrufen von Konventionen und C ++ - Namensveränderung .

Die dynamische Bibliothek finden

Wenn die externe Methode zum ersten Mal aufgerufen wird, sucht das C # -Programm nach der entsprechenden DLL und lädt diese. Weitere Informationen dazu, wo gesucht wird, um die DLL zu finden, und wie Sie die Suchpositionen beeinflussen können, finden Sie in dieser Stackoverflow-Frage .

Einfacher Code, um Klasse für com verfügbar zu machen

```
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 ++ - Namensverstümmelung

C ++ - Compiler codieren zusätzliche Informationen in den Namen exportierter Funktionen, z. B. Argumenttypen, um Überladungen mit verschiedenen Argumenten zu ermöglichen. Dieser Vorgang wird als Namensveränderung bezeichnet . Dies führt zu Problemen mit Funktionen in C # (und Interop mit anderen Sprachen im Allgemeinen), wie der Name des Import `int add(int a, int b)` Funktion ist nicht mehr `add` , kann es sein `?add@@YAHHH@Z` , `_add@8` oder etwas anderes, abhängig vom Compiler und der aufrufenden Konvention.

Es gibt verschiedene Wege, um das Problem der Namensveränderung zu lösen:

- Exportieren von Funktionen mit `extern "C"` zum Umschalten auf externe C-Verknüpfung, die C-Namensverstümmelung verwendet:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  Der Funktionsname wird immer noch `_add@8` (`_add@8`), aber `StdCall` + `extern "C"` Namensveränderungen wird vom C # -Compiler erkannt.

- Exportierte Funktionsnamen in der `myDLL.def` :

```
EXPORTS
  add
```

```
int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  Der Funktionsname wird in diesem Fall rein `add` .

- Verstümmelten Namen importieren Sie benötigen einen DLL-Viewer, um den beschädigten Namen zu sehen. Dann können Sie ihn explizit angeben:

```
__declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

Konventionen aufrufen

Es gibt verschiedene Konventionen für den Aufruf von Funktionen, die angeben, wer (Aufrufer oder Aufpasser) Argumente aus dem Stack abbricht, wie und in welcher Reihenfolge Argumente übergeben werden. C ++ verwendet `Cdecl` die `Cdecl` Aufrufkonvention, C # erwartet jedoch `StdCall` , das normalerweise von der Windows-API verwendet wird. Sie müssen das eine oder das andere ändern:

- Ändern Sie die Aufrufkonvention in `StdCall` in C ++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- Oder ändern Sie die Aufrufkonvention in `Cdecl` in `Cdecl` :

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Wenn Sie eine Funktion mit der `Cdecl` Aufrufkonvention und einem überflüssigen Namen verwenden möchten, `Cdecl` Ihr Code folgendermaßen aus:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** (**__thiscall**) wird hauptsächlich in Funktionen verwendet, die Mitglieder einer Klasse sind.
- Wenn eine Funktion verwendet **Thiscall (__thiscall),** ein Zeiger auf die Klasse wird als der erste Parameter weitergegeben.

Dynamisches Laden und Entladen von nicht verwalteten DLLs

Wenn Sie das Attribut `DllImport` , müssen Sie die korrekte DLL und den richtigen Methodennamen zur *Kompilierzeit kennen* . Wenn Sie flexibler sein und zur *Laufzeit* entscheiden möchten, welche DLLs und Methoden geladen werden sollen, können Sie die Windows-API-Methoden `LoadLibrary()` , `GetProcAddress()` und `FreeLibrary()` . Dies kann hilfreich sein, wenn die zu verwendende Bibliothek von den Laufzeitbedingungen abhängt.

Der Zeiger zurückgegeben durch `GetProcAddress()` kann in einen Delegierten gegossen werden unter Verwendung von `Marshal.GetDelegateForFunctionPointer()` .

Das folgende Codebeispiel demonstriert dies mit der `myDLL.dll` aus den vorherigen Beispielen:

```
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);
    }
}
```

Umgang mit Win32-Fehlern

Wenn Sie Interop-Methoden verwenden, können Sie die **GetLastError-** API verwenden, um zusätzliche Informationen zu **Ihren** API-Aufrufen **abzurufen** .

DllImport-Attribut SetLastError-Attribut

SetLastError = true

Gibt an, dass der Aufgerufene SetLastError (Win32-API-Funktion) aufruft.

SetLastError = false

Gibt an, dass der Angerufene SetLastError (Win32-API-Funktion) **nicht** aufruft. Daher erhalten Sie keine Fehlerinformationen.

- Wenn SetLastError nicht festgelegt ist, wird es auf false gesetzt (Standardwert).
- Sie können den Fehlercode mithilfe der Marshal.GetLastWin32Error-Methode erhalten:

Beispiel:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

Wenn Sie versuchen, einen nicht vorhandenen Mutex zu öffnen, gibt GetLastError **ERROR_FILE_NOT_FOUND zurück** .

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

Systemfehlercodes finden Sie hier:

https://msdn.microsoft.com/de-de/library/windows/desktop/ms681382(v=vs.85).aspx

GetLastError-API

Es gibt eine native **GetLastError-** API, die Sie ebenfalls verwenden können:

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- Wenn Sie die Win32-API über verwalteten Code aufrufen, müssen Sie immer **Marshal.GetLastWin32Error verwenden** .

Hier ist der Grund:

Zwischen Ihrem Win32-Aufruf, der den Fehler festlegt (Aufrufe von SetLastError), kann die CLR andere Win32-Aufrufe aufrufen, die ebenfalls **SetLastError** aufrufen **könnten** . Dieses Verhalten kann Ihren Fehlerwert überschreiben. Wenn Sie in diesem Szenario **GetLastError** aufrufen, **können** Sie einen ungültigen Fehler erhalten.

Durch Festlegen von **SetLastError = true** wird sichergestellt, dass die CLR den Fehlercode abruft, bevor andere Win32-Aufrufe ausgeführt werden.

Gepinntes Objekt

GC (Garbage Collector) ist für die Reinigung unseres Mülls verantwortlich.

Während **GC** unseren Müll säubert, entfernt er die nicht verwendeten Objekte aus dem verwalteten Heap, wodurch eine Heap-Fragmentierung verursacht wird. Wenn der **GC** mit dem Entfernen fertig ist, führt er eine Heap-Komprimierung (Defragmentierung) durch, bei der Objekte auf dem Heap verschoben werden.

Da **GC** nicht deterministisch ist, wenn Objektreferenz / Zeiger auf nativen Code verwaltet geben, können **GC** jederzeit eintreten, wenn es nur nach Inerop Anruf auftritt, gibt es eine sehr gute Möglichkeit, dass Objekt (die sich auf einheimische bestanden) auf dem verwalteten Heap verschoben werden - als Ergebnis erhalten wir auf der verwalteten Seite eine ungültige Referenz.

In diesem Szenario sollten Sie das Objekt **stecken** , bevor sie an nativen Code übergeben.

Gepinntes Objekt

Ein fixiertes Objekt ist ein Objekt, das von GC nicht verschoben werden darf.

Gc Fixierter Griff

Sie können ein Pin-Objekt mit der **Gc.Alloc-** Methode **erstellen**

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- **Wenn Sie** ein gepinntes **GCHandle-** Objekt an ein verwaltetes Objekt **beziehen** , wird ein bestimmtes Objekt als ein Objekt markiert, das vom **GC** nicht verschoben werden kann, bis der Griff **freigegeben** wird

Beispiel:

```
[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()
    }
}
```

Vorsichtsmaßnahmen

- Beim Feststecken (besonders bei großen Objekten) versuchen Sie, das festgesteckte **GcHandle** so schnell wie möglich **freizugeben** , da dies die Heap-Defragmentierung unterbricht.
- Wenn Sie vergessen, **GcHandle** zu befreien, **wird** nichts **passieren** . Tun Sie es in einem sicheren Code-Abschnitt (wie finaly)

Strukturen lesen mit Marschall

Die Marshal-Klasse enthält eine Funktion namens **PtrToStructure** . Diese Funktion gibt uns die Möglichkeit, Strukturen über einen nicht verwalteten Zeiger zu lesen.

Die PtrToStructure- Funktion hat viele Überladungen, aber alle haben die gleiche Absicht.

Generische **PtrToStructure** :

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - Strukturtyp.

ptr - Ein Zeiger auf einen nicht verwalteten Speicherblock.

Beispiel:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- Wenn Sie sich beim Lesen nativer Strukturen mit verwalteten Objekten beschäftigen, vergessen Sie nicht, Ihr Objekt zu fixieren :)

```
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;
   }
```

Interoperabilität online lesen: https://riptutorial.com/de/csharp/topic/3278/interoperabilitat

Kapitel 80: IQueryable-Schnittstelle

Examples

Übersetzen einer LINQ-Abfrage in eine SQL-Abfrage

Mit den Schnittstellen `IQueryable` und `IQueryable<T>` können Entwickler eine LINQ-Abfrage (eine sprachintegrierte Abfrage) in eine bestimmte Datenquelle, z. B. eine relationale Datenbank, übersetzen. Nehmen Sie diese in C # geschriebene LINQ-Abfrage:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

Wenn die Variable `books` ein Typ ist, der `IQueryable<Book>` implementiert, wird die `IQueryable<Book>` Abfrage in Form eines Ausdrucksbaums an den Anbieter übergeben (in der Eigenschaft `IQueryable.Provider`). `IQueryable.Provider` ist eine Datenstruktur, die die Struktur des Codes widerspiegelt .

Der Provider kann zur Laufzeit den Ausdrucksbaum überprüfen, um Folgendes zu ermitteln:

- dass es ein Prädikat für die `Author` Eigenschaft der `Book` Klasse gibt;
- dass die verwendete Vergleichsmethode gleich ist (`==`);
- dass der Wert, den es gleich sein sollte, `"Stephen King"` .

Mit diesen Informationen kann der Anbieter die C # -Anfrage zur Laufzeit in eine SQL-Abfrage konvertieren und diese Abfrage an eine relationale Datenbank übergeben, um nur die Bücher abzurufen, die dem Prädikat entsprechen:

```
select *
from Books
where Author = 'Stephen King'
```

Der Provider aufgerufen wird , wenn die `query` Variable iteriert (`IQueryable` implementiert `IEnumerable`).

(Der in diesem Beispiel verwendete Provider würde einige zusätzliche Metadaten erfordern, um zu wissen, welche Tabelle abgefragt werden soll, und um zu erfahren, wie die Eigenschaften der C # -Klasse mit den Tabellenspalten `IQueryable` werden. `IQueryable` Metadaten liegen jedoch außerhalb des Bereichs der `IQueryable` Schnittstelle.)

IQueryable-Schnittstelle online lesen: https://riptutorial.com/de/csharp/topic/3094/iqueryable-schnittstelle

Kapitel 81: Iteratoren

Bemerkungen

Ein Iterator ist eine Methode, ein Abruf-Accessor oder ein Operator, der eine benutzerdefinierte Iteration über ein Array oder eine Auflistungsklasse mit dem Schlüsselwort ertrag durchführt

Examples

Einfaches numerisches Iterator-Beispiel

Ein häufiger Anwendungsfall für Iteratoren ist es, einige Operationen mit einer Reihe von Zahlen durchzuführen. Das folgende Beispiel veranschaulicht, wie jedes Element innerhalb eines Zahlenfeldes einzeln auf die Konsole gedruckt werden kann.

Dies ist möglich, weil Arrays die `IEnumerable` Schnittstelle implementieren, sodass Clients mithilfe der `GetEnumerator()` Methode einen Iterator für das Array `GetEnumerator()` . Diese Methode gibt einen *Enumerator zurück* , der ein schreibgeschützter Vorwärtscursor über jeder Zahl im Array ist.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Ausgabe

```
1
2
3
4
5
```

Es ist auch möglich, die gleichen Ergebnisse mit einer `foreach` Anweisung zu erzielen:

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Erstellen von Iteratoren mithilfe von Ertrag

Iteratoren *erzeugen* Enumeratoren. In C #, werden durch die Definition Enumeratoren Methoden, Eigenschaften oder Indexer hergestellt , die enthalten `yield` Aussagen.

Die meisten Methoden geben die Kontrolle über ihre normalen `return` Anweisungen an ihren Aufrufer `return` . Im Gegensatz dazu Methoden , die verwendet wird `yield` Aussagen erlauben sie mehrere Werte an den Aufrufer auf Anfrage zurückzukehren , während lokalen Zustand in-zwischen diesen Werten zu *bewahren* zurück. Diese zurückgegebenen Werte bilden eine Sequenz. Es gibt zwei Arten von `yield` , die in Iteratoren verwendet werden:

- `yield return` , die die Kontrolle an den Anrufer zurückgibt, aber den Status beibehält. Der Angerufene setzt die Ausführung ab dieser Zeile fort, wenn die Kontrolle wieder an ihn übergeben wird.

- `yield break` , die ähnlich wie eine normale `return` Anweisung funktioniert - dies bedeutet das Ende der Sequenz. Normale `return` selbst sind innerhalb eines Iteratorblocks unzulässig.

Dieses Beispiel veranschaulicht eine Iterator-Methode, mit der die Fibonacci-Sequenz generiert werden kann :

```
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;
    }
}
```

Dieser Iterator kann dann verwendet werden, um einen Enumerator der Fibonacci-Sequenz zu erzeugen, der von einer aufrufenden Methode verarbeitet werden kann. Der folgende Code veranschaulicht, wie die ersten zehn Terme in der Fibonacci-Sequenz aufgelistet werden können:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Ausgabe

```
1
1
2
3
5
8
13
21
34
55
```

Iteratoren online lesen: https://riptutorial.com/de/csharp/topic/4243/iteratoren

Kapitel 82: Json.net verwenden

Einführung

Verwendung der JSON.net JsonConverter- Klasse.

Examples

Verwendung von JsonConverter für einfache Werte

Beispiel mit JsonCoverter zur Deserialisierung der Laufzeiteigenschaft aus der API-Antwort in ein Timespan- Objekt im Movies-Modell

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"
}
```

Filmmodell

```
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);
        }
    }
}
```

Es anrufen

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

Sammeln Sie alle Felder des JSON-Objekts

```
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;
}
```

Verwendungszweck:

```
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}'");
```

Demo
Für dieses JSON-Objekt

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

erwartete Ausgabe wird sein:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

Json.net verwenden online lesen: https://riptutorial.com/de/csharp/topic/9879/json-net-verwenden

Kapitel 83: Kodex-Verträge und Zusicherungen

Examples

Assertions zur Überprüfung der Logik sollten immer wahr sein

Assertions werden nicht dazu verwendet, Eingabeparameter zu testen, sondern um sicherzustellen, dass der Programmablauf korrekt ist, dh dass Sie zu einem bestimmten Zeitpunkt bestimmte Annahmen über Ihren Code treffen können. Mit anderen Worten: Ein mit `Debug.Assert` durchgeführter Test sollte *immer* davon ausgehen, dass der getestete Wert " `true` .

Debug.Assert wird nur in DEBUG-Builds ausgeführt. Es wird aus RELEASE-Builds herausgefiltert. Es muss als Debugging-Tool zusätzlich zum Komponententest betrachtet werden und nicht als Ersatz für Codeverträge oder Eingabevalidierungsmethoden.

Dies ist zum Beispiel eine gute Behauptung:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Hier ist Assert eine gute Wahl, da wir davon ausgehen können, dass RetrieveSystemConfiguration () einen gültigen Wert zurückgibt und niemals NULL zurückgibt.

Hier ist ein weiteres gutes Beispiel:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

Zunächst können wir davon ausgehen, dass RetrieveUserData () einen gültigen Wert zurückgibt. Vor der Verwendung der Age-Eigenschaft überprüfen wir dann die Annahme (die immer wahr sein sollte), dass das Alter des Benutzers absolut positiv ist.

Dies ist ein schlechtes Beispiel für die Behauptung:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert ist nicht für die Überprüfung von Eingaben, da die Annahme, dass diese Assertion immer wahr ist, falsch ist. Sie müssen dafür Eingabevalidierungsmethoden verwenden. In dem obigen Fall sollten Sie auch sicherstellen, dass der Eingabewert an erster Stelle eine Zahl ist.

Kodex-Verträge und Zusicherungen online lesen:
https://riptutorial.com/de/csharp/topic/4349/kodex-vertrage-und-zusicherungen

Kapitel 84: Kommentare und Regionen

Examples

Bemerkungen

Die Verwendung von Kommentaren in Ihren Projekten ist eine praktische Methode, um Erklärungen zu Ihren Entwurfsentscheidungen zu hinterlassen, und sollte das Leben (oder das einer anderen Person) erleichtern, wenn Sie den Code verwalten oder ergänzen.

Es gibt zwei Möglichkeiten, dem Code einen Kommentar hinzuzufügen.

Einzeilige Kommentare

Jeder nach `//` platzierte Text wird als Kommentar behandelt.

```
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();
    }
}
```

Mehrzeilige oder begrenzte Kommentare

Jeder Text zwischen `/*` und `*/` wird als Kommentar behandelt.

```
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();
    }
```

```
}
```

Regionen

Eine Region ist ein ausblendbarer Codeblock, der die Lesbarkeit und Organisation Ihres Codes verbessern kann.

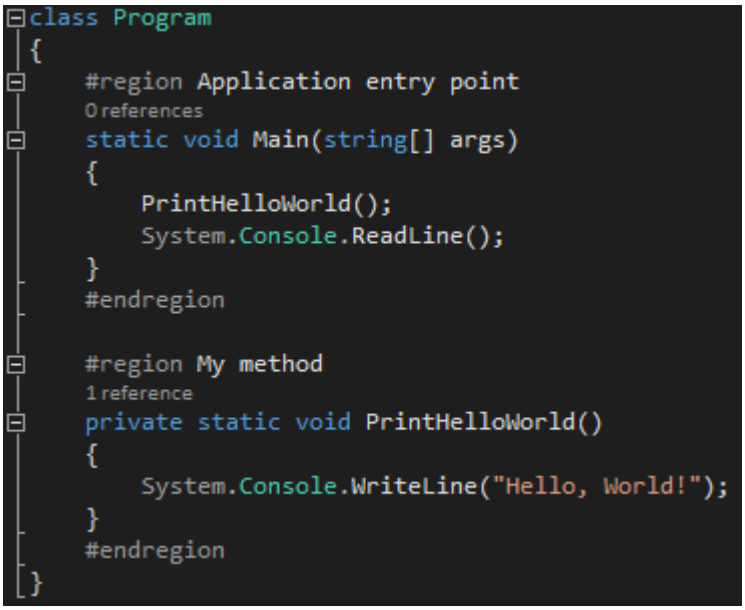
HINWEIS **: Die** StyleCop-Regel SA1124 DoNotUseRegions rät von der Verwendung von Regionen ab. Sie sind in der Regel ein Zeichen für schlecht organisierten Code, da C # Teilklassen und andere Funktionen enthält, die Regionen überflüssig machen.

Sie können Regionen auf folgende Weise verwenden:

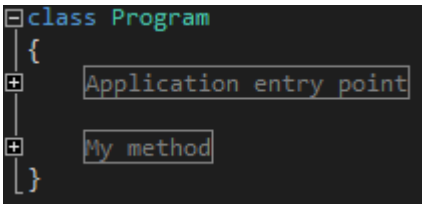
```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

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

Wenn der obige Code in einer IDE angezeigt wird, können Sie den Code mithilfe der + und - Symbole ausblenden und erweitern.

Erweitert



Zusammengebrochen

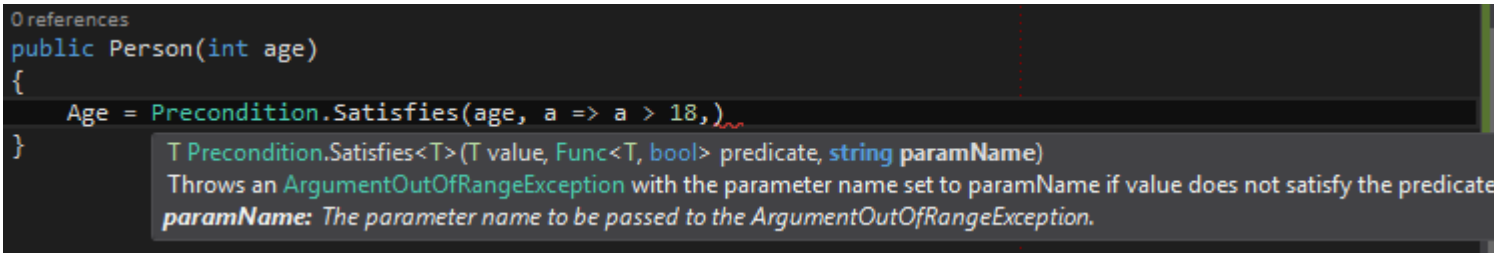


Kommentare zur Dokumentation

XML-Dokumentationskommentare können verwendet werden, um API-Dokumentation bereitzustellen, die von Tools problemlos verarbeitet werden kann:

```
/// <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;
    }
}
```

Die Dokumentation wird von IntelliSense sofort abgeholt:



Kommentare und Regionen online lesen: https://riptutorial.com/de/csharp/topic/5346/kommentare-und-regionen

Kapitel 85: Konsolenanwendung mit einem Nur-Text-Editor und dem C # -Compiler erstellen (csc.exe)

Examples

Erstellen einer Konsolenanwendung mit einem Nur-Text-Editor und dem C # -Compiler

Um mit einem Nur-Text-Editor eine in C # geschriebene Konsolenanwendung zu erstellen, benötigen Sie den C # -Compiler. Der C # -Compiler (csc.exe) befindet sich an folgendem Speicherort: `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

Hinweis: Je nachdem, welche Version von .NET Framework auf Ihrem System installiert ist, müssen Sie möglicherweise den obigen Pfad entsprechend ändern.

Code speichern

Der Zweck dieses Themas besteht nicht darin, Ihnen beizubringen, *wie* Sie eine Konsolenanwendung schreiben, sondern wie Sie eine *kompilieren* [um eine einzige ausführbare Datei zu *erstellen*], außer dem C # -Compiler und einem beliebigen Nur-Text-Editor (z. B. Notizblock).

1. Öffnen Sie das Dialogfeld Ausführen mit der Tastenkombination `Windows-Taste` + `R`
2. `Geben Sie notepad` und `notepad` die `Eingabetaste`
3. Fügen Sie den folgenden Beispielcode in den Editor ein
4. Speichern Sie die Datei als `ConsoleApp.cs` . Gehen Sie dazu auf **Datei** → **Speichern unter ...** , geben Sie `ConsoleApp.cs` in das `ConsoleApp.cs` 'Dateiname' ein und wählen Sie als Dateityp `All Files` .
5. Klicken Sie auf `Save`

Quellcode kompilieren

1. Öffnen Sie das Dialogfeld Ausführen mit `Windows-Taste` + `R`
2. Geben Sie ein:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Gehen Sie jetzt zu dem Ort zurück, an dem Sie Ihre `ConsoleApp.cs` Datei ursprünglich gespeichert `ConsoleApp.cs` . Sie sollten jetzt eine ausführbare Datei (`ConsoleApp.exe`) sehen. Doppelklicken Sie auf `ConsoleApp.exe` , um es zu öffnen.

Das ist es! Ihre Konsolenanwendung wurde kompiliert. Eine ausführbare Datei wurde erstellt und Sie haben jetzt eine funktionierende Konsolen-App.

```
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;
                }
            }
        }
    }
}
```

Konsolenanwendung mit einem Nur-Text-Editor und dem C # -Compiler erstellen (csc.exe) online lesen: https://riptutorial.com/de/csharp/topic/6676/konsolenanwendung-mit-einem-nur-text-editor-und-dem-c-sharp--compiler-erstellen--csc-exe-

Kapitel 86: Konstruktoren und Finalisierer

Einführung

Konstruktoren sind Methoden in einer Klasse, die aufgerufen werden, wenn eine Instanz dieser Klasse erstellt wird. Ihre Hauptaufgabe besteht darin, das neue Objekt in einem nützlichen und konsistenten Zustand zu belassen.

Destruktoren / Finalizer sind Methoden in einer Klasse, die aufgerufen werden, wenn eine Instanz davon zerstört wird. In C # werden sie selten explizit geschrieben / verwendet.

Bemerkungen

C # hat eigentlich keine Destruktoren, sondern Finalizer, die die Destruktorsyntax im C ++ - Stil verwenden. Durch die Angabe eines Destruktors wird die `Object.Finalize()` Methode überschrieben, die nicht direkt aufgerufen werden kann.

Im Gegensatz zu anderen Sprachen mit ähnlicher Syntax werden diese Methoden *nicht* aufgerufen, wenn Objekte den Gültigkeitsbereich verlassen, sondern beim Ausführen des Garbage Collector, was unter bestimmten Bedingungen auftritt. Daher können sie *nicht* in einer bestimmten Reihenfolge ausgeführt werden.

Finalizers sollte **nur** zur Reinigung von nicht verwalteten Ressourcen verantwortlich sein (Zeiger über die Marshal - Klasse erworben, erhalten durch p / Invoke (Systemaufrufe) oder rohe Zeiger innerhalb unsichere Blöcke verwendet). Um bereinigte Ressourcen zu bereinigen, überprüfen Sie bitte IDisposable, das Dispose-Muster und die `using` Anweisung.

(Weiterführende Literatur: Wann sollte ich einen Destruktor erstellen?)

Examples

Standardkonstruktor

Wenn ein Typ ohne Konstruktor definiert wird:

```
public class Animal
{
}
```

dann generiert der Compiler einen Standardkonstruktor, der dem folgenden entspricht:

```
public class Animal
{
    public Animal() {}
}
```

Die Definition eines beliebigen Konstruktors für den Typ unterdrückt die Erstellung des Standardkonstruktors. Wenn der Typ wie folgt definiert wurde:

```
public class Animal
{
    public Animal(string name) {}
}
```

dann kann ein `Animal` nur durch Aufrufen des deklarierten Konstruktors erstellt werden.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

Für das zweite Beispiel zeigt der Compiler eine Fehlermeldung an:

> 'Animal' enthält keinen Konstruktor, der 0 Argumente akzeptiert

Wenn eine Klasse sowohl einen parameterlosen Konstruktor als auch einen Konstruktor haben soll, der einen Parameter übernimmt, können Sie dies tun, indem Sie beide Konstruktoren explizit implementieren.

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

Der Compiler kann keinen Standardkonstruktor generieren, wenn die Klasse eine andere Klasse erweitert, die keinen parameterlosen Konstruktor besitzt. Wenn wir zum Beispiel eine Klasse `Creature` :

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

Dann wird `Animal` als `class Animal : Creature {}` nicht kompiliert.

Aufruf eines Konstruktors aus einem anderen Konstruktor

```
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.
```

Statischer Konstruktor

Ein statischer Konstruktor wird aufgerufen, wenn zum ersten Mal ein Member eines Typs initialisiert wird, ein statischer Klassenmitglied aufgerufen wird oder eine statische Methode. Der statische Konstruktor ist threadsicher. Ein statischer Konstruktor wird normalerweise verwendet, um:

- Initialisieren Sie den statischen Status, dh den Status, der von verschiedenen Instanzen derselben Klasse gemeinsam genutzt wird.
- Erstellen Sie ein Singleton

Beispiel:

```
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();
```

Ausgabe:

Tier initialisiert
Tier erstellt
Tier erstellt

Demo anzeigen

Wenn der erste Aufruf eine statische Methode ist, wird der statische Konstruktor ohne den Instanzkonstruktor aufgerufen. Dies ist in Ordnung, da die statische Methode sowieso nicht auf den Instanzstatus zugreifen kann.

```
Animal.Yawn();
```

Dies wird ausgegeben:

> Tier initialisiert
> Gähnen!

Siehe auch Ausnahmen in statischen Konstruktoren und generischen statischen Konstruktoren .

Singleton-Beispiel:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Aufruf des Basisklassenkonstruktors

Ein Konstruktor einer Basisklasse wird aufgerufen, bevor ein Konstruktor einer abgeleiteten Klasse ausgeführt wird. Wenn beispielsweise `Mammal Animal` , wird der im Konstruktor von `Animal` enthaltene Code beim Erstellen einer Instanz eines `Mammal` zuerst aufgerufen.

Wenn eine abgeleitete Klasse nicht explizit angibt, welcher Konstruktor der Basisklasse aufgerufen werden soll, nimmt der Compiler den parameterlosen Konstruktor an.

```
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.");
    }
}
```

In diesem Fall wird das Instantiieren eines `Mammal` durch Aufrufen des `new Mammal("George the Cat")` gedruckt

> Ein unbekanntes Tier wird geboren.

George the Cat ist ein Säugetier.

Demo anzeigen

Das Aufrufen eines anderen Konstruktors der Basisklasse erfolgt durch Platzieren von `: base(args)` zwischen der Signatur des Konstruktors und seinem Rumpf:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Das Aufrufen eines `new Mammal("George the Cat")` wird jetzt gedruckt:

George the Cat wird geboren.
George the Cat ist ein Säugetier.

Demo anzeigen

Finalisierer für abgeleitete Klassen

Wenn ein Objektdiagramm abgeschlossen ist, ist die Reihenfolge in umgekehrter Reihenfolge der Konstruktion. Der Supertyp wird beispielsweise vor dem Basistyp abgeschlossen, wie der folgende Code veranschaulicht:

```
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!
```

Singleton-Konstruktormuster

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

Da der Konstruktor privat ist, können keine neuen Instanzen von `SingletonClass` erstellt werden, indem Code verwendet wird. Die einzige Möglichkeit, auf die einzelne Instanz von `SingletonClass` zuzugreifen, ist die statische Eigenschaft `SingletonClass.Instance` .

Die `Instance` wird von einem statischen Konstruktor zugewiesen, den der C # -Compiler generiert. Die .NET-Laufzeitumgebung gewährleistet, dass der statische Konstruktor höchstens einmal ausgeführt wird, bevor die `Instance` zuerst gelesen wird. Daher werden alle Synchronisierungs- und Initialisierungsprobleme von der Laufzeitumgebung ausgeführt.

Beachten Sie, dass die `Singleton` Klasse bei einem Ausfall des statischen Konstruktors für die gesamte Lebensdauer der AppDomain dauerhaft unbrauchbar wird.

Es kann auch nicht garantiert werden, dass der statische Konstruktor beim ersten Zugriff von `Instance` . Es wird vielmehr *irgendwann davor* laufen. Dies macht den Zeitpunkt, zu dem die Initialisierung stattfindet, nicht deterministisch. In praktischen Fällen ruft das JIT häufig den statischen Konstruktor während der *Kompilierung* (nicht der Ausführung) einer auf die `Instance` referenzierenden Methode auf. Dies ist eine Leistungsoptimierung.

Auf der Singleton-Implementierungsseite finden Sie weitere Möglichkeiten zum Implementieren des Singleton-Musters.

Erzwingen des Aufrufs eines statischen Konstruktors

Während statische Konstruktoren immer vor der ersten Verwendung eines Typs aufgerufen werden, kann es manchmal nützlich sein, sie zum Aufruf zu zwingen, und die `RuntimeHelpers` Klasse stellt einen Helfer dafür bereit:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

***Anmerkung* : Es wird die** gesamte statische Initialisierung (beispielsweise Feldinitialisierer) ausgeführt, nicht nur der Konstruktor selbst.

***Mögliche Verwendungen* :** Erzwingen der Initialisierung während des Begrüßungsbildschirms in einer UI-Anwendung oder Sicherstellen, dass ein statischer Konstruktor bei einem Komponententest nicht fehlschlägt.

Virtuelle Methoden im Konstruktor aufrufen

Im Gegensatz zu C ++ in C # können Sie eine virtuelle Methode vom Klassenkonstruktor aus aufrufen (OK, Sie können dies auch in C ++ tun, aber das Verhalten ist zunächst überraschend). Zum Beispiel:

```
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
```

Wenn Sie aus einem C ++ - Hintergrund stammen, ist dies überraschend. Der Konstruktor der Basisklasse sieht bereits eine virtuelle Methodentabelle der abgeleiteten Klasse!

Seien Sie vorsichtig : Die abgeleitete Klasse wurde möglicherweise noch nicht vollständig initialisiert (ihr Konstruktor wird nach dem Konstruktor der Basisklasse ausgeführt), und diese Technik ist gefährlich (es gibt auch eine StyleCop-Warnung). Normalerweise wird dies als schlechte Praxis angesehen.

Generische statische Konstruktoren

Wenn der Typ, für den der statische Konstruktor deklariert ist, generisch ist, wird der statische Konstruktor einmal für jede eindeutige Kombination generischer Argumente aufgerufen.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
```

```
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

Dies wird ausgegeben:

> System.Object
> System.String

Siehe auch Wie funktionieren statische Konstruktoren für generische Typen?

Ausnahmen bei statischen Konstruktoren

Wenn ein statischer Konstruktor eine Ausnahme auslöst, wird er nie wiederholt. Der Typ kann für die Lebensdauer der AppDomain nicht verwendet werden. Bei weiteren Verwendungen des Typs wird eine `TypeInitializationException` die die ursprüngliche Ausnahme `TypeInitializationException` .

```
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());
}
```

Dies wird ausgegeben:

> Statischer ctor
>
> System.TypeInitializationException: Der Typinitialisierer für 'Animal' hat eine Ausnahme ausgelöst. ---> System.Exception: Ausnahme des Typs 'System.Exception' wurde ausgelöst.

[...]

> System.TypeInitializationException: Der Typinitialisierer für 'Animal' hat eine Ausnahme ausgelöst. ---> System.Exception: Ausnahme des Typs 'System.Exception' wurde ausgelöst.

Dort sehen Sie, dass der eigentliche Konstruktor nur einmal ausgeführt wird und die Ausnahme erneut verwendet wird.

Konstruktor- und Eigenschaftsinitialisierung

Soll die Eigenschaftswertzuweisung *vor* oder *nach* dem Konstruktor der Klasse ausgeführt werden?

```
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 };
```

`TestProperty` der `TestProperty` Wert im `TestProperty` Beispiel `1` im Konstruktor der Klasse oder nach dem Klassenkonstruktor sein?

Zuweisen von Eigenschaftswerten bei der Instanzerstellung wie folgt:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Wird ausgeführt, ***nachdem*** der Konstruktor ausgeführt wurde. Initialisieren Sie den Eigenschaftswert in der Klasseneigenschaft in C # 6.0 jedoch wie folgt:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

wird ausgeführt, ***bevor*** der Konstruktor ausgeführt wird.

Die beiden obigen Konzepte in einem einzigen Beispiel kombinieren:

```
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
}
```

Endergebnis:

```
"Or shall this be executed"
"1"
```

Erläuterung:

Der `TestProperty` Wert wird zuerst als `2` zugewiesen. Anschließend wird der `TestClass` Konstruktor ausgeführt, was zum Ausdruck von führt

```
"Or shall this be executed"
```

Und dann wird der `TestProperty` wird als zugeordnet werden `1` aufgrund `new TestClass() { TestProperty = 1 }` , für die die endgültige Wert machen `TestProperty` gedruckt von `Console.WriteLine(testInstance.TestProperty)` zu sein ,

```
"1"
```

Konstruktoren und Finalisierer online lesen:
https://riptutorial.com/de/csharp/topic/25/konstruktoren-und-finalisierer

Kapitel 87: Kreationelle Designmuster

Bemerkungen

Die Schöpfungsmuster zielen darauf ab, ein System von der Erstellung, Zusammenstellung und Darstellung seiner Objekte zu trennen. Sie erhöhen die Flexibilität des Systems hinsichtlich des Was, Wer, Wie und Wann der Objekterstellung. Kreationsmuster kapseln das Wissen darüber, welche Klassen ein System verwendet, verdecken jedoch die Details, wie die Instanzen dieser Klassen erstellt und zusammengefügt werden. Programmierer haben erkannt, dass das Erstellen von Systemen mit Vererbung diese Systeme zu starr macht. Die kreativen Muster sollen diese enge Kopplung aufheben.

Examples

Singleton-Muster

Das Singleton-Muster dient dazu, die Erstellung einer Klasse auf genau eine einzige Instanz zu beschränken.

Dieses Muster wird in einem Szenario verwendet, in dem es sinnvoll ist, nur eines davon zu haben, z.

- eine einzige Klasse, die die Interaktionen anderer Objekte orchestriert, z. Manager-Klasse
- oder eine Klasse, die eine eindeutige, einzige Ressource darstellt, z. Protokollierungskomponente

Eine der häufigsten Methoden zum Implementieren des Singleton-Musters ist eine statische **Factory-Methode** wie `CreateInstance()` oder `GetInstance()` (oder eine statische Eigenschaft in C #, `Instance`), die darauf ausgelegt ist, immer dieselbe Instanz zurückzugeben.

Beim ersten Aufruf der Methode oder Eigenschaft wird die Singleton-Instanz erstellt und zurückgegeben. Danach gibt die Methode immer dieselbe Instanz zurück. Auf diese Weise gibt es immer nur eine Instanz des Singleton-Objekts.

Das Erstellen von Instanzen über `new` kann verhindert werden, indem die Klassenkonstruktoren als `private`.

Hier ein typisches Codebeispiel zum Implementieren eines Singleton-Musters in C #:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
```

```
    // against unintentional misuse.
    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

Um dieses Muster weiter zu veranschaulichen, prüft der folgende Code, ob eine identische Instanz des Singleton zurückgegeben wird, wenn die Instance-Eigenschaft mehrmals aufgerufen wird.

```
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");
        }
    }
}
```

Hinweis: Diese Implementierung ist nicht threadsicher.

Weitere Beispiele, wie dieser Thread sicher gemacht werden kann, finden Sie unter Singleton-Implementierung

Singletons sind konzeptionell einem globalen Wert ähnlich und verursachen ähnliche Designfehler und Probleme. Aus diesem Grund wird das Singleton-Muster allgemein als ein Anti-Muster angesehen.

Besuchen Sie "Was ist so schlimm an Singletons?" Weitere Informationen zu den Problemen, die bei ihrer Verwendung auftreten.

In C # haben Sie die Möglichkeit, eine Klasse `static` zu machen, wodurch alle Mitglieder statisch werden und die Klasse nicht instanziiert werden kann. Aus diesem Grund ist es üblich, statische Klassen anstelle des Singleton-Musters zu sehen.

Die wichtigsten Unterschiede zwischen den beiden finden Sie unter C # Singleton Pattern versus Static Class .

Fabrikmethode Muster

Factory Method ist eines der kreativen Designmuster. Es wird verwendet, um das Problem der Erstellung von Objekten zu lösen, ohne den genauen Ergebnistyp anzugeben. In diesem Dokument erfahren Sie, wie Sie Factory Method DP richtig verwenden.

Lassen Sie mich Ihnen die Idee an einem einfachen Beispiel erklären. Stellen Sie sich vor, Sie arbeiten in einer Fabrik, in der drei Arten von Geräten hergestellt werden: Amperemeter, Voltmeter und Widerstandsmesser. Sie schreiben ein Programm für einen zentralen Computer, das ein ausgewähltes Gerät erstellt, aber Sie wissen nicht die endgültige Entscheidung Ihres Chefs darüber, was er produzieren soll.

Erstellen wir ein Schnittstellen- `IDevice` mit einigen allgemeinen Funktionen, die alle Geräte haben:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Jetzt können wir Klassen erstellen, die unsere Geräte darstellen. Diese Klassen müssen die `IDevice` Schnittstelle implementieren:

```
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..."); }
}
```

Jetzt müssen wir die Factory-Methode definieren. Erstellen wir die `DeviceFactory` Klasse mit einer statischen Methode:

```
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();
        }
    }
}
```

Großartig! Lassen Sie uns unseren Code testen:

```
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();
    }
}
```

Dies ist die Beispielausgabe, die nach dem Ausführen dieses Codes angezeigt wird:

AmMeter schaltet sich ein ...

36

6

33

43

24

AmMeter blinkt Lichter und verabschiedet sich!

VoltMeter schaltet sich ein ...

102

-61

85

138

36

VoltMeter blinkt Lichter zum Abschied!

OhmMeter schaltet sich ein ...

723828

368536

685412

800266

578595

OhmMeter blinkt Lichter zum Abschied!

Generator-Muster

Trennen Sie die Konstruktion eines komplexen Objekts von seiner Repräsentation, sodass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erstellen kann und ein hohes Maß an Kontrolle über den Zusammenbau der Objekte bietet.

In diesem Beispiel wird das Builder-Muster veranschaulicht, in dem verschiedene Fahrzeuge Schritt für Schritt zusammengebaut werden. Der Shop verwendet VehicleBuilders, um verschiedene Fahrzeuge in einer Reihe von aufeinanderfolgenden Schritten zu konstruieren.

```
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"]);
    }
  }
}
```

Ausgabe

Fahrzeugtyp: Scooter Frame: Scooter Frame
Motor: keiner
#Räder: 2
#Türen: 0

Fahrzeugtyp: Auto
Rahmen: Auto Frame
Motor: 2500 ccm
#Räder: 4
#Türen: 4

Fahrzeugtyp: Motorrad
Frame: MotorCycle Frame
Motor: 500 ccm

#Räder: 2
#Türen: 0

Prototypmuster

Geben Sie die Art der Objekte an, die mithilfe einer prototypischen Instanz erstellt werden sollen, und erstellen Sie neue Objekte, indem Sie diesen Prototyp kopieren.

In diesem Beispiel wird das Prototypmuster veranschaulicht, in dem neue Color-Objekte erstellt werden, indem bereits vorhandene benutzerdefinierte Farben desselben Typs kopiert werden.

```
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); }
    }
  }
}
```

Ausgabe:

Klonfarbe RGB: 255, 0, 0

Klonierungsfarbe RGB: 128,211,128

Klonierungsfarbe RGB: 211, 34, 20

Abstraktes Fabrikmuster

Bieten Sie eine Schnittstelle zum Erstellen von Familien verwandter oder abhängiger Objekte, ohne deren konkrete Klassen anzugeben.

In diesem Beispiel wird die Erstellung verschiedener Tierwelten für ein Computerspiel anhand

verschiedener Fabriken veranschaulicht. Obwohl die von den Kontinentfabriken geschaffenen Tiere unterschiedlich sind, bleiben die Interaktionen zwischen den Tieren gleich.

```
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);
    }
  }
}
```

Ausgabe:

> Löwe isst Gnus
>
> Wolf isst Bison

Kreationelle Designmuster online lesen: https://riptutorial.com/de/csharp/topic/6654/kreationelle-designmuster

Kapitel 88: Kryptographie (System.Security.Cryptography)

Examples

Moderne Beispiele für die symmetrische authentifizierte Verschlüsselung einer Zeichenfolge

Kryptographie ist etwas sehr Schwieriges, und nachdem ich viel Zeit mit dem Lesen verschiedener Beispiele verbracht und gesehen hatte, wie einfach es ist, eine Form von Schwachstellen einzuführen, fand ich eine Antwort, die ursprünglich von @jbtule geschrieben wurde und die meiner Meinung nach sehr gut ist. Viel Spaß beim Lesen:

Die generelle bewährte Methode für die symmetrische Verschlüsselung ist die Verwendung von Authenticated Encryption mit assoziierten Daten (AEAD). Dies ist jedoch nicht Teil der Standard-.net- Kryptobibliotheken . Das erste Beispiel verwendet also AES256 und dann HMAC256 , dann zwei Schritte für Encrypt MAC , was mehr Aufwand und mehr Schlüssel erfordert.

Das zweite Beispiel verwendet die einfachere Praxis von AES256- GCM unter Verwendung des Open Source Bouncy Castle (via Nuget).

Beide Beispiele haben eine Hauptfunktion, die eine geheime Meldungszeichenfolge, Schlüssel und eine optionale nicht-geheime Nutzlast sowie eine zurückgegebene und authentifizierte verschlüsselte Zeichenfolge verwendet, die optional mit den nicht-geheimen Daten versehen ist. Idealerweise würden Sie diese mit zufällig generierten `NewKey()` Bit-Schlüsseln verwenden, siehe `NewKey()` .

Beide Beispiele verfügen auch über Hilfsmethoden, die zum Generieren der Schlüssel ein String-Kennwort verwenden. Diese Hilfsmethoden werden als Annehmlichkeit für andere Beispiele bereitgestellt. Sie sind jedoch *weitaus weniger sicher,* da die Stärke des Kennworts *weitaus schwächer sein wird als bei einem 256-Bit-Schlüssel* .

Update: `byte[]` -Überladungen wurden hinzugefügt, und nur die Gist hat die vollständige Formatierung mit 4 Leerzeichen-Einzug und Api-Docs aufgrund der StackOverflow-Antwortgrenzen. "

Eingebaute .NET-Verschlüsselung (AES) - Dann-MAC (HMAC) [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
```

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

namespace Encryption
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
HMac-Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
```

```
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
                        int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
```

```
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }
```

```
      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!",
KeyBitSize), "cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];
```

```
        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
              binaryWriter.Write(
                encryptedMessage,
                nonSecretPayloadLength + iv.Length,
                encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
              );
            }
            //Return Plain Text
            return plainTextStream.ToArray();
          }
        }
      }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length ==0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      int payloadIndex = nonSecretPayload.Length;

      byte[] cryptKey;
      byte[] authKey;
      //Use Random Salt to prevent pre-generated weak password attacks.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
```

```
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);

        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
      }

      //Deriving separate key, might be less efficient than using HKDF,
      //but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      }

      return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
```

```
authSalt.Length + nonSecretPayloadLength);
    }
  }
}
```

Bouncy Castle AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
```

```
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33
Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
```

```
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
```

```
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
```

```
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +
nonSecretPayloadLength);
```

```
    }
  }
}
```

Einführung in die symmetrische und asymmetrische Verschlüsselung

Sie können die Sicherheit für die Datenübertragung oder -speicherung verbessern, indem Sie Verschlüsselungstechniken implementieren. Grundsätzlich gibt es bei der Verwendung von *System.Security.Cryptography* zwei Ansätze: **symmetrisch** und **asymmetrisch.**

Symmetrische Verschlüsselung

Diese Methode verwendet einen privaten Schlüssel, um die Datenumwandlung durchzuführen.

Pros:

- Symmetrische Algorithmen verbrauchen weniger Ressourcen und sind schneller als asymmetrische.
- Die Datenmenge, die Sie verschlüsseln können, ist unbegrenzt.

Nachteile:

- Verschlüsselung und Entschlüsselung verwenden denselben Schlüssel. Jemand kann Ihre Daten entschlüsseln, wenn der Schlüssel gefährdet ist.
- Wenn Sie einen anderen geheimen Schlüssel für andere Daten verwenden, können Sie viele verschiedene geheime Schlüssel verwalten.

Unter System.Security.Cryptography gibt es verschiedene Klassen, die eine symmetrische Verschlüsselung durchführen. Sie werden als Blockchiffren bezeichnet :

- AesManaged (AES- Algorithmus).
- AesCryptoServiceProvider (AES- Algorithmus, FIPS 140-2-Beschwerde).
- DESCryptoServiceProvider (DES- Algorithmus).
- RC2CryptoServiceProvider (Rivest Cipher 2- Algorithmus).
- RijndaelManaged (AES- Algorithmus). *Hinweis* : RijndaelManaged ist **keine** FIPS-197-Beschwerde.
- TripleDES (TripleDES- Algorithmus).

Asymmetrische Verschlüsselung

Diese Methode verwendet eine Kombination aus öffentlichen und privaten Schlüsseln, um die Datentransformation durchzuführen.

Pros:

- Es verwendet größere Schlüssel als symmetrische Algorithmen, daher sind sie weniger

anfällig für Rissbildung durch rohe Gewalt.
- Es ist einfacher zu gewährleisten, wer die Daten ver- und entschlüsseln kann, da zwei Schlüssel (öffentlich und privat) verwendet werden.

Nachteile:

- Die Datenmenge, die Sie verschlüsseln können, ist begrenzt. Die Grenze ist für jeden Algorithmus unterschiedlich und ist normalerweise proportional zur Schlüsselgröße des Algorithmus. Ein RSACryptoServiceProvider-Objekt mit einer Schlüssellänge von 1.024 Bits kann beispielsweise nur eine Nachricht verschlüsseln, die kleiner als 128 Byte ist.
- Asymmetrische Algorithmen sind im Vergleich zu symmetrischen Algorithmen sehr langsam.

Unter System.Security.Cryptography haben Sie Zugriff auf verschiedene Klassen, die eine asymmetrische Verschlüsselung durchführen:

- DSACryptoServiceProvider (Algorithmus für digitale Signaturalgorithmen)
- RSACryptoServiceProvider (RSA-Algorithmus- Algorithmus)

Passwort-Hashing

Passwörter sollten niemals als Klartext gespeichert werden! Sie sollten mit einem zufällig generierten Salt (zur Abwehr von Regenbogentischangriffen) mit einem langsamen Passwort-Hash-Algorithmus gehasht werden. Eine große Anzahl von Iterationen (> 10.000) kann verwendet werden, um Brute-Force-Angriffe zu verlangsamen. Eine Verzögerung von ~ 100ms ist für einen Benutzer akzeptabel, macht es jedoch schwierig, ein langes Passwort zu brechen. Bei der Auswahl einer Reihe von Iterationen sollten Sie den maximal zulässigen Wert für Ihre Anwendung verwenden und diesen Wert erhöhen, wenn sich die Computerleistung verbessert. Sie müssen auch in Erwägung ziehen, wiederholte Anfragen zu stoppen, die als DoS-Angriff verwendet werden könnten.

Wenn beim ersten Hashing ein Salt für Sie generiert werden kann, können der resultierende Hash und Salt in einer Datei gespeichert werden.

```
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
}
```

Überprüfen Sie ein bestehendes Benutzerpasswort, lesen Sie dessen Hash und Salt aus einer Datei und vergleichen Sie es mit dem Hash des eingegebenen Passworts

```
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;
}
```

Einfache symmetrische Dateiverschlüsselung

Das folgende Codebeispiel veranschaulicht ein schnelles und einfaches Mittel zum Ver- und Entschlüsseln von Dateien mithilfe des symmetrischen AES-Verschlüsselungsalgorithmus.

Der Code generiert bei jeder Verschlüsselung einer Datei zufällig die Salt- und Initialisierungsvektoren. Dies bedeutet, dass die Verschlüsselung derselben Datei mit demselben Kennwort immer zu einer anderen Ausgabe führt. Der Salt und IV werden in die Ausgabedatei geschrieben, so dass nur das Kennwort zum Entschlüsseln erforderlich ist.

```
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);
```

```
        }
    }
}
```

Kryptografisch sichere Zufallsdaten

Es kann vorkommen, dass die Random () - Klasse des Frameworks nicht als zufällig angesehen wird, da sie auf einem Pseudo-Zufallszahlengenerator basiert. Die Crypto-Klassen des Frameworks bieten jedoch mit RNGCryptoServiceProvider etwas Robusteres.

In den folgenden Codebeispielen wird veranschaulicht, wie Arrays, Strings und Zahlen für kryptografisch sichere Bytes generiert werden.

Zufälliges Byte-Array

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

Zufällige ganze Zahl (mit gerader Verteilung)

```
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);
}
```

Zufällige Zeichenfolge

```
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);
}
```

Schnelle asymmetrische Dateiverschlüsselung

Für die Übertragung von Nachrichten an andere Parteien wird die asymmetrische Verschlüsselung häufig der Symmetrischen Verschlüsselung vorgezogen. Dies ist vor allem darauf zurückzuführen, dass viele der mit dem Austausch eines gemeinsam genutzten Schlüssels verbundenen Risiken aufgehoben werden und sichergestellt wird, dass jeder Benutzer mit dem öffentlichen Schlüssel eine Nachricht für den beabsichtigten Empfänger verschlüsseln kann, aber nur dieser Empfänger diese entschlüsseln kann. Leider ist der größte Nachteil asymmetrischer Verschlüsselungsalgorithmen, dass sie wesentlich langsamer sind als ihre symmetrischen Verwandten. Daher kann die asymmetrische Verschlüsselung von Dateien, insbesondere großen Dateien, oftmals sehr rechenintensiv sein.

Um sowohl Sicherheit als auch Leistung zu bieten, kann ein hybrider Ansatz gewählt werden. Dies beinhaltet die kryptografisch zufällige Erzeugung eines Schlüssels und eines Initialisierungsvektors für die *symmetrische* Verschlüsselung. Diese Werte werden dann mit einem *asymmetrischen* Algorithmus verschlüsselt und in die Ausgabedatei geschrieben, bevor die Quelldaten *symmetrisch* verschlüsselt und an die Ausgabe angehängt werden.

Dieser Ansatz bietet ein hohes Maß an Leistung und Sicherheit, da die Daten mit einem symmetrischen Algorithmus (schnell) verschlüsselt werden und der Schlüssel und iv, die beide zufällig generiert (sicher), durch einen asymmetrischen Algorithmus (sicher) verschlüsselt werden. Es hat auch den zusätzlichen Vorteil, dass die gleiche Nutzlast, die zu verschiedenen Zeitpunkten verschlüsselt wird, einen sehr unterschiedlichen Cyphertext hat, da die symmetrischen Schlüssel jedes Mal zufällig generiert werden.

Die folgende Klasse demonstriert die asymmetrische Verschlüsselung von Zeichenfolgen und Byte-Arrays sowie die Hybrid-Dateiverschlüsselung.

```
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
}
```

Anwendungsbeispiel:

```
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");
}
```

Kryptographie (System.Security.Cryptography) online lesen:
https://riptutorial.com/de/csharp/topic/2988/kryptographie--system-security-cryptography-

Kapitel 89: Lambda-Ausdrücke

Bemerkungen

Ein Lambda-Ausdruck ist eine Syntax zum Erstellen anonymer Funktionen inline. Formaler aus dem C # -Programmierhandbuch :

> Ein Lambda-Ausdruck ist eine anonyme Funktion, die Sie zum Erstellen von Delegaten oder Ausdrucksbaumtypen verwenden können. Mithilfe von Lambda-Ausdrücken können Sie lokale Funktionen schreiben, die als Argumente übergeben oder als Wert von Funktionsaufrufen zurückgegeben werden können.

Ein Lambda-Ausdruck wird mit dem Operator `=>` . Setzen Sie die Parameter auf der linken Seite des Operators. Fügen Sie auf der rechten Seite einen Ausdruck ein, der diese Parameter verwenden kann. Dieser Ausdruck wird als Rückgabewert der Funktion aufgelöst. Seltener kann bei Bedarf ein ganzer `{code block}` auf der rechten Seite verwendet werden. Wenn der Rückgabetyp nicht ungültig ist, enthält der Block eine Rückgabeanweisung.

Examples

Übergeben eines Lambda-Ausdrucks als Parameter an eine Methode

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Hier ist `x => x > 6` ein Lambda-Ausdruck, der als Prädikat fungiert und dafür sorgt, dass nur Elemente über 6 zurückgegeben werden.

Lambda-Ausdrücke als Abkürzung für die Initialisierung von Delegaten

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

Die obige Lambda-Ausdruckssyntax entspricht dem folgenden ausführlichen Code:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

Lambdas sowohl für "Func" als auch für "Action"

Typischerweise werden Lambdas zur Definition einfacher *Funktionen verwendet* (im Allgemeinen im Kontext eines linq-Ausdrucks)

```
var incremented = myEnumerable.Select(x => x + 1);
```

Hier ist die `return` implizit.

Es ist jedoch auch möglich, *Aktionen* als Lambdas zu übergeben:

```
myObservable.Do(x => Console.WriteLine(x));
```

Lambda-Ausdrücke mit mehreren Parametern oder ohne Parameter

Verwenden Sie in Klammern den Ausdruck links vom Operator `=>` , um mehrere Parameter anzugeben.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

In ähnlicher Weise zeigt ein leerer Satz von Klammern an, dass die Funktion keine Parameter akzeptiert.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

Mehrere Anweisungen in eine Anweisung einfügen Lambda

Im Gegensatz zu einem Ausdruck Lambda kann ein Anweisungs-Lambda mehrere durch Semikola getrennte Anweisungen enthalten.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Beachten Sie, dass die Anweisungen in geschweiften Klammern `{}` .

Beachten Sie, dass Anweisungslambas nicht zum Erstellen von Ausdrucksbäumen verwendet werden können.

Lambdas können sowohl als "Func" als auch als "Expression" ausgegeben werden

Angenommen, die folgende `Person` :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Das folgende Lambda:

```
p => p.Age > 18
```

Kann als Argument an beide Methoden übergeben werden:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Weil der Compiler Lambdas sowohl in Delegierte als auch in `Expression` s umwandeln kann.

Offensichtlich verlassen sich LINQ-Provider stark auf `Expression` s (hauptsächlich über die `IQueryable<T>` -Schnittstelle), um Abfragen analysieren und in Abfragen speichern zu können.

Lambda-Ausdruck als Ereignishandler

Mit Lambda-Ausdrücken können Ereignisse behandelt werden. Dies ist nützlich, wenn:

- Der Handler ist kurz.
- Der Handler muss niemals abgemeldet werden.

Eine gute Situation, in der ein Lambda-Event-Handler verwendet werden kann, ist unten angegeben:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

Wenn das Abbestellen eines registrierten Ereignishandlers zu einem späteren Zeitpunkt im Code erforderlich ist, muss der Ausdruck des Ereignishandlers in einer Variablen gespeichert werden, und die Registrierung / Aufhebung der Registrierung erfolgt über diese Variable:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

Der Grund dafür ist, dass Sie den Lambda-Ausdruck nicht einfach wörtlich erneut eingeben (`-=`), weil der C # -Compiler die beiden Ausdrücke nicht unbedingt als gleich betrachtet:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Wenn dem Lambda-Ausdruck zusätzliche Anweisungen hinzugefügt werden, können versehentlich die erforderlichen umgebenden geschweiften Klammern weggelassen werden, ohne einen Kompilierungsfehler zu verursachen. Zum Beispiel:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

Dies wird kompiliert, führt jedoch zum Hinzufügen des Lambda-Ausdrucks `(sender, args) => Console.WriteLine("Email sent");` als Event-Handler und Ausführen der Anweisung `emailSendButton.Enabled = true;` sofort. Um dies zu beheben, muss der Inhalt des Lambda von geschweiften Klammern umgeben sein. Dies kann vermieden werden, indem von Anfang an geschweifte Klammern verwendet werden, wenn Sie zusätzliche Anweisungen zu einem Lambda-Event-Handler hinzufügen oder das Lambda von Anfang an in runde Klammern einschließen:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Lambda-Ausdrücke online lesen: https://riptutorial.com/de/csharp/topic/46/lambda-ausdrucke

Kapitel 90: Lambda-Ausdrücke

Bemerkungen

Verschlüsse

Lambda-Ausdrücke erfassen implizit die verwendeten Variablen und erzeugen eine Schließung . Ein Abschluss ist eine Funktion zusammen mit einem Zustandskontext. Der Compiler generiert einen Abschluss, wenn ein Lambda-Ausdruck einen Wert aus dem umgebenden Kontext "umschließt".

ZB wenn das Folgende ausgeführt wird

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

`safeApplyFilterPredicate` bezieht sich auf ein neu erstelltes Objekt, das einen privaten Verweis auf den aktuellen Wert des `filterer` hat und dessen `Invoke` Methode sich so verhält

```
o => (o != null) && filterer.Predicate(i);
```

Dies kann wichtig sein, denn solange der Verweis auf den Wert in `safeApplyFilterPredicate` beibehalten wird, gibt es einen Verweis auf das Objekt, auf das sich der `filterer` aktuell bezieht. Dies hat Auswirkungen auf die Garbage Collection und kann zu einem unerwarteten Verhalten führen, wenn das Objekt, auf das der `filterer` aktuell verweist, mutiert ist.

Auf der anderen Seite können Verschlüsse verwendet werden, um absichtlich einen Effekt abzukapseln, um ein Verhalten zu kapseln, das Verweise auf andere Objekte beinhaltet.

Z.B

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

Verschlüsse können auch zum Modellieren von Zustandsmaschinen verwendet werden:

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

Examples

Grundlegende Lambda-Ausdrücke

```
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
```

Grundlegende Lambda-Ausdrücke mit LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Verwenden der Lambda-Syntax zum Erstellen eines Abschlusses

Siehe Anmerkungen zur Erörterung von Schließungen. Angenommen, wir haben eine Schnittstelle:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

und dann wird folgendes ausgeführt:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

`machineClosure` bezieht sich jetzt auf eine Funktion von `int` bis `int` , die im Hintergrund die Instanz von `IMachine` verwendet, auf die sich die `machine` bezieht, um die Berechnung durchzuführen. Auch

wenn die Referenz `machine` außerhalb des Bereichs geht, solange das `machineClosure` Objekt beibehalten wird, das ursprüngliche `IMachine` wird beispielsweise als Teil einer ‚Schließung' zurückgehalten wird, automatisch durch den Compiler definiert.

Achtung: Dies kann bedeuten, dass der gleiche Funktionsaufruf zu unterschiedlichen Zeitpunkten unterschiedliche Werte zurückgibt (z. B. in diesem Beispiel, wenn die Maschine eine Summe ihrer Eingaben enthält). In vielen Fällen kann dies unerwartet sein und ist für jeden Code in einem funktionalen Stil zu vermeiden - versehentliche und unerwartete Schließungen können Fehler verursachen.

Lambda-Syntax mit Anweisungsblockkörper

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Lambda-Ausdrücke mit System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Lambda-Ausdrücke online lesen: https://riptutorial.com/de/csharp/topic/7057/lambda-ausdrucke

Kapitel 91: Laufzeit kompilieren

Examples

RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript` ist eine neue C # -Skript-Engine.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

Sie können beliebige Anweisungen, Variablen, Methoden, Klassen oder beliebige Codesegmente kompilieren und ausführen.

CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider` kann zum Kompilieren von C # -Klassen verwendet werden.

```
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
```

Laufzeit kompilieren online lesen: https://riptutorial.com/de/csharp/topic/3139/laufzeit-kompilieren

Kapitel 92: Linq zu Objekten

Einführung

LINQ to Objects bezieht sich auf die Verwendung von LINQ-Abfragen mit einer beliebigen IEnumerable-Auflistung.

Examples

Wie LINQ to Object Abfragen ausführt

LINQ-Abfragen werden nicht sofort ausgeführt. Wenn Sie die Abfrage erstellen, speichern Sie sie einfach für die zukünftige Ausführung. Nur wenn Sie tatsächlich die Abfrage wiederholen möchten, wird die Abfrage ausgeführt (z. B. in einer for-Schleife, beim Aufruf von ToList, Count, Max, Average, First usw.).

Dies wird als *aufgeschobene Ausführung betrachtet* . Auf diese Weise können Sie die Abfrage in mehreren Schritten aufbauen, möglicherweise basierend auf Bedingungsanweisungen ändern und sie später nur dann ausführen, wenn Sie das Ergebnis benötigen.

Gegeben der Code:

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

Im obigen Beispiel wird die Abfrage nur in der `query` gespeichert. Die Abfrage wird nicht ausgeführt.

Die `foreach` Anweisung erzwingt die Abfrageausführung:

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

Einige LINQ-Methoden lösen auch die Abfrageausführung aus: `Count` , `First` , `Max` , `Average` . Sie geben einzelne Werte zurück. `ToList` und `ToArray` sammelt das Ergebnis und `ToArray` sie in eine Liste bzw. ein Array um.

Beachten Sie, dass Sie die Abfrage mehrmals durchlaufen können, wenn Sie mehrere LINQ-Funktionen für dieselbe Abfrage aufrufen. Dies kann bei jedem Anruf zu unterschiedlichen Ergebnissen führen. Wenn Sie nur mit einem Datensatz arbeiten möchten, müssen Sie ihn in einer Liste oder einem Array speichern.

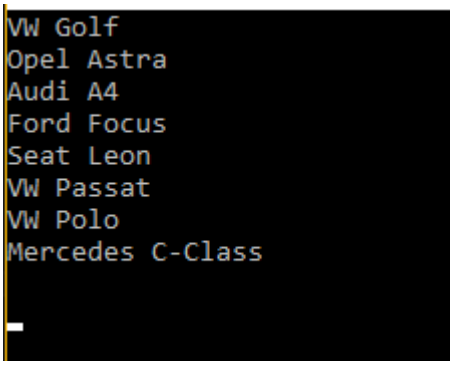
Verwenden von LINQ zu Objekten in c

Eine einfache SELECT-Abfrage in Linq

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

Im obigen Beispiel wird ein Array von Strings (Autos) als eine Sammlung von Objekten verwendet, die mit LINQ abgefragt werden sollen. In einer LINQ-Abfrage steht die from-Klausel an erster Stelle, um die Datenquelle (cars) und die Bereichsvariable (car) einzuführen. Wenn die Abfrage ausgeführt wird, dient die Bereichsvariable als Referenz für jedes nachfolgende Element in Autos. Da der Compiler auf den Fahrzeugtyp schließen kann, müssen Sie ihn nicht explizit angeben

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:

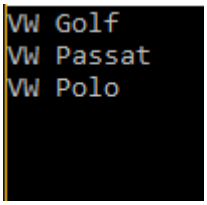


SELECT mit einer WHERE-Klausel

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

Die WHERE-Klausel wird verwendet, um das String-Array (cars) abzufragen, um eine Teilmenge des Arrays zu suchen und zurückzugeben, die die WHERE-Klausel erfüllt.

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:

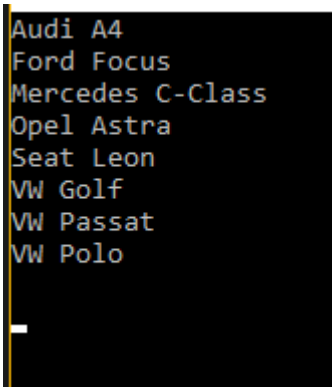


Eine geordnete Liste erstellen

```
var list = from car in cars
           orderby car ascending
           select car;
```

Manchmal ist es nützlich, die zurückgegebenen Daten zu sortieren. Die orderby-Klausel bewirkt, dass die Elemente nach dem Standardvergleicher für den zu sortierenden Typ sortiert werden.

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:



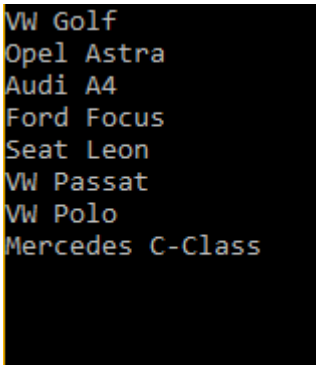
Mit einem benutzerdefinierten Typ arbeiten

In diesem Beispiel wird eine typisierte Liste erstellt, gefüllt und dann abgefragt

```
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();
    }
}
```

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:

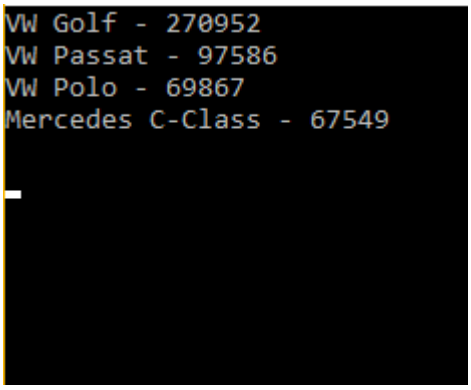


Bis jetzt scheinen die Beispiele nicht erstaunlich zu sein, da man einfach durch das Array iterieren kann, um im Grunde dasselbe zu tun. In den folgenden Beispielen können Sie jedoch sehen, wie Sie komplexere Abfragen mit LINQ to Objects erstellen und mit viel weniger Code mehr erreichen.

Im folgenden Beispiel können wir Autos auswählen, die über 60000 Einheiten verkauft wurden, und sie nach der Anzahl der verkauften Einheiten sortieren:

```
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());
```

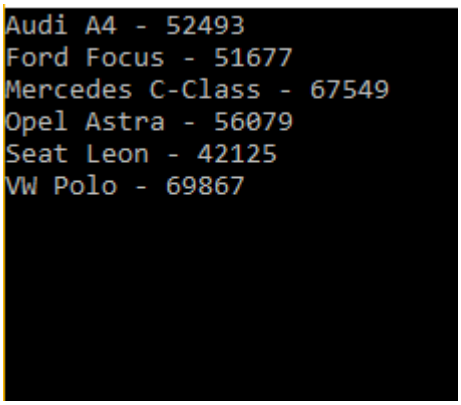
Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:



Im folgenden Beispiel können wir Autos auswählen, die eine ungerade Anzahl von Einheiten verkauft haben, und diese alphabetisch nach ihrem Namen sortieren:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:



Linq zu Objekten online lesen: https://riptutorial.com/de/csharp/topic/9405/linq-zu-objekten

Kapitel 93: LINQ zu XML

Examples

Lesen Sie XML mit LINQ to XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

So lesen Sie diese XML-Datei mit LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

Zugriff auf ein einzelnes Element

```
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);
}
```

Zugriff auf mehrere Elemente

```
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);
}
```

Zugriff auf alle Elemente mit einem bestimmten Attribut

```
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);
```

Zugriff auf ein bestimmtes Element mit einem bestimmten Attribut

```
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);
}
```

LINQ zu XML online lesen: https://riptutorial.com/de/csharp/topic/2773/linq-zu-xml

Kapitel 94: LINQ-Abfragen

Einführung

LINQ ist eine Abkürzung für **L** anguage **IN** tegrated **Q** uery. Es ist ein Konzept, das eine Abfragesprache integriert, indem ein konsistentes Modell für das Arbeiten mit Daten über verschiedene Arten von Datenquellen und -formaten hinweg bereitgestellt wird. Sie verwenden dieselben grundlegenden Codierungsmuster zum Abfragen und Umwandeln von Daten in XML-Dokumenten, SQL-Datenbanken, ADO.NET-Datensätzen, .NET-Sammlungen und allen anderen Formaten, für die ein LINQ-Anbieter verfügbar ist.

Syntax

- Abfragesyntax:
    - von <Bereichsvariable> in <Sammlung>
    - [von <Bereichsvariable> in <Sammlung>, ...]
    - <Filter, Verknüpfen, Gruppieren, Aggregatoperatoren, ...> <Lambda-Ausdruck>
    - <select oder groupBy operator> <formulieren Sie das Ergebnis>
- Methodensyntax:
    - Enumerable.Aggregate (func)
    - Enumerable.Aggregate (Seed, func)
    - Enumerable.Aggregate (Seed, func, resultSelector)
    - Enumerable.All (Prädikat)
    - Enumerable.Any ()
    - Enumerable.Any (Prädikat)
    - Enumerable.AsEnumerable ()
    - Aufzählungszeichen. Durchschnitt ()
    - Enumerable.Average (Auswahl)
    - Enumerable.Cast <Ergebnis> ()
    - Enumerable.Concat (Sekunde)
    - Enumerable.Contains (Wert)
    - Enumerable.Contains (Wert, Vergleicher)
    - Enumerable.Count ()
    - Enumerable.Count (Prädikat)
    - Enumerable.DefaultIfEmpty ()
    - Enumerable.DefaultIfEmpty (defaultValue)
    - Enumerable.Distinct ()
    - Enumerable.Distinct (Vergleicher)
    - Enumerable.ElementAt (Index)
    - Enumerable.ElementAtOrDefault (Index)
    - Enumerable.Empty ()
    - Enumerable.Except (Sekunde)

- Enumerable.Except (zweiter Vergleicher)
- Enumerable.First ()
- Enumerable.First (Prädikat)
- Enumerable.FirstOrDefault ()
- Enumerable.FirstOrDefault (Prädikat)
- Enumerable.GroupBy (keySelector)
- Enumerable.GroupBy (keySelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector)
- Enumerable.GroupBy (keySelector, Vergleicher)
- Enumerable.GroupBy (keySelector, resultSelector, vergleicher)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector, Vergleicher)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector, Vergleicher)
- Enumerable.Intersect (Sekunde)
- Enumerable.Intersect (zweiter Vergleicher)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, Vergleicher)
- Enumerable.Last ()
- Enumerable.Last (Prädikat)
- Enumerable.LastOrDefault ()
- Enumerable.LastOrDefault (Prädikat)
- Enumerable.LongCount ()
- Enumerable.LongCount (Prädikat)
- Enumerable.Max ()
- Enumerable.Max (Auswahl)
- Enumerable.Min ()
- Enumerable.Min (Auswahl)
- Enumerable.OfType <TResult> ()
- Enumerable.OrderBy (keySelector)
- Enumerable.OrderBy (keySelector, Vergleicher)
- Enumerable.OrderByDescending (keySelector)
- Enumerable.OrderByDescending (keySelector, Vergleicher)
- Enumerable.Range (Start, Anzahl)
- Enumerable.Repeat (Element, Anzahl)
- Enumerable.Reverse ()
- Enumerable.Select (Selektor)
- Enumerable.SelectMany (Selektor)
- Enumerable.SelectMany (collectionSelector, resultSelector)
- Enumerable.SequenceEqual (second)
- Enumerable.SequenceEqual (zweiter Vergleicher)
- Enumerable.Single ()
- Enumerable.Single (Prädikat)
- Enumerable.SingleOrDefault ()
- Enumerable.SingleOrDefault (Prädikat)
- Enumerable.Skip (Anzahl)

- Enumerable.SkipWhile (Prädikat)
- Enumerable.Sum ()
- Enumerable.Sum (Selektor)
- Enumerable.Take (Anzahl)
- Enumerable.TakeWhile (Prädikat)
- orderEnumerable.ThenBy (keySelector)
- orderEnumerable.ThenBy (keySelector, comparer)
- orderEnumerable.ThenByDescending (keySelector)
- orderEnumerable.ThenByDescending (keySelector, comparer)
- Enumerable.ToArray ()
- Enumerable.ToDictionary (keySelector)
- Enumerable.ToDictionary (keySelector, elementSelector)
- Enumerable.ToDictionary (keySelector, Vergleicher)
- Enumerable.ToDictionary (keySelector, elementSelector, comparer)
- Enumerable.ToList ()
- Enumerable.ToLookup (keySelector)
- Enumerable.ToLookup (keySelector, elementSelector)
- Enumerable.ToLookup (keySelector, Vergleicher)
- Enumerable.ToLookup (keySelector, elementSelector, Vergleicher)
- Enumerable.Union (Sekunde)
- Enumerable.Union (zweiter Vergleicher)
- Enumerable.Where (Prädikat)
- Enumerable.Zip (zweitens resultSelector)

Bemerkungen

Um LINQ-Abfragen verwenden zu können, müssen Sie `System.Linq` importieren.

Die Methodensyntax ist leistungsfähiger und flexibler, aber die Abfragesyntax ist möglicherweise einfacher und bekannter. Alle in der Abfragesyntax geschriebenen Abfragen werden vom Compiler in die funktionale Syntax übersetzt. Die Leistung ist also gleich.

Abfrageobjekte werden erst ausgewertet, wenn sie verwendet werden. Sie können also ohne Leistungseinbußen geändert oder hinzugefügt werden.

Examples

Woher

Gibt eine Teilmenge von Elementen zurück, für die das angegebene Prädikat wahr ist.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Methodensyntax

```
// Select all trees with name of length 3
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Abfragesyntax

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

Wählen Sie - Elemente transformieren

Mit Select können Sie eine Transformation auf jedes Element in einer Datenstruktur anwenden, die IEnumerable implementiert.

Abrufen des ersten Zeichens jeder Zeichenfolge in der folgenden Liste:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Verwenden der regulären (Lambda) Syntax

```
//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);
}
```

Ausgabe:

O
B
B
E
H
M

Live-Demo zu .NET-Geige

Verwenden der LINQ-Abfragesyntax

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

Verkettungsmethoden

Viele LINQ-Funktionen arbeiten sowohl mit einem `IEnumerable<TSource>` als auch mit einem `IEnumerable<TResult>` . Die Typparameter `TSource` und `TResult` können sich abhängig von der `TResult` Methode und den an sie übergebenen Funktionen auf denselben Typ beziehen.

Einige Beispiele dafür sind

```
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
)
```

Bei einigen Methodenverkettungen kann es vorkommen, dass vor dem Fortfahren ein ganzer Satz bearbeitet werden muss. LINQ nutzt jedoch die verzögerte Ausführung, indem er Rendite-Return- **MSDN verwendet** , wodurch ein Enumerable und ein Enumerator hinter den Kulissen erstellt werden. Beim Verketten in LINQ wird im Wesentlichen ein Aufzählungszeichen (Iterator) für den ursprünglichen Satz erstellt, der zurückgestellt wird, bis er durch Aufzählung des Aufzählungszeichens verwirklicht wird .

Dadurch können diese Funktionen fließend in einem **Wiki** verkettet werden , wobei eine Funktion direkt auf das Ergebnis einer anderen wirken kann. Mit dieser Art von Code können viele sequenzbasierte Operationen in einer einzigen Anweisung ausgeführt werden.

Beispielsweise können Sie `Select` , `Where` und `OrderBy` kombinieren, um eine Sequenz in einer einzelnen Anweisung zu transformieren, zu filtern und zu sortieren.

```
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
```

Ausgabe:

2
4
8

Live-Demo zu .NET-Geige

Alle Funktionen, die den generischen `IEnumerable<T>` -Typ erweitern und zurückgeben, können als verkettete Klauseln in einer einzelnen Anweisung verwendet werden. Diese Art der fließenden Programmierung ist leistungsstark und sollte bei der Erstellung eigener Erweiterungsmethoden berücksichtigt werden .

Bereich und Wiederholung

Mit den statischen Methoden `Range` und `Repeat` von `Enumerable` können einfache Sequenzen generiert werden.

Angebot

`Enumerable.Range()` generiert eine Folge von Ganzzahlen mit einem Startwert und einer Zählung.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Live-Demo zu .NET-Geige

Wiederholen

`Enumerable.Repeat()` generiert eine Folge sich wiederholender Elemente, die einem Element und der Anzahl der erforderlichen Wiederholungen zugeordnet sind.

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

Live-Demo zu .NET-Geige

Überspringen und nehmen

Die Skip-Methode gibt eine Auflistung zurück, die eine Anzahl von Elementen am Anfang der Quellauflistung ausschließt. Die Anzahl der ausgeschlossenen Elemente ist die als Argument angegebene Anzahl. Wenn die Sammlung weniger Elemente enthält als im Argument angegeben, wird eine leere Sammlung zurückgegeben.

Die Take-Methode gibt eine Auflistung zurück, die eine Reihe von Elementen vom Anfang der Quellauflistung enthält. Die Anzahl der enthaltenen Elemente ist die als Argument angegebene Anzahl. Wenn die Sammlung weniger Elemente enthält als im Argument angegeben, enthält die zurückgegebene Sammlung die gleichen Elemente wie die Quellensammlung.

```
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
```

Live-Demo zu .NET-Geige

Skip und Take werden häufig zusammen verwendet, um Ergebnisse zu paginieren, zum Beispiel:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

Warnung: LINQ to Entities unterstützt nur das Überspringen von bestellten Abfragen . Wenn Sie versuchen, Skip ohne Reihenfolge zu verwenden, erhalten Sie eine **NotSupportedException** mit der Meldung "Die Methode 'Skip' wird nur für sortierte Eingaben in LINQ to Entities unterstützt. Die Methode 'OrderBy' muss vor der Methode 'Skip' aufgerufen werden.

Zuerst FirstOrDefault, Last, LastOrDefault, Single und SingleOrDefault

Alle sechs Methoden geben einen einzelnen Wert des Sequenztyps zurück und können mit oder ohne Prädikat aufgerufen werden.

Abhängig von der Anzahl der Elemente, die dem `predicate` oder, wenn kein `predicate` ist, der Anzahl der Elemente in der Quellsequenz, verhalten sie sich wie folgt:

Zuerst()

- Gibt das erste Element einer Sequenz oder das erste Element zurück, das dem angegebenen `predicate` .
- Wenn die Sequenz keine Elemente enthält, wird eine `InvalidOperationException` mit der folgenden Meldung ausgelöst: "Sequenz enthält keine Elemente".
- Wenn die Sequenz keine Elemente enthält, die mit dem bereitgestellten `predicate` übereinstimmen, wird eine `InvalidOperationException` mit der Nachricht "Sequenz enthält kein übereinstimmendes Element" ausgelöst.

Beispiel

```
// 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();
```

Live-Demo zu .NET-Geige

FirstOrDefault ()

- Gibt das erste Element einer Sequenz oder das erste Element zurück, das dem

angegebenen `predicate` .
- Wenn die Sequenz keine Elemente oder keine Elemente enthält, die dem angegebenen `predicate` , wird der Standardwert des Sequenztyps mit `default(T)` .

Beispiel

```
// 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();
```

Live-Demo zu .NET-Geige

Zuletzt()

- Gibt das letzte Element einer Sequenz oder das letzte Element zurück, das mit dem angegebenen `predicate` übereinstimmt.
- Wenn die Sequenz keine Elemente enthält, wird eine `InvalidOperationException` mit der Meldung "Sequenz enthält keine Elemente" ausgelöst.
- Wenn die Sequenz keine Elemente enthält, die mit dem bereitgestellten `predicate` übereinstimmen, wird eine `InvalidOperationException` mit der Nachricht "Sequenz enthält kein übereinstimmendes Element" ausgelöst.

Beispiel

```
// 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 ()

- Gibt das letzte Element einer Sequenz oder das letzte Element zurück, das mit dem angegebenen `predicate` übereinstimmt.
- Wenn die Sequenz keine Elemente oder keine Elemente enthält, die dem angegebenen `predicate` , wird der Standardwert des Sequenztyps mit `default(T)` .

Beispiel

```
// 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();
```

Single()

- Wenn die Sequenz genau ein Element oder genau ein Element enthält, das mit dem angegebenen `predicate` übereinstimmt, wird dieses Element zurückgegeben.
- Wenn die Sequenz keine Elemente enthält oder keine Elemente, die dem angegebenen `predicate` , wird eine `InvalidOperationException` mit der Nachricht "Sequenz enthält keine Elemente" ausgelöst.
- Wenn die Sequenz mehr als ein Element oder mehr als ein Element enthält, das mit dem bereitgestellten `predicate` übereinstimmt, wird eine `InvalidOperationException` mit der Meldung "Sequenz enthält mehr als ein Element" ausgelöst.
- **Hinweis:** Um auszuwerten, ob die Sequenz genau ein Element enthält, müssen höchstens zwei Elemente aufgelistet werden.

Beispiel

```
// 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 ()

- Wenn die Sequenz genau ein Element oder genau ein Element enthält, das mit dem angegebenen `predicate` übereinstimmt, wird dieses Element zurückgegeben.
- Wenn die Sequenz keine Elemente enthält oder keine Elemente, die dem angegebenen `predicate` , wird `default(T)` zurückgegeben.
- Wenn die Sequenz mehr als ein Element oder mehr als ein Element enthält, das mit dem bereitgestellten `predicate` übereinstimmt, wird eine `InvalidOperationException` mit der Meldung "Sequenz enthält mehr als ein Element" ausgelöst.
- Wenn die Sequenz keine Elemente enthält, die dem bereitgestellten `predicate` , wird der Standardwert des Sequenztyps mit `default(T)` .
- **Hinweis:** Um auszuwerten, ob die Sequenz genau ein Element enthält, müssen höchstens zwei Elemente aufgelistet werden.

Beispiel

```
// 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();
```

Empfehlungen

- Obwohl Sie `FirstOrDefault` , `LastOrDefault` oder `SingleOrDefault` , um zu prüfen, ob eine

Sequenz Elemente enthält, sind `Any` oder `Count` zuverlässiger. Dies liegt daran, dass ein Rückgabewert von `default(T)` aus einer dieser drei Methoden nicht beweist, dass die Sequenz leer ist, da der Wert des ersten / letzten / einzelnen Elements der Sequenz gleichermaßen `default(T)`

- Entscheiden Sie, welche Methode für Ihren Code am besten geeignet ist. Verwenden Sie beispielsweise `Single` nur dann, wenn Sie sicherstellen müssen, dass ein einzelnes Element in der Auflistung Ihrem Prädikat entspricht. Andernfalls verwenden Sie `First` . als `Single` eine Ausnahme, wenn die Sequenz mehr als ein übereinstimmendes Element enthält. Dies gilt natürlich auch für die "* OrDefault" -Counterparts.

- In Bezug auf die Effizienz: Obwohl es oft angebracht ist, sicherzustellen, dass es nur ein Element (`Single`) oder entweder nur ein oder null (`SingleOrDefault`) `SingleOrDefault` , die von einer Abfrage zurückgegeben werden, erfordern beide Methoden mehr und oft die gesamte Sammlung der Auflistung geprüft werden, um sicherzustellen, dass keine zweite Übereinstimmung mit der Abfrage besteht. Dies unterscheidet sich beispielsweise vom Verhalten der `First` Methode, die nach dem Finden der ersten Übereinstimmung erfüllt werden kann.

Außer

Die Except-Methode gibt die Menge der Elemente zurück, die in der ersten Sammlung enthalten sind, aber nicht in der zweiten. Der Standard- `IEqualityComparer` wird verwendet, um die Elemente in den beiden Sätzen zu vergleichen. Es gibt eine Überladung, die einen `IEqualityComparer` als Argument akzeptiert.

Beispiel:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Ausgabe:

1
4

Live-Demo zu .NET-Geige

In diesem Fall `.Except(second)` schließt in dem Array enthaltenen Elemente `second` , nämlich 2 und 3 (0 und 5 sind nicht in der enthaltenen `first` Array und werden übersprungen).

Beachten Sie, dass `Except` impliziert `Distinct` (dh es werden wiederholte Elemente entfernt). Zum Beispiel:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
```

```
// inThirdButNotInSecond = { 1, 4 }
```

Ausgabe:

1
4

Live-Demo zu .NET-Geige

In diesem Fall werden die Elemente 1 und 4 nur einmal zurückgegeben.

`IEquatable` oder die Funktion eines `IEqualityComparer` , können Sie die Elemente mit einer anderen Methode vergleichen. Beachten Sie, dass die `GetHashCode` Methode auch überschrieben werden sollte, damit ein identischer Hashcode für ein `object` , das gemäß der `IEquatable` Implementierung identisch ist.

Beispiel mit IEquatable:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
```

```
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
    }
}
```

Ausgabe:

    Chanukka

Live-Demo zu .NET-Geige

SelectMany: Reduzieren einer Sequenz von Sequenzen

```
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 }
```

Verwenden Sie `SelectMany()` wenn Sie bereits eine Sequenz von Sequenzen haben oder erstellen, das Ergebnis jedoch als eine lange Sequenz erstellt werden soll.

In LINQ-Abfragesyntax:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

Wenn Sie über eine Sammlung von Sammlungen verfügen und gleichzeitig Daten aus der übergeordneten und `SelectMany` Sammlung `SelectMany` , ist dies auch mit `SelectMany` möglich.

Definieren wir einfache Klassen

```
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; }
}
```

Nehmen wir an, wir haben folgende Sammlung.

```
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"
            }
        }
    }
};
```

Jetzt möchten wir Kommentare `Content` zusammen mit der `Id` von `BlogPost` die diesem Kommentar zugeordnet ist. Dazu können wir eine entsprechende `SelectMany` Überladung verwenden.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId =
post.Id, CommentContent = comment.Content });
```

Unsere `commentsWithIds` sieht so aus

```
{
    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

Die SelectMany linq-Methode "flachtet" einen `IEnumerable<IEnumerable<T>>` `IEnumerable<T>` in einen `IEnumerable<T>` . Alle T-Elemente innerhalb der `IEnumerable` Instanzen, die im Quell- `IEnumerable` sind, werden zu einem einzigen `IEnumerable` .

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

Wenn Sie eine Auswahlfunktion verwenden, die Eingabeelemente in Sequenzen umwandelt, werden die Elemente dieser Sequenzen nacheinander zurückgegeben.

Beachten Sie, dass im Gegensatz zu `Select()` die Anzahl der Elemente in der Ausgabe nicht dieselbe sein muss wie in der Eingabe.

Mehr Praxisbeispiel

```
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);
}
```

Ausgabe:

Bob
Jack
Jim
John

Live-Demo zu .NET-Geige

Alles

`All` wird verwendet, um zu überprüfen, ob alle Elemente einer Sammlung einer Bedingung

entsprechen oder nicht.
siehe auch: .Any

1. Leerer Parameter

All : darf nicht mit leeren Parametern verwendet werden.

2. Lambda-Ausdruck als Parameter

All : Gibt " `true` wenn alle Elemente der Collection den Lambda-Ausdruck erfüllen, und " `false` :

```
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. Leere Sammlung

All : Gibt " `true` wenn die Auflistung leer ist und ein Lambda-Ausdruck angegeben wird:

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

Hinweis: `All` stoppen die Iteration der Sammlung, sobald ein Element gefunden wird, das **nicht** der Bedingung entspricht. Dies bedeutet, dass die Sammlung nicht unbedingt vollständig aufgezählt wird. Es wird nur so weit aufgezählt, dass der erste Artikel **nicht** der Bedingung entspricht.

Abfragesammlung nach Typ / Cast-Elementen zum Typ

```
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`

```
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

```
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
```

Union

Führt zwei Sammlungen zusammen, um mithilfe des Standardgleichheitsvergleichs eine eigene Sammlung zu erstellen

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

Live-Demo zu .NET-Geige

VERBINDUNGEN

Joins werden verwendet, um verschiedene Listen oder Tabellen, die Daten enthalten, über einen gemeinsamen Schlüssel zu kombinieren.

Wie in SQL werden in LINQ die folgenden Arten von Joins unterstützt:
Inner, links, rechts, Flanke und **Full Outer** Joins.

Die folgenden zwei Listen werden in den folgenden Beispielen verwendet:

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

(Inner) Join

```
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"}
```

Linke äußere Verbindung

```
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 });
```

Rechter äußerer Join

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};

// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

Cross Join

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

Voller äußerer Join

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
//         {null,"d"}
```

Praktisches Beispiel

Die obigen Beispiele haben eine einfache Datenstruktur, sodass Sie sich darauf konzentrieren können, die verschiedenen LINQ-Joins technisch zu verstehen. In der realen Welt würden Sie jedoch Tabellen mit Spalten haben, die Sie verknüpfen müssen.

Im folgenden Beispiel wird nur eine Klasse `Region` verwendet. In Wirklichkeit würden Sie zwei oder mehr verschiedene Tabellen mit demselben Schlüssel verknüpfen (in diesem Beispiel werden die `first` und die `second` über die gemeinsame Schlüssel- `ID`).

Beispiel: Betrachten Sie die folgende Datenstruktur:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Bereiten Sie nun die Daten vor (dh füllen Sie sie mit Daten auf):

```
// 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")
                };
```

Sie können in diesem Beispiel sehen , dass `first` keine Region Beschreibungen enthält , damit Sie sie aus anschließen möchten `second` . Dann würde der innere Join aussehen:

```
// 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"}
```

Dieses Ergebnis hat anonyme Objekte `select new { f.ID, s.RegionDescription };` erstellt, was in Ordnung ist, aber wir haben bereits eine richtige Klasse erstellt - so können wir sie angeben: Anstelle von `select new { f.ID, s.RegionDescription };` wir können " `select new Region(f.ID, s.RegionDescription);` , die dieselben Daten zurückgeben, aber Objekte vom Typ `Region` erstellen, wodurch die Kompatibilität mit den anderen Objekten erhalten bleibt.

Live-Demo zu .NET-Geige

Eindeutig

Gibt eindeutige Werte aus einem `IEnumerable` . Die Eindeutigkeit wird mit dem Standardgleichheitsvergleicher ermittelt.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

Um einen benutzerdefinierten Datentyp vergleichen zu können, müssen Sie die `IEquatable<T>` Schnittstelle `IEquatable<T>` implementieren und `GetHashCode` und `Equals` Methoden für den Typ bereitstellen. Oder der Gleichheitsvergleicher kann außer Kraft gesetzt werden:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

GroupBy ein oder mehrere Felder

Nehmen wir an, wir haben ein Filmmodell:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Gruppieren nach Kategorie-Eigenschaft:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
}
```

Gruppieren nach Kategorie und Jahr:

```
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();
}
```

Verwendung von Range mit verschiedenen Linq-Methoden

Sie können die Enumerable-Klasse zusammen mit Linq-Abfragen verwenden, um Loops in Linq-One-Liners zu konvertieren.

Wählen Sie Beispiel

Gegenteil davon:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

Du kannst das:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

Wo Beispiel

In diesem Beispiel werden 100 Zahlen generiert und sogar Zahlen extrahiert

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Abfrage Bestellung - OrderBy () ThenBy () OrderByDescending () ThenByDescending ()

```
string[] names= { "mark", "steve", "adam" };
```

Aufsteigend:

Abfragesyntax

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Methodensyntax

```
var sortedNames = names.OrderBy(name => name);
```

SortierteNames enthält die Namen in der folgenden Reihenfolge: "adam", "mark", "steve"

Absteigend:

Abfragesyntax

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

Methodensyntax

```
var sortedNames = names.OrderByDescending(name => name);
```

SortierteNames enthält die Namen in der folgenden Reihenfolge: "Steve", "Mark", "Adam"

Nach mehreren Feldern bestellen

```
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}
};
```

Abfragesyntax

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

Methodensyntax

```
 sortedPeople = people.OrderBy(person => person.LastName)
                      .ThenBy(person => person.FirstName)
                      .ThenByDescending(person => person.Age);
```

Ergebnis

```
1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins  28
4. Steve Collins 30
```

Grundlagen

LINQ ist für das Abfragen von Sammlungen (oder Arrays) von großem Nutzen.

Angenommen, die folgenden Beispieldaten:

```
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 }
}
```

Wir können diese Daten mithilfe der LINQ-Syntax "abfragen". So können Sie beispielsweise alle Schüler abrufen, die heute einen Snack erhalten:

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

Oder um Schüler mit einer Note von 90 oder höher abzurufen und nur ihren Namen und nicht das vollständige `Student` :

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

Die LINQ-Funktion besteht aus zwei Syntaxen, die die gleichen Funktionen ausführen, eine nahezu identische Leistung haben, jedoch sehr unterschiedlich geschrieben sind. Die Syntax im obigen Beispiel wird als **Abfragesyntax bezeichnet** . Das folgende Beispiel veranschaulicht jedoch die **Methodensyntax** . Die gleichen Daten werden wie im obigen Beispiel zurückgegeben. Die Abfrage ist jedoch anders.

```
var topStudentNames = classroom.Students
                                .Where(s => s.Grade >= 90)
                                .Select(s => s.Name);
```

Gruppiere nach

GroupBy ist eine einfache Möglichkeit, eine `IEnumerable<T>` -Sammlung von Elementen in verschiedene Gruppen zu sortieren.

Einfaches Beispiel

In diesem ersten Beispiel erhalten wir zwei Gruppen, ungerade und gerade Elemente.

```
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)
    }
}
```

Komplexeres Beispiel

Nehmen wir als Beispiel eine Liste von Personen nach Alter. Zuerst erstellen wir ein Personenobjekt mit zwei Eigenschaften, Name und Alter.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Dann erstellen wir eine Musterliste von Personen mit verschiedenen Namen und Alter.

```
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"});
```

Dann erstellen wir eine LINQ-Abfrage, um die Liste der Personen nach Alter zu gruppieren.

```
var query = people.GroupBy(x => x.Age);
```

Auf diese Weise können wir das Alter für jede Gruppe sehen und eine Liste aller Personen in der Gruppe haben.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

Daraus ergibt sich folgende Ausgabe:

```
20
Mouse
30
Neo
Trinity
40
Morpheus
Dozer
Smith
```

Sie können mit der Live-Demo auf .NET Fiddle spielen

Irgendein

`Any` wird geprüft, ob **ein** Element einer Collection einer Bedingung entspricht oder nicht.
Siehe auch: .All , Any und FirstOrDefault: Best Practice

1. Leerer Parameter

Any : Gibt `true` wenn die Collection Elemente enthält, und `false` wenn die Collection leer ist:

```
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. Lambda-Ausdruck als Parameter

Any : Gibt `true` wenn die Auflistung ein oder mehrere Elemente enthält, die die Bedingung im Lambda-Ausdruck erfüllen:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

3. Leere Sammlung

Any : Gibt `false` wenn die Sammlung leer ist und ein Lambda-Ausdruck angegeben wird:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Hinweis: `Any` stoppt die Iteration der Sammlung, sobald ein der Bedingung entsprechendes Element gefunden wird. Dies bedeutet, dass die Sammlung nicht unbedingt vollständig aufgezählt wird. Es wird nur so weit aufgezählt, dass der erste Artikel gefunden wird, der der Bedingung entspricht.

Live-Demo zu .NET-Geige

ToDictionary

Mit der `ToDictionary()` LINQ-Methode kann eine `Dictionary<TKey, TElement>` -Sammlung basierend auf einer bestimmten `IEnumerable<T>` .

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

In diesem Beispiel hat das einzige an `ToDictionary` Argument den Typ `Func<TSource, TKey>` , der den Schlüssel für jedes Element zurückgibt.

Dies ist eine prägnante Möglichkeit, die folgende Operation auszuführen:

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

Sie können auch einen zweiten Parameter an die `ToDictionary` Methode übergeben, der vom Typ `Func<TSource, TElement>` und den für jeden Eintrag hinzuzufügenden `Value` zurückgibt.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

Es ist auch möglich, den `IComparer` anzugeben, der zum Vergleich von Schlüsselwerten verwendet wird. Dies kann nützlich sein, wenn der Schlüssel eine Zeichenfolge ist und die Groß- / Kleinschreibung berücksichtigt werden soll.

```
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
```

Hinweis: Für die `ToDictionary` Methode müssen alle Schlüssel eindeutig sein. Es dürfen keine doppelten Schlüssel vorhanden sein. Wenn `ArgumentException: An item with the same key has already been added.` , wird eine Ausnahme ausgelöst: `ArgumentException: An item with the same key has already been added.` Wenn Sie ein Szenario haben, in dem Sie wissen, dass Sie mehrere Elemente mit demselben Schlüssel haben werden, sollten Sie lieber `ToLookup` verwenden.

Aggregat

`Aggregate` Wendet eine Akkumulatorfunktion auf eine Sequenz an.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- Im ersten Schritt ist `prevSum = 1`
- Am zweiten `prevSum = prevSum(at the first step) + 2`
- Im i-ten Schritt `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 !"
```

Bei einer zweiten Überladung von `Aggregate` auch ein `seed` Parameter empfangen, der der anfängliche Akkumulatorwert ist. Dies kann verwendet werden, um mehrere Bedingungen für eine Sammlung zu berechnen, ohne sie mehrmals zu durchlaufen.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Für die Sammlung von `items` wollen wir berechnen

1. Die Summe `.Count`
2. Die Anzahl der geraden Zahlen
3. Sammle jeden weiteren Gegenstand ein

Mit `Aggregate` kann man so vorgehen:

```
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]
```

Beachten Sie, dass bei Verwendung eines anonymen Typs als Startwert ein neues Objekt für jedes Element instanziiert werden muss, da die Eigenschaften schreibgeschützt sind. Mit einer benutzerdefinierten Klasse kann man die Informationen einfach zuweisen und es ist keine `new` erforderlich (nur wenn der anfängliche `seed` angegeben wird)

Definieren einer Variablen in einer Linq-Abfrage (Schlüsselwort let)

Um eine Variable innerhalb eines linq-Ausdrucks zu definieren, können Sie das **let-** Schlüsselwort verwenden. Dies geschieht normalerweise, um die Ergebnisse von Zwischenabfragen zu speichern, zum Beispiel:

```
 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));
```

```
}
```

Ausgabe:

Der Durchschnitt der Zahlen ist 4,5.
Das Abfrageergebnis enthält die Nummer 3 mit dem Quadrat von 9.
Das Abfrageergebnis enthält die Nummer 4 mit dem Quadrat von 16.
Das Abfrageergebnis enthält die Nummer 5 mit dem Quadrat von 25.
Das Abfrageergebnis enthält die Nummer 6 mit dem Quadrat von 36.
Das Abfrageergebnis enthält die Nummer 7 mit dem Quadrat von 49.
Das Abfrageergebnis enthält die Nummer 8 mit einem Quadrat von 64.
Das Abfrageergebnis enthält die Nummer 9 mit dem Quadrat von 81.

Demo anzeigen

SkipWährend

`SkipWhile()` wird verwendet, um Elemente bis zur ersten Nichtübereinstimmung auszuschließen.

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

DefaultIfEmpty

DefaultIfEmpty wird verwendet, um ein Standardelement zurückzugeben, wenn die Sequenz keine Elemente enthält. Dieses Element kann der Standard des Typs oder eine benutzerdefinierte Instanz dieses Typs sein. Beispiel:

```
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;
```

Verwendung in Left Joins :

Mit `DefaultIfEmpty` der herkömmliche Linq-Join ein Standardobjekt zurückgeben, wenn keine Übereinstimmung gefunden wurde. So fungiert sie als linker Join von SQL. Beispiel:

```
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` ein `DefaultIfEmpty` verwendet wird (ohne einen Standardwert anzugeben) und dies zu keinen übereinstimmenden Elementen in der richtigen Sequenz führt, müssen Sie sicherstellen, dass das Objekt nicht `null` bevor Sie auf seine Eigenschaften zugreifen. Andernfalls führt dies zu einer `NullReferenceException` . Beispiel:

```
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` wird verwendet, um zwei `IEnumerable<T>` -Sequenzen miteinander zu vergleichen.

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

```
bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Count und LongCount

`Count` gibt die Anzahl der Elemente in einem `IEnumerable<T>` . `Count` stellt auch einen optionalen Prädikatparameter bereit, mit dem Sie die Elemente filtern können, die Sie zählen möchten.

```
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` funktioniert genauso wie `Count` , hat jedoch den Rückgabetyp `long` und wird zum Zählen von `IEnumerable<T>` -Sequenzen verwendet, die länger als `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
```

Inkrementelles Erstellen einer Abfrage

Da LINQ die **verzögerte Ausführung verwendet** , können wir ein Abfrageobjekt haben, das die Werte nicht tatsächlich enthält, aber bei der Auswertung die Werte zurückgibt. Auf diese Weise können wir die Abfrage basierend auf unserem Kontrollfluss dynamisch erstellen und auswerten, sobald wir fertig sind:

```
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
        });
```

Wir können Filter bedingt anwenden:

```
    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);
        }
    }
```

Wir können der Abfrage eine Sortierreihenfolge basierend auf einer Bedingung hinzufügen:

```
    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.
    }
```

Unsere Abfrage kann so definiert werden, dass sie an einem bestimmten Punkt beginnt:

```
    query = query.Skip(start - 1);
```

und definiert, um eine bestimmte Anzahl von Datensätzen zurückzugeben:

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
```

```
}
```

Sobald wir das Abfrageobjekt haben, können wir die Ergebnisse mit einer `foreach` Schleife oder einer der LINQ-Methoden auswerten, die eine Menge von Werten wie `ToList` oder `ToArray` :

```
SearchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

Postleitzahl

Die `Zip` Erweiterungsmethode wirkt auf zwei Sammlungen. Es paart jedes Element in den beiden Reihen nach Position. Bei einer `Func` Instanz verwenden wir `Zip` , um Elemente aus den beiden C # -Sammlungen paarweise zu behandeln. Wenn sich die Serien in der Größe unterscheiden, werden die zusätzlichen Elemente der größeren Serie ignoriert.

Um ein Beispiel aus dem Buch "C # in a Nutshell" zu nehmen,

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

Ausgabe:

3 = drei
5 = fünf
7 = sieben

Demo anzeigen

GroupJoin mit Variable für den äußeren Bereich

```
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 und ElementAtOrDefault

`ElementAt`

gibt das Element am Index `n` . Wenn sich `n` nicht innerhalb des Aufzählungsbereichs befindet, wird eine `ArgumentOutOfRangeException` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` gibt das Element an Index `n` . Wenn `n` nicht innerhalb des Aufzählungsbereichs liegt, wird ein `default(T)` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

Sowohl `ElementAt` als auch `ElementAtOrDefault` sind optimiert, wenn die Quelle eine `IList<T>` und in diesen Fällen die normale Indizierung verwendet wird.

Beachten Sie, dass für `ElementAt` , wenn der bereitgestellte Index größer als `IList<T>` , die Liste eine `ArgumentOutOfRangeException` `IList<T>` soll (technisch gesehen jedoch nicht garantiert).

Linq-Quantifizierer

Quantifiziereroperationen geben einen booleschen Wert zurück, wenn einige oder alle Elemente in einer Sequenz eine Bedingung erfüllen. In diesem Artikel werden einige allgemeine LINQ to Objects-Szenarien beschrieben, in denen wir diese Operatoren verwenden können. Es gibt drei Quantifizierervorgänge, die in LINQ verwendet werden können:

`All` - Wird verwendet, um zu bestimmen, ob alle Elemente einer Sequenz eine Bedingung erfüllen. Z.B:

```
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` - `Any` verwendet, um zu bestimmen, ob Elemente in einer Sequenz eine Bedingung erfüllen. Z.B:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` - Bestimmt, ob eine Sequenz ein bestimmtes Element enthält. Z.B:

```
//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");
```

Mehrere Sequenzen verbinden

Betrachten Sie die Entitäten `Customer` , `Purchase` und `PurchaseItem` wie folgt:

```
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; }
}
```

Betrachten Sie die folgenden Beispieldaten für die obigen Entitäten:

```
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"
    }
};
```

Betrachten Sie nun die folgende linq-Abfrage:

```
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
             };
```

So geben Sie das Ergebnis der obigen Abfrage aus:

```
foreach(var resultItem in result)
```

```
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

Die Ausgabe der Abfrage wäre:

Kunde1, Kunde1-Kauf1, Kauf1-PurchaseItem1

Kunde1, Kunde1-Einkauf2, Einkauf2-Einkaufselement1

Kunde1, Kunde1-Kauf2, Kauf2-PurchaseItem2

Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

Live-Demo zu .NET-Geige

Zusammenfügen auf mehreren Schlüsseln

```
  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
        };
```

Beachten Sie, dass anonyme Typen in der obigen `join` dieselben Eigenschaften enthalten müssen, da Objekte nur dann als gleich betrachtet werden, wenn alle ihre Eigenschaften gleich sind. Andernfalls wird die Abfrage nicht kompiliert.

Wählen Sie mit Func Selector - Verwenden Sie diese Option, um die Rangfolge der Elemente abzurufen

Auf den Überlastungen der `Select` Erweiterungsmethoden führt auch den `index` des aktuellen Elements in der Sammlung seines `select` hrsg. Dies sind einige Anwendungen davon.

Holen Sie sich die "Zeilennummer" der Artikel

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

Liefert den Rang eines Gegenstandes *innerhalb* seiner Gruppe

```
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();
```

Rufen Sie das Ranking von Gruppen ab (auch in Oracle als dense_rank bekannt).

```
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();
```

Zum Testen können Sie Folgendes verwenden:

```
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);
    }
}
```

Und Daten:

```
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` gibt Elemente aus einer Sequenz zurück, solange die Bedingung erfüllt ist

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

Summe

Die Erweiterungsmethode `Enumerable.Sum` berechnet die Summe der numerischen Werte.

Falls die Elemente der Sammlung selbst Zahlen sind, können Sie die Summe direkt berechnen.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

Wenn der Typ der Elemente ein komplexer Typ ist, können Sie einen Lambda-Ausdruck verwenden, um den Wert anzugeben, der berechnet werden soll:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Die Summenerweiterungsmethode kann mit den folgenden Typen berechnen:

- Int32
- Int64
- Single
- Doppelt
- Dezimal

Wenn Ihre Sammlung nullfähige Typen enthält, können Sie den Nullkoaleszenzoperator verwenden, um einen Standardwert für Nullelemente festzulegen:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

Nachschlagen

> ToLookup gibt eine Datenstruktur zurück, die die Indizierung ermöglicht. Es ist eine Erweiterungsmethode. Sie erzeugt eine ILookup-Instanz, die mit einer foreach-Schleife indiziert oder aufgezählt werden kann. Die Einträge werden bei jeder Taste zu Gruppierungen zusammengefasst. - 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
```

Ein anderes Beispiel:

```
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
```

Erstellen Sie Ihre eigenen Linq-Operatoren für IEnumerable

Eines der großen Dinge an Linq ist, dass es so einfach zu erweitern ist. Sie müssen lediglich eine Erweiterungsmethode erstellen , deren Argument `IEnumerable<T>` .

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

In diesem Beispiel werden die Elemente in einem `IEnumerable<T>` in Listen fester Größe aufgeteilt. Die letzte Liste enthält den Rest der Elemente. Beachten Sie, wie das Objekt , auf dem das Verlängerungsverfahren angewendet wird , wird in (Argument übergeben `source`) als das anfängliche Argument die Verwendung `this` Schlüsselwort. Dann wird das `yield` Schlüsselwort verwendet, um das nächste Element im Ausgabe- `IEnumerable<T>` bevor die Ausführung ab diesem Punkt fortgesetzt wird (siehe Yield-Schlüsselwort).

Dieses Beispiel würde in Ihrem Code folgendermaßen verwendet:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
```

```
}
```

In der ersten Schleife wäre die Unterliste `{2, 3, 4}` und in der zweiten `{5, 6}` .

Benutzerdefinierte LinQ-Methoden können auch mit Standard-LinQ-Methoden kombiniert werden. z.B:

```
//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
```

Diese Abfrage gibt gerade Zahlen zurück, die in Gruppen mit einer Größe von 3 gruppiert sind: `{0, 2, 4}, {6, 8, 10}, {12}`

Denken Sie daran, dass Sie `using MyNamespace;` Zeile, um auf die Erweiterungsmethode zugreifen zu können.

Verwenden von SelectMany anstelle von verschachtelten Schleifen

Gegeben 2 Listen

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

Wenn Sie alle Permutationen ausgeben möchten, können Sie verschachtelte Schleifen wie verwenden

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

Mit SelectMany können Sie dieselbe Operation wie ausführen

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Any and First (OrDefault) - Best Practice

Ich werde nicht erklären, was `Any` und `FirstOrDefault` , da es bereits zwei gute Beispiele dafür gibt. Weitere Informationen finden Sie unter Any und First, FirstOrDefault, Last, LastOrDefault, Single und SingleOrDefault .

Ein Muster, das ich oft im Code sehe, **sollte vermieden** werden

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
```

```
}
```

Es könnte so effizienter geschrieben werden

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

Wenn Sie das zweite Beispiel verwenden, wird die Sammlung nur einmal durchsucht und ergibt dasselbe Ergebnis wie das erste. Die gleiche Idee kann auf `Single` angewendet werden.

GroupBy-Summe und Anzahl

Nehmen wir eine Probestunde an:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Lassen Sie uns nun eine Liste von Transaktionen betrachten:

```
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) },
};
```

Wenn Sie die kategoriebezogene Summe aus Betrag und Anzahl berechnen möchten, können Sie GroupBy folgendermaßen verwenden:

```
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}"));
```

Alternativ können Sie dies in einem Schritt tun:

```
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}"));
```

Die Ausgabe für beide obigen Abfragen wäre gleich:

> Kategorie: Konto speichern, Betrag: 66, Anzahl: 2
>
> Kategorie: Kreditkarte, Betrag: 71, Anzahl: 2
>
> Kategorie: Laufendes Konto, Anzahl: 100, Anzahl: 1

Live-Demo in .NET-Geige

Umkehren

- Kehrt die Reihenfolge der Elemente in einer Reihenfolge um.
- Wenn keine Elemente vorhanden sind, wird eine `ArgumentNullException: source is null.`

Beispiel:

```
// 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);
```

Live-Code-Beispiel

Beachten Sie, dass `Reverse()` abhängig von der Kettenreihenfolge Ihrer LINQ-Anweisungen möglicherweise anders arbeitet.

```
        //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
```

Live-Code-Beispiel

Reverse () funktioniert, indem es alles puffert und dann rückwärts durchläuft, was nicht sehr effizient ist, aber OrderBy ist diesbezüglich auch nicht.

In LINQ-to-Objects gibt es Pufferoperationen (Reverse, OrderBy, GroupBy usw.) und Nichtpufferoperationen (Where, Take, Skip usw.).

Beispiel: Nichtpufferung Rückwärtsverlängerung

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Live-Code-Beispiel

Diese Methode kann auf Probleme stoßen, wenn Sie die Liste während der Iteration ändern.

Aufzählung der Aufzählungszeichen

Die IEnumerable <T> -Schnittstelle ist die Basisschnittstelle für alle generischen Enumeratoren und ein wesentlicher Bestandteil des Verständnisses von LINQ. Im Kern repräsentiert es die Sequenz.

Diese zugrunde liegende Schnittstelle wird von allen generischen Sammlungen vererbt, z. B. Collection <T> , Array , List <T> , Dictionary <TKey, TValue> -Klasse und HashSet <T> .

Zusätzlich zur Darstellung der Sequenz muss jede Klasse, die von IEnumerable <T> erbt, einen IEnumerator <T> bereitstellen. Der Enumerator macht den Iterator für das Enumerable verfügbar, und diese beiden miteinander verbundenen Schnittstellen und Ideen sind die Quelle für das Sprichwort "Enumerate the Enumerable".

"Aufzählen des Aufzählers" ist ein wichtiger Satz. Das Aufzählbare ist einfach eine Struktur für die Wiederholung, es enthält keine materialisierten Objekte. Bei der Sortierung kann ein Aufzählungszeichen beispielsweise die Kriterien des Felds für die Sortierung enthalten. `.OrderBy()` jedoch `.OrderBy()` , wird ein IEnumerable <T> zurückgegeben, das nur wissen kann, *wie* es sortiert werden soll. Die Verwendung eines Aufrufs, der die Objekte materialisiert, wie beim Durchlaufen der Menge, wird als Aufzählung bezeichnet (z. B. `.ToList()`). Der Aufzählungsprozess verwendet die aufzählbare Definition, *wie* die Reihe durchlaufen werden soll und die relevanten Objekte (in der Reihenfolge gefiltert, projiziert usw.) zurückgegeben werden.

Erst wenn das Aufzählungszeichen aufgezählt wurde, führt dies zur Materialisierung der Objekte. Dies ist der Zeitpunkt, zu dem Metriken wie Zeitkomplexität (wie lange sie in Bezug auf die

Seriengröße benötigt wird) und räumliche Komplexität (wie viel Platz in Bezug auf die Seriengröße beansprucht wird) können gemessen werden.

Das Erstellen einer eigenen Klasse, die von IEnumerable <T> erbt, kann etwas kompliziert sein, abhängig von der zugrunde liegenden Reihe, die aufgezählt werden muss. Im Allgemeinen ist es am besten, eine der vorhandenen generischen Sammlungen zu verwenden. Das heißt, es ist auch möglich, von der IEnumerable <T> -Schnittstelle zu erben, ohne ein definiertes Array als zugrunde liegende Struktur zu haben.

Verwenden Sie beispielsweise die Fibonacci-Serie als zugrunde liegende Sequenz. Beachten Sie, dass mit dem Aufruf von `Where` einfach ein `IEnumerable` wird. Erst wenn ein Aufruf zur `IEnumerable` der Aufzählungszeichen erfolgt, werden alle Werte `IEnumerable` .

```
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;
        }
    }
}
```

Ausgabe

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

Die Stärke des zweiten Satzes (des fibMod612) ist, dass, obwohl wir den Aufruf zur Bestellung unseres gesamten Satzes von Fibonacci-Zahlen gemacht haben, da nur ein Wert mit `.First()` die Zeitkomplexität O (n) als nur 1 Wert war musste während der Ausführung des Bestellalgorithmus verglichen werden. Dies liegt daran, dass unser Enumerator nur nach einem Wert gefragt hat und die gesamte Aufzählung nicht materialisiert werden musste. Hätten wir `.Take(5)` anstelle von `.First()` der Enumerator nach 5 Werten gefragt, und höchstens 5 Werte müssten `.First()` werden. Im Vergleich zur Notwendigkeit, einen ganzen Satz zu bestellen *und dann* die ersten 5 Werte zu übernehmen, spart das Prinzip viel Ausführungszeit und -raum.

Sortieren nach

> Bestellt eine Sammlung mit einem angegebenen Wert.

Wenn der Wert eine **Ganzzahl** , ein **Doppel-** oder ein **Gleitkommawert ist** , beginnt er mit dem *Minimalwert* , was bedeutet, dass Sie zuerst die negativen Werte und dann die Nullwerte und danach die positiven Werte erhalten (siehe Beispiel 1).

Wenn man von einem **char** Um das Verfahren vergleicht den *ASCII - Wert* der Zeichen , die Sammlung zu sortieren (siehe Beispiel 2).

Wenn Sie **Zeichenfolgen** sortieren, vergleicht die OrderBy-Methode sie, indem Sie deren CultureInfo- Informationen betrachten, die jedoch normalerweise mit dem *ersten Buchstaben* des Alphabets beginnen (a, b, c ...).

Diese Art von Reihenfolge wird als aufsteigend bezeichnet, wenn Sie es andersherum wünschen, müssen Sie absteigend (siehe OrderByDescending).

Beispiel 1:

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

Beispiel 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', '{' }
```

Beispiel:

```
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

> Bestellt eine Sammlung mit einem angegebenen Wert.

Wenn der Wert eine **Ganzzahl** , ein **Doppel-** oder ein **Gleitkommawert ist** , beginnt er mit dem *Maximalwert* , was bedeutet, dass Sie zuerst die positiven Werte und dann die Nullwerte und danach die negativen Werte erhalten.

Wenn man von einem **char** Um das Verfahren vergleicht den *ASCII - Wert* der Zeichen , die Sammlung zu sortieren (siehe Beispiel 2).

Wenn Sie **Strings** sortieren, vergleicht die OrderBy-Methode sie, indem Sie deren CultureInfo-Informationen betrachten , die normalerweise mit dem *letzten Buchstaben* des Alphabets beginnen (z, y, x, ...).

Diese Art von Reihenfolge wird als absteigend bezeichnet, wenn Sie es andersherum wünschen, müssen Sie aufsteigend (siehe OrderBy).

Beispiel 1:

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

Beispiel 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', '+', '!', ' ' }
```

Beispiel 3:

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}
```

```
var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

Concat

Führt zwei Sammlungen zusammen (ohne Duplikate zu entfernen)

```
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
```

Enthält

MSDN:

> Bestimmt anhand eines angegebenen `IEqualityComparer<T>` ob eine Sequenz ein bestimmtes Element enthält

```
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
```

Verwenden eines benutzerdefinierten Objekts:

```
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
```

Verwenden der `Enumerable.Contains(value, comparer)` :

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

Eine intelligente Verwendung von `Contains` darin, mehrere `if` Klauseln für einen `Contains` Aufruf zu ersetzen.

Also statt dies zu tun:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Mach das:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

LINQ-Abfragen online lesen: https://riptutorial.com/de/csharp/topic/68/linq-abfragen

Kapitel 95: Literale

Syntax

- **bool:** wahr oder falsch
- **Byte:** Keiner, Integer-Literal wird implizit aus Int konvertiert
- **sbyte:** Keiner, Integer-Literal wird implizit aus int konvertiert
- **char:** Umschließen Sie den Wert in einfache Anführungszeichen
- **dezimal:** M oder m
- **double:** D, d oder eine reelle Zahl
- **float:** F oder f
- **int:** Keine, Standardwert für ganzzahlige Werte im Bereich von int
- **uint:** U, u oder ganzzahlige Werte innerhalb des Bereichs von uint
- **long:** L, l oder ganzzahlige Werte im Bereich von long
- **ulong:** UL, ul, ul, ul, LU, lu, Lu, lU oder ganzzahlige Werte im Bereich von ulong
- **short:** None, Integer-Literal implizit aus int konvertiert
- **ushort:** Keiner, Integer-Literal wird implizit aus int konvertiert
- **Zeichenfolge:** Der Wert wird in doppelte Anführungszeichen gesetzt, optional mit `@` vorangestellt.
- **null** : Das literal `null`

Examples

int Literale

`int` Literale werden definiert, indem ganzzahlige Werte innerhalb des Bereichs von `int` :

```
int i = 5;
```

Literale

`uint` Literale werden definiert, indem das Suffix `U` oder `u` , oder indem Integralwerte innerhalb des Bereichs von `uint` :

```
uint ui = 5U;
```

String-Literale

`string` werden definiert, indem der Wert in doppelte Anführungszeichen gesetzt wird `"` :

```
string s = "hello, this is a string literal";
```

String-Literale können Escape-Sequenzen enthalten. Siehe String-Escape-Sequenzen

Darüber hinaus unterstützt C # wörtliche Zeichenfolgenliterale (siehe Verbatim-Zeichenfolgen). Diese werden definiert, indem der Wert in doppelte Anführungszeichen gesetzt wird `"` und mit `@` vorangestellt wird. Escape-Sequenzen werden in wörtlichen String-Literalen ignoriert.

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

Char Literals

`char` - Literale werden durch Umwickeln Sie den Wert mit einfachen Anführungszeichen definiert `'` :

```
char c = 'h';
```

Zeichenliterale können Escape-Sequenzen enthalten. Siehe String-Escape-Sequenzen

Ein Zeichenliteral muss genau ein Zeichen lang sein (nachdem alle Escape-Sequenzen ausgewertet wurden). Leere Zeichenliterale sind nicht gültig. Das Standardzeichen (das von `default(char)` oder `new char()`) ist `'\0'` oder das NULL-Zeichen (nicht zu verwechseln mit dem `null` Literal und den Null-Referenzen).

Byte Literale

`byte` Typ hat kein literales Suffix. Integer-Literale werden implizit aus `int` konvertiert:

```
byte b = 127;
```

sbyte Literale

`sbyte` type hat kein literales Suffix. Integer-Literale werden implizit aus `int` konvertiert:

```
sbyte sb = 127;
```

Dezimal Literale

`decimal` Literale werden mit dem Suffix M oder m für eine reelle Zahl definiert:

```
decimal m = 30.5M;
```

Doppelliterale

`double` werden mit dem Suffix D oder d oder mit einer reellen Zahl definiert:

```
double d = 30.5D;
```

Float Literale

`float` Literale werden mit dem Suffix F oder f oder mit einer reellen Zahl definiert

```
float f = 30.5F;
```

lange Literale

`long` Literale werden definiert, indem das Suffix `L` oder `l` , oder indem ein Integralwert im Bereich von `long` :

```
long l = 5L;
```

Ulong wörtlich

`ulong` Literalen sind definiert durch Verwendung der Nachsilbe `UL` , `ul` , `Ul` , `uL` , `LU` , `lu` , `Lu` oder `lU` , oder durch ein integrales Werte innerhalb des Bereichs des Verwendens `ulong` :

```
ulong ul = 5UL;
```

kurz wörtlich

`short` hat kein Literal. Integer-Literale werden implizit aus `int` konvertiert:

```
short s = 127;
```

Ushort wörtlich

`ushort` type hat kein Suffix. Integer-Literale werden implizit aus `int` konvertiert:

```
ushort us = 127;
```

Bool Literale

`bool` Literale sind entweder `true` oder `false` ;

```
bool b = true;
```

Literale online lesen: https://riptutorial.com/de/csharp/topic/2655/literale

Kapitel 96: Lock-Anweisung

Syntax

- sperren (obj) {}

Bemerkungen

Mit der `lock` Anweisung können Sie den Zugriff verschiedener Threads auf Code innerhalb des Codeblocks steuern. Sie wird normalerweise verwendet, um Race-Bedingungen zu verhindern, z. B. mehrere Threads, die Elemente aus einer Sammlung lesen und entfernen. Da das Sperren dazu führt, dass Threads warten, bis andere Threads einen Codeblock verlassen, kann dies zu Verzögerungen führen, die mit anderen Synchronisationsmethoden behoben werden könnten.

MSDN

> Das Schlüsselwort lock kennzeichnet einen Anweisungsblock als einen kritischen Abschnitt, indem die gegenseitige Ausschlusssperre für ein bestimmtes Objekt abgerufen, eine Anweisung ausgeführt und dann die Sperre freigegeben wird.
>
> Das Schlüsselwort lock stellt sicher, dass ein Thread keinen kritischen Codeabschnitt eingibt, während sich ein anderer Thread im kritischen Abschnitt befindet. Wenn ein anderer Thread versucht, einen gesperrten Code einzugeben, wartet er, blockiert, bis das Objekt freigegeben wird.
>
> Es empfiehlt sich, ein **privates** Objekt zum Sperren oder eine **private statische** Objektvariable zu definieren, um Daten zu schützen, die in allen Instanzen gemeinsam sind.

In C # 5.0 und höher entspricht die `lock` :

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

Für C # 4.0 und früher entspricht die `lock` :

```
System.Threading.Monitor.Enter(refObject);
try
```

```
{
    // code
}
finally
{
     System.Threading.Monitor.Exit(refObject);
}
```

Examples

Einfache Benutzung

Die häufige Verwendung von `lock` ist ein kritischer Abschnitt.

Im folgenden Beispiel soll `ReserveRoom` aus verschiedenen Threads aufgerufen werden. Die Synchronisierung mit der `lock` ist der einfachste Weg, um einen Race-Zustand zu verhindern. Der Methodenkörper ist mit einer `lock` umgeben `lock` die sicherstellt, dass zwei oder mehr Threads sie nicht gleichzeitig ausführen können.

```
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
        }
    }
}
```

Wenn ein Thread erreicht `lock` -ed Block während eines anderen Thread innerhalb es läuft, wird die ehemaligen andere warten , um den Block zu verlassen.

> Es empfiehlt sich, ein privates Objekt zum Sperren oder eine private statische Objektvariable zu definieren, um Daten zu schützen, die für alle Instanzen gelten.

Ausnahme in einer Sperranweisung

Der folgende Code gibt die Sperre frei. Es wird kein Problem geben. Hinter den Kulissen funktioniert das Lock-Statement als `try finally`

```
lock(locker)
{
    throw new Exception();
}
```

Weitere Informationen finden Sie in der C # 5.0-Spezifikation :

Eine `lock` des Formulars

```
lock (x) ...
```

wobei `x` ein Ausdruck eines *Referenztyps ist* , ist genau äquivalent zu

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

mit der Ausnahme, dass `x` nur einmal ausgewertet wird.

Rückgabe in einer Sperranweisung

Der folgende Code gibt die Sperre frei.

```
lock(locker)
{
    return 5;
}
```

Für eine detaillierte Erklärung wird diese SO-Antwort empfohlen.

Instanzen von Object für die Sperre verwenden

Wenn Sie die eingebaute `lock` Anweisung von C # verwenden, ist eine Instanz eines bestimmten Typs erforderlich, der Status ist jedoch unerheblich. Eine Instanz eines `object` ist dafür perfekt:

```
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 . Instanzen von `Type` sollten dafür nicht verwendet werden (im obigen Code von `typeof(ThreadSafe)`), da Instances von `Type` AppDomains gemeinsam genutzt werden und der Umfang der Sperre erwartungsgemäß Code enthalten kann, in den er nicht sollte (z. B. wenn `ThreadSafe` geladen wird zwei AppDomains in demselben Prozess, die dann für ihre `Type` Instanz gesperrt werden, sperren sich gegenseitig).

Anti-Patterns und Gotchas

Sperren für eine stapelzugeordnete / lokale Variable

Einer der Fehler bei der Verwendung von `lock` ist die Verwendung lokaler Objekte als Schließfach in einer Funktion. Da diese lokalen Objektinstanzen bei jedem Aufruf der Funktion unterschiedlich sind, wird die `lock` nicht wie erwartet ausgeführt.

```
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);
    }
}
```

Angenommen, das Sperren schränkt den Zugriff auf das Synchronisierungsobjekt selbst ein

Wenn ein Thread aufruft: `lock(obj)` und ein anderer Thread ruft `obj.ToString()` zweite Thread nicht blockiert.

```
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
 }
```

Erwarten, dass Unterklassen wissen, wann gesperrt werden soll

Manchmal werden Basisklassen so konzipiert, dass ihre Unterklassen beim Zugriff auf bestimmte geschützte Felder eine Sperre verwenden müssen:

```
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!
    }
}
```

Es ist viel sicherer, *das Sperren* mithilfe einer Vorlagenmethode zu *kapseln* :

```
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
    }
}
```

Das Sperren einer ValueType-Variablen mit Box wird nicht synchronisiert

Im folgenden Beispiel wird eine private Variable implizit eingebettet, da sie als `object` für eine Funktion bereitgestellt wird und erwartet, dass eine Überwachungsressource gesperrt wird. Das Boxen tritt direkt vor dem Aufrufen der IncInSync-Funktion auf, sodass die Box-Instanz bei jedem Aufruf der Funktion einem anderen Heap-Objekt entspricht.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

Boxen erfolgt in der `Inc` Funktion:

```
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
```

Das bedeutet nicht, dass ein geschachtelter ValueType überhaupt nicht für das Sperren des Monitors verwendet werden kann:

```
private readonly object counterLock = 1;
```

Das Boxen wird jetzt im Konstruktor ausgeführt, was für das Sperren geeignet ist:

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

Sperren unnötig verwenden, wenn eine sicherere Alternative vorhanden ist

Ein sehr verbreitetes Muster ist die Verwendung einer privaten `List` oder eines privaten `Dictionary` in einer sicheren Klasse für den Thread und die Sperre bei jedem Zugriff:

```
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 */
}
```

Wenn mehrere Methoden auf das `values` zugreifen, kann der Code sehr lang werden und, was wichtiger ist, die permanente Sperrung verdeckt seine *Absicht* . Die Verriegelung ist auch sehr leicht zu vergessen und das Fehlen einer ordnungsgemäßen Verriegelung kann dazu führen, dass Fehler nur schwer gefunden werden.

Durch die Verwendung eines `ConcurrentDictionary` können Sie das Sperren vollständig vermeiden:

```
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 */
}
```

Die Verwendung von gleichzeitigen Auflistungen verbessert auch die Leistung, da alle bis zu einem gewissen Grad sperrenfreie Techniken verwenden .

Lock-Anweisung online lesen: https://riptutorial.com/de/csharp/topic/1495/lock-anweisung

Kapitel 97: Looping

Examples

Schleifenarten

Während

Der trivialste Schleifentyp. Der einzige Nachteil ist, dass es keinen intrinsischen Hinweis gibt, wo Sie sich in der Schleife befinden.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Tun

Ähnlich wie `while` , aber die Bedingung wird am Ende der Schleife anstatt am Anfang ausgewertet. Dies führt dazu, dass die Schleifen mindestens einmal ausgeführt werden.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

Zum

Ein weiterer trivialer Loop-Stil. Während der Schleife wird ein Index (`i`) erhöht und Sie können ihn verwenden. Es wird normalerweise für die Handhabung von Arrays verwendet.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Für jeden

Modernisierte Methode zum Durchlaufen von `IEnumarable` Objekten. Gut, dass Sie nicht über den Index des Artikels oder die Anzahl der Elemente in der Liste nachdenken müssen.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

Foreach-Methode

Während die anderen Stile zum Auswählen oder Aktualisieren der Elemente in Auflistungen verwendet werden, wird dieser Stil normalerweise für den *sofortigen Aufruf einer Methode* für alle Elemente in einer Auflistung verwendet.

```
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);
```

Es ist wichtig zu beachten, dass diese Methode nur für `List<T>` -Instanzen und als statische Methode für `Array` verfügbar ist - sie ist **nicht** Teil von Linq.

Linq Parallel Foreach

Genau wie Linq Foreach, außer dieser erledigt die Arbeit parallel. Das bedeutet, dass alle Elemente in der Sammlung gleichzeitig die angegebene Aktion ausführen.

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

brechen

Manchmal sollte der Schleifenzustand in der Mitte der Schleife überprüft werden. Ersteres ist wohl eleganter als das Letztere:

```
for (;;)
{
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

Alternative:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
```

```
    }
}
```

Hinweis: In verschachtelten Schleifen und / oder `switch` muss mehr als nur eine einfache `break` .

Foreach-Schleife

Foreach wird jedes Objekt einer Klasse, die `IEnumerable` implementiert, `IEnumerable` (beachten Sie, dass `IEnumerable<T>` erbt). Zu diesen Objekten gehören einige eingebaute, sind jedoch nicht auf `Dictionary<TKey, TSource>` beschränkt: `List<T>` , `T[]` (Arrays eines beliebigen Typs), `Dictionary<TKey, TSource>` sowie Schnittstellen wie `IQueryable` und `ICollection` usw.

Syntax

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

Bemerkungen

1. Der Typ `ItemType` muss nicht mit dem genauen Typ der Elemente übereinstimmen, sondern muss nur vom Typ der Elemente `ItemType` werden
2. Anstelle von `ItemType` kann alternativ `var` verwendet werden, um den `IEnumerable` aus dem `IEnumerable` , indem das generische Argument der `IEnumerable` Implementierung `IEnumerable`
3. Die Anweisung kann ein Block, eine einzelne Anweisung oder sogar eine leere Anweisung (`;`) sein.
4. Wenn `enumerableObject` nicht Umsetzung `IEnumerable` , wird der Code nicht kompilieren
5. Während jeder Iteration wird das aktuelle Element in `ItemType` (auch wenn dies nicht angegeben ist, aber vom Compiler über `var`) und wenn das Element nicht umgewandelt werden kann, wird eine `InvalidCastException` ausgelöst.

Betrachten Sie dieses Beispiel:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

ist äquivalent zu:

```
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-Schleife

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Ausgabe:

0
1
2
3
4

IEnumerators können mit einer while-Schleife durchlaufen werden:

```
// 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);
}
```

Beispielausgabe:

Tokyo / Yokohama
New Yorker Metro
Sao Paulo
Seoul / Incheon

Für Schleife

A For Loop eignet sich hervorragend, um eine gewisse Zeit zu erledigen. Es ist wie eine While-Schleife, aber das Inkrement ist in der Bedingung enthalten.

Eine For-Schleife wird folgendermaßen eingerichtet:

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

Initialisierung - Erstellt eine neue lokale Variable, die nur in der Schleife verwendet werden kann.
Bedingung - Die Schleife wird nur ausgeführt, wenn die Bedingung erfüllt ist.
Inkrement - Wie sich die Variable bei jeder Ausführung der Schleife ändert.

Ein Beispiel:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Ausgabe:

0
1
2
3
4

Sie können auch Leerzeichen in der For-Schleife auslassen, aber Sie müssen alle Semikola haben, damit sie funktioniert.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Ausgabe für 3:

3
5
7
9
11

Do - While-Schleife

Sie ähnelt einer `while` Schleife, testet jedoch die Bedingung am *Ende* des Schleifenkörpers. Die Do-While-Schleife führt die Schleife einmal aus, unabhängig davon, ob die Bedingung erfüllt ist oder nicht.

```
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
```

Verschachtelte Schleifen

```
// 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);
    }
}
```

fortsetzen

Neben `break` gibt es auch das Schlüsselwort `continue` . Anstatt die Schleife vollständig zu unterbrechen, wird die aktuelle Iteration einfach übersprungen. Dies kann nützlich sein, wenn Sie nicht möchten, dass ein Code ausgeführt wird, wenn ein bestimmter Wert festgelegt ist.

Hier ist ein einfaches Beispiel:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

Wird darin enden, dass:

```
9
10
```

Hinweis: `Continue` ist während while- oder do-while-Schleifen oft am nützlichsten. For-Loops mit gut definierten Exit-Bedingungen profitieren möglicherweise weniger.

Looping online lesen: https://riptutorial.com/de/csharp/topic/2064/looping

Kapitel 98: Methoden

Examples

Eine Methode deklarieren

Jede Methode verfügt über eine eindeutige Signatur, die aus einem Zugriffsmechanismus (`public` , `private` , ...), einem optionalen Modifikator (`abstract`), einem Namen und bei Bedarf Methodenparametern besteht. Beachten Sie, dass der Rückgabetyp nicht Teil der Signatur ist. Ein Methodenprototyp sieht wie folgt aus:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` kann `public` , `protected` , `pirvate` oder standardmäßig `internal` .

`OptionalModifier` kann eine `static abstract virtual override new` oder `sealed` .

`ReturnType` kann für keine Rückgabe `void` oder es kann ein beliebiger Typ von den grundlegenden Typen sein, wie `int` bis zu komplexen Klassen.

Eine Methode kann einige oder keine Eingabeparameter haben. Um Parameter für eine Methode festzulegen, sollten Sie jede wie normale Variablendeklarationen deklarieren (wie `int a`), und für mehrere Parameter sollten Sie ein Komma zwischen ihnen verwenden (wie `int a, int b`).

Parameter können Standardwerte haben. Dazu sollten Sie einen Wert für den Parameter einstellen (wie `int a = 0`). Wenn ein Parameter einen Standardwert hat, ist die Einstellung des Eingabewerts optional.

Das folgende Methodenbeispiel gibt die Summe zweier Ganzzahlen zurück:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Methode aufrufen

Eine statische Methode aufrufen:

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Eine statische Methode aufrufen und ihren Rückgabewert speichern:

```
string input = System.Console.ReadLine();
```

Aufruf einer Instanzmethode:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Eine generische Methode aufrufen

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

Parameter und Argumente

Eine Methode kann eine beliebige Anzahl von Parametern deklarieren (in diesem Beispiel sind `i` , `s` und `o` die Parameter):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Mithilfe von Parametern können Werte an eine Methode übergeben werden, sodass die Methode damit arbeiten kann. Dies kann jede Art von Arbeit sein, z. B. das Drucken der Werte oder das Ändern von Objekten, auf die ein Parameter verweist, oder das Speichern der Werte.

Wenn Sie die Methode aufrufen, müssen Sie für jeden Parameter einen tatsächlichen Wert übergeben. An diesem Punkt werden die Werte, die Sie tatsächlich an den Methodenaufruf übergeben, als Argumente bezeichnet:

```
DoSomething(x, "hello", new object());
```

Rückgabewerte

Eine Methode kann entweder nichts (`void`) oder einen Wert eines angegebenen Typs zurückgeben:

```
// 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";
}
```

Wenn Ihre Methode einen Rückgabewert angibt, *muss* die Methode einen Wert zurückgeben. Sie machen dies mit der `return` Anweisung. Sobald eine `return` Anweisung erreicht ist, gibt sie den angegebenen Wert und den Code zurück, der nicht mehr ausgeführt wird (Ausnahmen sind `finally` Blöcke, die noch ausgeführt werden, bevor die Methode zurückkehrt).

Wenn Ihre Methode nichts (`void`) zurückgibt, können Sie die `return` Anweisung immer noch ohne Wert verwenden, wenn Sie die Methode sofort zurückgeben möchten. Am Ende einer solchen Methode wäre eine `return` Anweisung jedoch nicht erforderlich.

Beispiele für gültige `return` :

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Durch das Auslösen einer Ausnahme kann die Ausführung der Methode beendet werden, ohne dass ein Wert zurückgegeben wird. Es gibt auch Iteratorblöcke, in denen Rückgabewerte mithilfe des Schlüsselworts „yield“ generiert werden. Dies sind jedoch Sonderfälle, die an dieser Stelle nicht erläutert werden.

Standardparameter

Sie können Standardparameter verwenden, wenn Sie die Option zum Auslassen von Parametern angeben möchten:

```
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")
}
```

Wenn Sie eine solche Methode aufrufen und einen Parameter auslassen, für den ein Standardwert bereitgestellt wird, fügt der Compiler diesen Standardwert für Sie ein.

Beachten Sie, dass Parameter mit Standardwerten **nach** Parametern ohne Standardwerte geschrieben werden müssen.

```
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);
    }
```

WARNUNG : Da dies so funktioniert, können Standardwerte in einigen Fällen problematisch sein. Wenn Sie den Standardwert eines Methodenparameters ändern und nicht alle Aufrufer dieser Methode neu kompilieren, verwenden diese Anrufer weiterhin den Standardwert, der beim Kompilieren vorhanden war, was zu Inkonsistenzen führen kann.

Überladung der Methode

Definition: Wenn mehrere Methoden mit demselben Namen mit unterschiedlichen Parametern deklariert werden, spricht man von Methodenüberladung. Das Überladen von Methoden stellt normalerweise Funktionen dar, die in ihrem Zweck identisch sind, jedoch so geschrieben werden, dass sie unterschiedliche Datentypen als Parameter akzeptieren.

Beeinflussende Faktoren

- Anzahl der Argumente
- Art der Argumente
- Rückgabetyp **

Stellen Sie sich eine Methode namens `Area` , die Berechnungsfunktionen ausführt, die verschiedene Argumente akzeptiert und das Ergebnis zurückgibt.

Beispiel

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

Diese Methode akzeptiert ein Argument und gibt einen String zurück. Wenn wir die Methode mit einer Ganzzahl aufrufen (z. B. `5`), wird die Ausgabe `"Area of Square is 25"` .

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

Wenn wir dieser Methode zwei doppelte Werte übergeben, ist die Ausgabe das Produkt der beiden Werte und vom Typ double. Dies kann sowohl für die Multiplikation als auch für das Finden der Fläche von Rechtecken verwendet werden

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

Dies kann speziell zum Ermitteln des Kreisbereichs verwendet werden, der einen doppelten Wert (`radius`) akzeptiert und einen anderen doppelten Wert als seinen Bereich zurückgibt.

Jede dieser Methoden kann normalerweise ohne Konflikte aufgerufen werden. Der Compiler überprüft die Parameter jedes Methodenaufrufs, um festzustellen, welche Version von `Area`

verwendet werden muss.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

** Beachten Sie, dass der Rückgabetyp *allein* nicht zwischen zwei Methoden unterscheiden kann. Wenn wir zum Beispiel zwei Definitionen für Area hatten, die die gleichen Parameter hatten, wie folgt:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

Wenn unsere Klasse dieselben Methodennamen verwenden muss, die unterschiedliche Werte zurückgeben, können wir das Problem der Mehrdeutigkeit beseitigen, indem Sie eine Schnittstelle implementieren und deren Verwendung explizit definieren.

```
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) { ... }
```

Anonyme Methode

Anonyme Methoden bieten eine Technik zum Übergeben eines Codeblocks als Delegatparameter. Sie sind Methoden mit Körper, aber ohne Namen.

```
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));
    }
}
```

Zugangsrechte

```
// 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()
```

Methoden online lesen: https://riptutorial.com/de/csharp/topic/60/methoden

Kapitel 99: Microsoft.Exchange.WebServices

Examples

Abruf der angegebenen Abwesenheits-Einstellungen des Benutzers abrufen

Zuerst erstellen wir ein `ExchangeManager` Objekt, bei dem der Konstruktor für uns eine Verbindung zu den Diensten herstellt. Es hat auch eine `GetOofSettings` Methode, die das `OofSettings` Objekt für die angegebene E-Mail-Adresse `OofSettings` :

```
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);
        }
    }
}
```

Wir können das jetzt anderswo so nennen:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

Aktualisieren Sie die Abwesenheits-Einstellungen bestimmter Benutzer

Mit der folgenden Klasse können wir eine Verbindung zu Exchange herstellen und dann mit `UpdateUserOof` eines bestimmten Benutzers `UpdateUserOof` :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Aktualisieren Sie die Benutzereinstellungen folgendermaßen:

```
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);
```

Beachten Sie, dass Sie die Nachrichten mit Standard- `html` Tags formatieren können.

Microsoft.Exchange.WebServices online lesen:
https://riptutorial.com/de/csharp/topic/4863/microsoft-exchange-webservices

Kapitel 100: Müllsammler in .Net

Examples

Große Objekthaufenverdichtung

Standardmäßig wird der Large Object Heap nicht komprimiert, im Gegensatz zum klassischen Object Heap, der zu einer Fragmentierung des Speichers und darüber hinaus zu `OutOfMemoryException` s führen kann

Ab .NET 4.5.1 besteht die Möglichkeit , den Large Object Heap (zusammen mit einer Garbage Collection) explizit zu komprimieren:

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

Genauso wie jede explizite Garbage-Collection-Anforderung (diese wird als Anforderung bezeichnet, weil die CLR nicht dazu gezwungen ist, sie auszuführen), wird diese mit Vorsicht verwendet und standardmäßig vermieden, wenn dies möglich ist, da `GC` Statistiken deaktiviert werden können, was deren Leistung beeinträchtigt.

Schwache Referenzen

In .NET ordnet der GC Objekte zu, wenn keine Referenzen mehr vorhanden sind. Daher kann der GC dieses Objekt nicht zuweisen, obwohl ein Objekt immer noch über Code erreichbar ist (es gibt einen starken Verweis darauf). Dies kann zu einem Problem werden, wenn viele große Objekte vorhanden sind.

Eine schwache Referenz ist eine Referenz, die es dem GC ermöglicht, das Objekt zu erfassen und gleichzeitig auf das Objekt zuzugreifen. Eine schwache Referenz gilt nur während der unbestimmten Zeit, bis das Objekt erfasst wird, wenn keine starken Referenzen vorhanden sind. Wenn Sie eine schwache Referenz verwenden, kann die Anwendung immer noch eine starke Referenz auf das Objekt erhalten, wodurch verhindert wird, dass es erfasst wird. Schwache Referenzen können daher nützlich sein, um große Objekte festzuhalten, deren Initialisierung teuer ist, die aber für die Speicherbereinigung zur Verfügung stehen sollten, wenn sie nicht aktiv verwendet werden.

Einfache Verwendung:

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

GC.Collect();

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

So könnten schwache Referenzen verwendet werden, um beispielsweise einen Cache von Objekten zu verwalten. Es ist jedoch wichtig zu wissen, dass immer die Gefahr besteht, dass der Speicherbereiniger das Objekt erreicht, bevor eine starke Referenz erstellt wird.

Schwache Referenzen sind auch hilfreich, um Speicherlecks zu vermeiden. Ein typischer Anwendungsfall betrifft Ereignisse.

Angenommen, wir haben einen Handler für ein Ereignis in einer Quelle:

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

Dieser Code registriert einen Ereignishandler und erstellt eine starke Referenz von der Ereignisquelle auf das empfangende Objekt. Wenn das Quellobjekt eine längere Lebensdauer als der Listener hat und der Listener das Ereignis nicht mehr benötigt, wenn keine anderen Verweise darauf vorhanden sind, führt die Verwendung normaler .NET-Ereignisse zu einem Speicherverlust: Das Quellobjekt enthält Listener-Objekte im Speicher sollte Müll gesammelt werden.

In diesem Fall empfiehlt es sich, das Weak Event Pattern zu verwenden .

So etwas wie:

```
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);
        }
    }
```

und so verwendet:

```
 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); });
```

In diesem Fall haben wir natürlich einige Einschränkungen - die Veranstaltung muss eine sein

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

Wie MSDN vorschlägt:

- Verwenden Sie lange, schwache Referenzen nur, wenn dies erforderlich ist, da der Status des Objekts nach der Fertigstellung nicht vorhersagbar ist.
- Vermeiden Sie die Verwendung schwacher Verweise auf kleine Objekte, da der Zeiger selbst so groß oder größer sein kann.
- Vermeiden Sie die Verwendung schwacher Verweise als automatische Lösung für Probleme bei der Speicherverwaltung. Entwickeln Sie stattdessen eine effektive Caching-Richtlinie für die Handhabung der Objekte Ihrer Anwendung.

Müllsammler in .Net online lesen: https://riptutorial.com/de/csharp/topic/1287/mullsammler-in--net

Kapitel 101: Name des Betreibers

Einführung

Mit `nameof` Operator `nameof` können Sie den Namen einer **Variablen** , eines **Typs** oder eines **Members** in Zeichenfolgenform `nameof` , ohne sie als Literal hart zu codieren.

Die Operation wird zur Kompilierzeit ausgewertet. Das heißt, Sie können einen referenzierten Bezeichner mithilfe der Umbenennungsfunktion einer IDE umbenennen. Die Namenszeichenfolge wird damit aktualisiert.

Syntax

- nameof (Ausdruck)

Examples

Grundlegende Verwendung: Drucken eines Variablennamens

Mit `nameof` Operator `nameof` können Sie den Namen einer Variablen, eines Typs oder eines Members in Zeichenfolgenform `nameof` , ohne sie als Literal hart zu codieren. Die Operation wird zur Kompilierzeit ausgewertet. Das bedeutet, dass Sie mit der Umbenennungsfunktion einer IDE einen referenzierten Bezeichner umbenennen können und die Namenszeichenfolge damit aktualisiert wird.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Würde ausgeben

    myString

weil der Name der Variablen "myString" ist. Durch Umgestaltung des Variablennamens wird die Zeichenfolge geändert.

Bei Aufruf für einen Referenztyp gibt der Name des Operators den Namen der aktuellen Referenz zurück, *nicht* den Namen oder den `nameof` des zugrunde liegenden Objekts. Zum Beispiel:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Einen Parameternamen drucken

Ausschnitt

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Konsolenausgabe

paramValue

PropertyChanged-Ereignis auslösen

Ausschnitt

```
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";
```

Konsolenausgabe

Adresse

Behandlung von PropertyChanged-Ereignissen

Ausschnitt

```
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;
```

Konsolenausgabe

Titel geändert in Alles brennt und ist kaputt

Status wurde in ShowStopper geändert

Wird auf einen generischen Typparameter angewendet

Ausschnitt

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}

...

var myClass = new SomeClass<int>();
```

```
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

Konsolenausgabe

TItem

SomeClass

Auf qualifizierte Bezeichner angewendet

Ausschnitt

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Konsolenausgabe

MeinNamensraum

Meine Klasse

MyNestedClass

MyStaticProperty

Argumentprüfung und Schutzklauseln

Bevorzugen

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Über

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

Die Verwendung der Funktion `nameof` erleichtert das `nameof` Methodenparametern.

Stark typisierte MVC-Aktionslinks

Anstelle der üblichen locker getippt:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

Sie können jetzt stark typisierte Aktionslinks erstellen:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Wenn Sie nun Ihren Code umgestalten und die `UserController.LogIn` Methode in `UserController.SignIn` umbenennen möchten, müssen Sie sich nicht um die Suche nach allen Vorkommen von Zeichenfolgen kümmern. Der Compiler erledigt die Arbeit.

Name des Betreibers online lesen: https://riptutorial.com/de/csharp/topic/80/name-des-betreibers

Kapitel 102: Nicht zulässige Typen

Syntax

- `Nullable<int> i = 10;`
- int? j = 11;
- int? k = null;
- Terminzeit? DateOfBirth = DateTime.Now;
- Dezimal? Betrag = 1,0 m;
- bool IsAvailable = true;
- verkohlen? Buchstabe = 'a';
- (Art)? Variablennamen

Bemerkungen

Nullwerttypen können alle Werte eines zugrunde liegenden Typs sowie `null` .

Die Syntax `T?` ist eine Abkürzung für `Nullable<T>`

Nicht zulässige Werte sind eigentlich `System.ValueType` Objekte. Sie können daher in Box- oder Unbox-Formaten eingeschlossen werden. Außerdem ist der `null` eines nullfähigen Objekts nicht gleich dem `null` eines Referenzobjekts, sondern lediglich ein Flag.

Wenn ein Objekt auf NULL festlegbare boxing wird der Nullwert umgewandelt `null` Referenz und Nicht-Null - Wert wird auf NULL-zugrunde liegenden Typen umgewandelt.

```
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
```

Die zweite Regel führt zu korrektem, aber paradoxem Code:

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

In Kurzform:

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

Examples

Initialisierung einer nullbaren

Für `null` :

```
Nullable<int> i = null;
```

Oder:

```
int? i = null;
```

Oder:

```
var i = (int?)null;
```

Für Nicht-Nullwerte:

```
Nullable<int> i = 0;
```

Oder:

```
int? i = 0;
```

Prüfen Sie, ob ein Nullwert einen Wert hat

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Welches ist das Gleiche wie:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Rufen Sie den Wert eines nullfähigen Typs ab

Gegebene folgende nullable `int`

```
int? i = 10;
```

Falls ein Standardwert erforderlich ist, können Sie einen mit dem Koaleszenzoperator " `GetValueOrDefault` Methode " `GetValueOrDefault` oder vor der Zuweisung prüfen, ob "nullable" in `HasValue` .

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

Die folgende Verwendung ist immer *unsicher* . Wenn `i` zur Laufzeit null ist, wird eine `System.InvalidOperationException` ausgelöst. Wenn zur Entwurfszeit kein Wert festgelegt ist `Use of unassigned local variable 'i'` Fehler " `Use of unassigned local variable 'i'` angezeigt.

```
int j = i.Value;
```

Abrufen eines Standardwerts aus einer NULL-Eigenschaft

Die `.GetValueOrDefault()` Methode gibt auch dann einen Wert zurück, wenn die `.HasValue` Eigenschaft false ist (im Gegensatz zur Value-Eigenschaft, die eine Ausnahme `.HasValue`).

```
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
    }
}
```

Ausgabe:

```
0
1
```

Prüfen Sie, ob es sich bei einem generischen Typparameter um einen nullfähigen Typ handelt

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

Der Standardwert für nullfähige Typen ist null

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Ausgabe:

Null

Effektive Nutzung des zugrunde liegenden Nullable Streit

Jeder nullfähige Typ ist ein **generischer** Typ. Und jede Nullable Type ist ein **Werttyp.**

Es gibt einige Tricks, die es ermöglichen, das Ergebnis der Nullable.GetUnderlyingType- Methode **effektiv zu verwenden** , wenn Sie Code erstellen, der mit Reflektions- / Codegenerierungszwecken zusammenhängt:

```
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)));
    }
}
```

Die Verwendung:

```
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.");
```

Ausgabe:

```
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. Der `NullableTypesCache` ist wie folgt definiert:

```
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));
    }
}
```

Nicht zulässige Typen online lesen: https://riptutorial.com/de/csharp/topic/1240/nicht-zulassige-typen

Kapitel 103: Nullbedingte Operatoren

Syntax

- X? .Y; // null, wenn X null ist, sonst XY
- X & le; Y & le; Z; // Null, wenn X Null oder Y Null ist, sonst XYZ
- X [Index]; // Null, wenn X Null ist, sonst X [Index]
- X? .ValueMethod (); // null wenn X gleich null ist sonst das Ergebnis von X.ValueMethod ();
- X.VoidMethod (); // nichts tun, wenn X null ist, sonst Aufruf X.VoidMethod ();

Bemerkungen

Wenn Sie den `Nullable<T>` einen `Nullable<T>` `T` Sie einen `Nullable<T>` zurück.

Examples

Nullbedingter Operator

Die `?.` Operator ist syntaktischer Zucker, um ausführliche Nullprüfungen zu vermeiden. Es ist auch als der sichere Navigationsoperator bekannt .

Klasse, die im folgenden Beispiel verwendet wird:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

Wenn ein Objekt möglicherweise null ist (z. B. eine Funktion, die einen Referenztyp zurückgibt), muss das Objekt zuerst auf null überprüft werden, um eine mögliche `NullReferenceException` zu verhindern. Ohne den nullbedingten Operator würde dies folgendermaßen aussehen:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

Dasselbe Beispiel mit dem null-bedingten Operator:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

Verkettung des Bedieners

Der nullbedingte Operator kann für die Member und Submitglieder eines Objekts kombiniert werden.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

Kombination mit dem Nullkoaleszenzoperator

Der nullbedingte Operator kann mit dem nullkoaleszierenden Operator kombiniert werden , um einen Standardwert bereitzustellen:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

Der null-bedingte Index

Ähnlich wie das `?.` operator, überprüft der nullbedingte Indexoperator bei der Indizierung in eine Sammlung, die möglicherweise null ist, auf Nullwerte

```
string item = collection?[index];
```

ist syntaktischer Zucker für

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

NullReferenceExceptions vermeiden

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

Dieser Effekt kann miteinander verkettet werden:

```
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;
```

Der bedingungslose Operator kann mit der Erweiterungsmethode verwendet werden

Die Erweiterungsmethode kann mit Nullreferenzen arbeiten , Sie können jedoch `?.` Verwenden `?.` Null-Check trotzdem.

```
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;
    }
}
```

Normalerweise wird die Methode für `null` ausgelöst und gibt -1 zurück:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Verwenden von `?.` Die Methode wird nicht für `null` ausgelöst und der Typ ist `int?` :

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

Dieses Verhalten wird eigentlich von der Art und Weise erwartet, in der das `?.` Der Operator funktioniert: Es werden keine Instanzmethodenaufrufe für `NullReferenceExceptions` , um `NullReferenceExceptions` zu vermeiden. Die gleiche Logik gilt jedoch für die Erweiterungsmethode, auch wenn die Methode unterschiedlich ist.

Weitere Informationen dazu, warum die Erweiterungsmethode im ersten Beispiel aufgerufen wird, finden Sie in der Dokumentation Erweiterungsmethoden - Nullprüfung .

Nullbedingte Operatoren online lesen: https://riptutorial.com/de/csharp/topic/41/nullbedingte-operatoren

Kapitel 104: Nullkoaleszenzoperator

Syntax

- var result = possibleNullObject ?? Standardwert;

Parameter

Parameter	Einzelheiten
`possibleNullObject`	Der Wert, der auf Nullwert getestet werden soll. Wenn nicht Null, wird dieser Wert zurückgegeben. Muss ein nullfähiger Typ sein.
`defaultValue`	Der zurückgegebene Wert, wenn `possibleNullObject` ist NULL. Muss den gleichen Typ wie `possibleNullObject` .

Bemerkungen

Der nullkoaleszierende Operator selbst besteht aus zwei aufeinander folgenden Fragezeichenzeichen: `??`

Es ist eine Abkürzung für den bedingten Ausdruck:

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

Der linke Operand (Objekt, das getestet wird) muss ein Wertetyp oder einen Referenztyp sein, der einen Wert annehmen kann. Andernfalls tritt ein Kompilierungsfehler auf.

Das ?? Der Operator arbeitet sowohl für Referenztypen als auch für Werttypen.

Examples

Grundlegende Verwendung

Wenn Sie den `null-coalescing operator (??)` können Sie einen Standardwert für einen nullfähigen Typ angeben, wenn der linke Operand `null` .

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Live-Demo zu .NET-Geige

Das ist logisch gleichbedeutend mit:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

oder mit dem ternären Operator (? :) Operator:

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Nullfall und Verkettung

Der linke Operand muss nullbar sein, während der rechte Operand möglicherweise ist oder nicht. Das Ergebnis wird entsprechend eingegeben.

Nicht nullfähig

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Ausgabe:

> Geben Sie Folgendes ein: System.Int32
> Wert: 3

Demo anzeigen

Nullwert

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output` wird vom Typ `int?` und gleich `b` oder `null` .

Mehrfache Koaleszenz

Koaleszenz kann auch in Ketten erfolgen:

```
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}");
```

Ausgabe:

> Geben Sie Folgendes ein: System.Int32
> Wert: 3

Demo anzeigen

Null bedingte Verkettung

Der Nullkoaleszenzoperator kann zusammen mit dem Nullausbreitungsoperator verwendet werden , um einen sichereren Zugriff auf die Eigenschaften von Objekten zu ermöglichen.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

Ausgabe:

> Geben Sie Folgendes ein: System.String
> value: Standardwert

Demo anzeigen

Nullkoaleszenzoperator mit Methodenaufrufen

Mit dem Nullkoaleszenzoperator kann leicht sichergestellt werden, dass eine Methode, die möglicherweise `null` zurückgibt, auf einen Standardwert zurückgesetzt wird.

Ohne den Nullkoaleszenzoperator:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

Mit dem Nullkoaleszenzoperator:

```
string name = GetName() ?? "Unknown!";
```

Vorhandene verwenden oder neu erstellen

Ein häufig verwendetes Szenario, bei dem dieses Feature wirklich hilfreich ist, ist die Suche nach einem Objekt in einer Sammlung und das Erstellen eines neuen Objekts, wenn es noch nicht vorhanden ist.

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Lazy Properties-Initialisierung mit Null-Koaleszenzoperator

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

Beim ersten Zugriff auf die Eigenschaft `.FooBars` wird die `_fooBars` Variable als `null` ausgewertet, sodass die Zuweisungsanweisung den `_fooBars` `.FooBars` `_fooBars` und auswertet.

Fadensicherheit

Dies ist **keine fadensichere** Methode zum Implementieren von Lazy-Eigenschaften. Verwenden Sie für Thread-sichere Faulheit die in .NET Framework integrierte Klasse `Lazy<T>` .

C # 6 Syntaktischer Zucker unter Verwendung von Expressionskörpern

Beachten Sie, dass diese Syntax seit C # 6 mit dem Ausdruck body für die Eigenschaft vereinfacht werden kann:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Nachfolgende Zugriffe auf die Eigenschaft ergeben den in der Variablen `_fooBars` gespeicherten Wert.

Beispiel im MVVM-Muster

Dies wird häufig beim Implementieren von Befehlen im MVVM-Muster verwendet. Anstatt die Befehle eifrig mit der Konstruktion eines Viewmodels zu initialisieren, werden die Befehle mit diesem Muster wie folgt faul initialisiert:

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Nullkoaleszenzoperator online lesen:

https://riptutorial.com/de/csharp/topic/37/nullkoaleszenzoperator

Kapitel 105: NullReferenceException

Examples

NullReferenceException erklärt

Wenn Sie versuchen, auf ein nicht statisches Member (Eigenschaft, Methode, Feld oder Ereignis) eines Referenzobjekts `NullReferenceException` wird eine `NullReferenceException` ausgelöst, die jedoch null ist.

```
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.
```

Um eine solche Ausnahme zu debuggen, ist das ganz einfach: In der Zeile, in der die Ausnahme ausgelöst wird, müssen Sie nur vor jedem 'suchen `.` "oder" `[` "oder in seltenen Fällen" `(` ".

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

Woher kommt meine Ausnahme? Entweder:

- `myGarage` ist `null`
- `myGarage.CarCollection` ist `null`
- `currentIndex` ist `null`
- `myGarage.CarCollection[currentIndex.Value]` ist `null`
- `theCarInTheStreet` ist `null`

Im Debug-Modus müssen Sie nur den Mauszeiger auf jedes dieser Elemente setzen, und Sie finden Ihre Nullreferenz. Dann müssen Sie verstehen, warum es keinen Wert hat. Die Korrektur hängt vollständig vom Ziel Ihrer Methode ab.

Haben Sie vergessen, es zu instanziieren / zu initialisieren?

```
myGarage.CarCollection = new Car[10];
```

Sollst du etwas anderes machen, wenn das Objekt null ist?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

Oder vielleicht hat dir jemand ein Nullargument gegeben und sollte nicht:

```
if (theCarInTheStreet == null)
```

```
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

Denken Sie in jedem Fall daran, dass eine Methode niemals eine NullReferenceException auslösen sollte. Wenn ja, haben Sie vergessen, etwas zu überprüfen.

NullReferenceException online lesen:
https://riptutorial.com/de/csharp/topic/2702/nullreferenceexception

Kapitel 106: O (n) Algorithmus für die Kreisrotation eines Arrays

Einführung

Auf meinem Weg zum Programmieren gab es einfache, aber interessante Probleme, die als Übungen gelöst werden konnten. Eines dieser Probleme bestand darin, ein Array (oder eine andere Sammlung) um einen bestimmten Wert zu drehen. Hier möchte ich Ihnen eine einfache Formel mitteilen.

Examples

Beispiel für eine generische Methode, mit der ein Array um eine bestimmte Schicht gedreht wird

Ich möchte darauf hinweisen, dass wir uns nach links drehen, wenn der Verschiebungswert negativ ist, und wir nach rechts drehen, wenn der Wert positiv ist.

```
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;
    }
```

Wichtig in diesem Code ist die Formel, mit der wir nach der Drehung den neuen Indexwert finden.

(index + array.Length + shiftCount% array.Length)% array.Length

Hier ein wenig mehr Informationen dazu:

(shiftCount% array.Length) -> Wir normalisieren den Verschiebungswert so, dass er in der Länge des Arrays liegt (da in einem Array mit der Länge 10 die Verschiebung von 1 oder 11 dasselbe ist, gilt dasselbe für -1 und -11) .

array.Length + (shiftCount% array.Length) -> Dies geschieht aufgrund von **Linksrotationen** , um sicherzustellen, dass wir nicht in einen negativen Index gehen, sondern bis zum Ende des Arrays drehen. Ohne ein Array mit der Länge 10 für Index 0 und einer Rotation -1 würden wir in eine negative Zahl (-1) gehen und nicht den realen Rotationsindexwert erhalten, der 9 ist. (10 + (-1% 10) = 9)

index + array.Length + (shiftCount% array.Length) -> nicht viel zu sagen, da wir die Drehung auf den Index anwenden, um den neuen Index zu erhalten. (0 + 10 + (-1% 10) = 9)

index + array.Length + (shiftCount% array.Length)% array.Length -> Die zweite Normalisierung stellt sicher, dass der neue Indexwert nicht außerhalb des Arrays liegt, sondern den Wert am Anfang des Arrays dreht. Es ist für Rechtsdrehungen, da in einem Array mit der Länge 10 ohne Index 9 und einer Rotation 1 der Index 10 außerhalb des Arrays aufgerufen würde und der Indexwert der realen Rotation nicht 0 ist ((9) + 10 + (1% 10)) 10 = 0)

O (n) Algorithmus für die Kreisrotation eines Arrays online lesen:
https://riptutorial.com/de/csharp/topic/9770/o--n--algorithmus-fur-die-kreisrotation-eines-arrays

Kapitel 107: Objektinitialisierer

Syntax

- SomeClass sc = new SomeClass {Eigenschaft1 = Wert1, Eigenschaft2 = Wert2, ...};
- SomeClass sc = new SomeClass (param1, param2, ...) {Eigenschaft1 = Wert1, Eigenschaft2 = Wert2, ...}

Bemerkungen

Die Konstruktorklammern können nur weggelassen werden, wenn für den zu instanziierenden Typ ein standardmäßiger (parameterloser) Konstruktor verfügbar ist.

Examples

Einfache Benutzung

Objektinitialisierer sind praktisch, wenn Sie ein Objekt erstellen und gleich einige Eigenschaften festlegen müssen, die verfügbaren Konstruktoren jedoch nicht ausreichen. Sagen Sie, Sie haben eine Klasse

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

So initialisieren Sie eine neue Instanz der Klasse mit einem Initialisierer:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

Das ist äquivalent zu

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Verwendung mit anonymen Typen

Objektinitialisierer sind die einzige Möglichkeit, anonyme Typen zu initialisieren, dh vom Compiler generierte Typen.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

Aus diesem Grund werden in LINQ-Select-Abfragen häufig Objektinitialisierer verwendet, da sie auf bequeme Weise angeben können, für welche Teile eines abgefragten Objekts Sie sich interessieren.

```
var albumTitles = from a in albums
                  select new
                  {
                     Title = a.Title,
                     Artist = a.Band
                  };
```

Verwendung mit nicht standardmäßigen Konstruktoren

Sie können Objektinitialisierer mit Konstruktoren kombinieren, um Typen bei Bedarf zu initialisieren. Nehmen Sie zum Beispiel eine Klasse, die als solche definiert ist:

```
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" }
```

Dadurch wird zunächst ein `Book` mit dem Konstruktor `Book(int)` instanziiert und anschließend jede Eigenschaft im Initialisierer festgelegt. Es ist äquivalent zu:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Objektinitialisierer online lesen: https://riptutorial.com/de/csharp/topic/738/objektinitialisierer

Kapitel 108: Objektorientierte Programmierung in C

Einführung

Dieses Thema versucht uns zu erklären, wie wir Programme schreiben können, die auf dem OOP-Ansatz beruhen. Wir werden folgende Themen behandeln: Klassen, Eigenschaften, Vererbung, Polymorphie, Schnittstellen und so weiter.

Examples

Klassen:

Das Skelett der deklarierenden Klasse ist:

<>: Erforderlich

[]:Wahlweise

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited
class][,][[Interface Name 1],[Interface Name 2],...]
{
    //Your code
}
```

Machen Sie sich keine Sorgen, wenn Sie die gesamte Syntax nicht verstehen können. Wir werden uns mit all dem vertraut machen. Für das erste Beispiel sollten Sie die folgende Klasse in Betracht ziehen:

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

In dieser Klasse erstellen wir die Variable `i` mit dem Typ `int` und den standardmäßigen privaten Zugriffsmodifizierern und der Methode `getMyValue()` mit öffentlichen Zugriffsmodifizierern.

Objektorientierte Programmierung in C # online lesen:
https://riptutorial.com/de/csharp/topic/9856/objektorientierte-programmierung-in-c-sharp

Kapitel 109: ObservableCollection

Examples

ObservableCollection initialisieren

`ObservableCollection` ist eine Auflistung des Typs `T` wie `List<T>` was bedeutet, dass Objekte des Typs `T` .

Aus der Dokumentation lesen wir folgendes:

> `ObservableCollection` stellt eine dynamische Datenerfassung dar, die Benachrichtigungen bereitstellt, wenn Elemente hinzugefügt oder entfernt werden oder wenn die gesamte Liste aktualisiert wird.

Der Hauptunterschied zu anderen Sammlungen besteht darin, dass `ObservableCollection` die Schnittstellen `INotifyCollectionChanged` und `INotifyPropertyChanged` und sofort ein Benachrichtigungsereignis `INotifyPropertyChanged` , wenn ein neues Objekt hinzugefügt oder entfernt wird und die Sammlung gelöscht wird.

Dies ist besonders nützlich, um die Benutzeroberfläche und das Backend einer Anwendung zu verbinden, ohne zusätzlichen Code schreiben zu müssen, da die Benutzeroberfläche automatisch aktualisiert wird, wenn ein Objekt zu einer beobachtbaren Sammlung hinzugefügt oder daraus entfernt wird.

Der erste Schritt, um es zu benutzen, ist zu enthalten

```
using System.Collections.ObjectModel
```

Sie können entweder eine leere Instanz einer Sammlung erstellen, z. B. vom Typ `string`

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

oder eine Instanz, die mit Daten gefüllt ist

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

Denken Sie daran, dass der Index bei allen IList-Auflistungen bei 0 beginnt (IList.Item-Eigenschaft).

ObservableCollection online lesen:
https://riptutorial.com/de/csharp/topic/7351/observablecollection--t-

Kapitel 110: Operatoren

Einführung

In C # ist ein Operator ein Programmelement, das auf einen oder mehrere Operanden in einem Ausdruck oder einer Anweisung angewendet wird. Operatoren, die einen Operanden verwenden, beispielsweise den Inkrementoperator (++) oder new, werden als unäre Operatoren bezeichnet. Operatoren, die zwei Operanden verwenden, wie beispielsweise arithmetische Operatoren (+, -, *, /), werden als binäre Operatoren bezeichnet. Ein Operator, der Bedingungsoperator (? :), nimmt drei Operanden auf und ist der einzige ternäre Operator in C #.

Syntax

- public static OperandType operator operatorSymbol (OperandType operand1)
- public static OperandType operator operatorSymbol (OperandType operand1, OperandType2 operand2)

Parameter

Parameter	Einzelheiten
operatorSymbol	Überladener Operator, zB +, -, /, *
OperandType	Der Typ, der vom überladenen Operator zurückgegeben wird.
operand1	Der erste Operand, der zur Ausführung der Operation verwendet wird.
operand2	Der zweite Operand, der beim Ausführen der Operation verwendet wird, wenn binäre Operationen ausgeführt werden.
Aussagen	Optionaler Code, der erforderlich ist, um die Operation auszuführen, bevor das Ergebnis zurückgegeben wird.

Bemerkungen

Alle Operatoren sind als `static methods` definiert, sie sind nicht `virtual` und werden nicht vererbt.

Vorrang des Bedieners

Alle Operatoren haben eine bestimmte "Priorität", abhängig davon, in welche Gruppe der Operator fällt (Operatoren derselben Gruppe haben gleiche Priorität). Das heißt, einige Operatoren werden vor anderen angewendet. Was folgt, ist eine Liste von Gruppen (mit ihren jeweiligen Operatoren), sortiert nach Rang (höchste zuerst):

- **Primäroperatoren**
  - `ab` - Mitgliederzugang
  - `a?.b` - `a?.b` bedingter `a?.b` .
  - `->` - Zeiger-Dereferenzierung in Kombination mit Memberzugriff.
  - `f(x)` - Funktionsaufruf.
  - `a[x]` - Indexer.
  - `a?[x]` - Nullbedingter Indexer.
  - `x++` - Postfix-Inkrement.
  - `x--` - Postfix-Dekrement.
  - `new` - Typinstanziierung.
  - `default(T)` - Gibt den voreingestellten Standardwert des Typs `T` .
  - `typeof` - Gibt das `Type` Objekt des Operanden zurück.
  - `checked` - Aktiviert die Überprüfung des numerischen Überlaufs.
  - `unchecked` markiert - Deaktiviert die Überprüfung des numerischen Überlaufs.
  - `delegate` - Deklariert und gibt eine Delegat-Instanz zurück.
  - `sizeof` - Gibt die Größe des Operanden vom Typ in Bytes zurück.
- **Unäre Operatoren**
  - `+x` - Gibt `x` .
  - `-x` - Numerische Negation.
  - `!x` - Logische Negation.
  - `~x` - Bitweise komplementieren / deklarieren Destruktoren.
  - `++x` - Präfixinkrement.
  - `--x` - `--x` .
  - `(T)x` - Gussteil.
  - `await` - Wartet auf eine `Task` .
  - `&x` - Liefert die Adresse (Zeiger) von `x` .
  - `*x` - Zeiger-Dereferenzierung.
- **Multiplikative Operatoren**
  - `x * y` - Multiplikation.
  - `x / y` - Division.
  - `x % y` - Modul.
- **Additive Operatoren**
  - `x + y` - Addition.
  - `x - y` - Subtraktion.
- **Bitweise Verschiebungsoperatoren**
  - `x << y` - Bits nach links verschieben.
  - `x >> y` - Bits nach rechts verschieben.
- **Relational / Typprüfoperatoren**

- `x < y` - Weniger als.
- `x > y` - Größer als.
- `x <= y` - kleiner oder gleich.
- `x >= y` - Größer als oder gleich.
- `is` - Typkompatibilität.
- `as` - Typumwandlung.

- **Gleichheitsoperatoren**
  - `x == y` - Gleichheit.
  - `x != y` - Nicht gleich.

- **Logischer AND-Operator**
  - `x & y` - logisch / bitweise UND.

- **Logischer XOR-Operator**
  - `x ^ y` - Logisch / bitweise XOR.

- **Logischer ODER-Operator**
  - `x | y` - logisch / bitweise ODER.

- **Bedingter UND Operator**
  - `x && y` - Kurzschließen des logischen UND.

- **Bedingter ODER-Operator**
  - `x || y` - Kurzschließen eines logischen ODER.

- **Nullkoaleszenzoperator**
  - `x ?? y` - Gibt `x` wenn es nicht null ist. ansonsten wird `y` .

- **Bedingter Operator**
  - `x ? y : z` - Wertet `y` wenn `x` wahr ist; ansonsten wird `z` .

Verwandte Inhalte

- Nullkoaleszenzoperator
- Nullbedingter Operator
- Name des Betreibers

Examples

Überladbare Operatoren

In C # können benutzerdefinierte Typen Operatoren überladen, indem statische Memberfunktionen mithilfe des `operator` Schlüsselworts definiert werden.
Das folgende Beispiel veranschaulicht eine Implementierung des Operators `+` .

Wenn wir eine `Complex` Klasse haben, die eine komplexe Zahl darstellt:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

Und wir möchten die Option hinzufügen, um den Operator `+` für diese Klasse zu verwenden. dh:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

Wir müssen den Operator `+` für die Klasse überladen. Dies geschieht mit einer statischen Funktion und dem `operator` Schlüsselwort:

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Operatoren wie `+` , `-` , `*` , `/` können alle überladen werden. Dies schließt auch Operatoren ein, die nicht denselben Typ zurückgeben (zum Beispiel `==` und `!=` Können trotz der Rückgabe von Booleans überladen werden). Die folgende Regel, die sich auf Paare bezieht, wird hier ebenfalls durchgesetzt.

Vergleichsoperatoren müssen paarweise überladen werden (zB wenn `<` überlastet ist, `>` muss auch überlastet werden).

Eine vollständige Liste überladbarer Operatoren (sowie nicht überladbarer Operatoren und der Einschränkungen bei einigen überladbaren Operatoren) finden Sie unter MSDN - Überladbare Operatoren (C # -Programmierhandbuch) .

7,0

Überladen des `operator is` wurde mit dem Pattern-Matching-Mechanismus von C # 7.0 eingeführt. Einzelheiten finden Sie unter Pattern Matching

Ein Typ `Cartesian` ist wie folgt definiert

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

Ein überladbarer `operator is` kann zB für `Polar` definiert werden

```
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;
    }
}
```

das kann so verwendet werden

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(Das Beispiel stammt aus der Roslyn Pattern Matching-Dokumentation.)

Beziehungsoperatoren

Gleich

Prüft, ob die angegebenen Operanden (Argumente) gleich sind

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

Im Gegensatz zu Java arbeitet der Gleichheitsvergleichsoperator nativ mit Zeichenfolgen.

Der Gleichheitsvergleichsoperator arbeitet mit Operanden unterschiedlichen Typs, wenn eine implizite Umwandlung von einer zur anderen besteht. Wenn keine geeignete implizite Umwandlung vorhanden ist, können Sie eine explizite Umwandlung aufrufen oder eine Methode zum Konvertieren in einen kompatiblen Typ verwenden.

```
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.
```

Im Gegensatz zu Visual Basic.NET stimmt der Gleichheitsvergleichsoperator nicht mit dem Gleichheitszuweisungsoperator überein.

```
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.
```

Nicht zu verwechseln mit dem Zuweisungsoperator (`=`).

Bei Werttypen gibt der Operator den Wert `true` zurück `true` wenn beide Operanden den gleichen Wert haben.
Bei Referenztypen gibt der Operator '' `true` zurück `true` wenn beide Operanden in der *Referenz* (nicht im Wert) gleich sind. Eine Ausnahme ist, dass String-Objekte mit der Wertegleichheit verglichen werden.

Nicht gleich

Prüft, ob die angegebenen Operanden *nicht* gleich sind.

```
"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.
```

Dieser Operator gibt effektiv das Gegenteil von dem Gleichheitsoperator (`==`) zurück

Größer als

Prüft, ob der erste Operand größer als der zweite Operand ist.

```
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.
```

Weniger als

Prüft, ob der erste Operand kleiner als der zweite Operand ist.

```
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.
```

Größer als gleich

Prüft, ob der erste Operand größer als der zweite Operand ist.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

Weniger als gleich

Prüft, ob der erste Operand kleiner als der zweite Operand ist.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

Kurzschließen der Operatoren

Per Definition werden die booleschen Operatoren für den Kurzschluss nur dann ausgewertet, wenn der erste Operand das Gesamtergebnis des Ausdrucks nicht bestimmen kann.

Es bedeutet , dass, wenn Sie Operator && als *firstCondition* verwenden *&& secondCondition* es *secondCondition* nur zu bewerten, wenn *firstCondition* wahr ist und ofcource das Gesamtergebnis nur dann , wenn beide *firstOperand* und *secondOperand* ausgewertet , um wahr wahr sein. Dies ist in vielen Szenarien hilfreich. Stellen Sie sich zum Beispiel vor, Sie möchten prüfen, obgleich Ihre Liste mehr als drei Elemente enthält. Sie müssen jedoch auch prüfen, ob die Liste initialisiert wurde, damit sie nicht in *NullReferenceException ausgeführt wird* . Sie können dies wie folgt erreichen:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count> 3 wird nicht geprüft, bis myList! = null erfüllt ist.

Logisches UND

`&&` ist das kurzschließende Gegenstück zum Standard-Booleschen AND (`&`) - Operator.

```
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).
```

Logisches ODER

`||` ist das kurzschließende Gegenstück des standardmäßigen booleschen OR (`|`) - Operators.

```
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).
```

Verwendungsbeispiel

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

Größe von

Gibt ein `int` , das die Größe eines Typs * in Byte enthält.

```
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.
```

** Unterstützt nur bestimmte primitive Typen im sicheren Kontext.*

In einem unsicheren Kontext kann `sizeof` verwendet werden, um die Größe anderer primitiver Typen und Strukturen zurückzugeben.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Überladen von Gleichheitsoperatoren

Es reicht nicht aus, nur Gleichheitsoperatoren zu überladen. Unter verschiedenen Umständen kann Folgendes aufgerufen werden:

1. `object.Equals` und `object.GetHashCode`
2. `IEquatable<T>.Equals` (optional, um Boxen zu vermeiden)
3. `operator ==` und `operator !=` (optional, ermöglicht die Verwendung von Operatoren)

Wenn Sie `Equals` überschreiben, muss auch `GetHashCode` überschrieben werden. Bei der Implementierung von `Equals` gibt es viele Sonderfälle: Vergleiche mit Objekten eines anderen Typs, Vergleich mit sich selbst usw.

Wenn die `Equals` Methode und der Operator `==` NICHT überschrieben werden, verhalten sie sich für Klassen und Strukturen unterschiedlich. Für Klassen werden nur Verweise verglichen, und für Strukturen werden Eigenschaftswerte durch Reflektion verglichen, was sich negativ auf die Leistung auswirken kann. `==` kann nicht für den Vergleich von Strukturen verwendet werden, es sei denn, es wird überschrieben.

Im Allgemeinen müssen bei der Gleichstellungsoperation die folgenden Regeln beachtet werden:

- Darf keine *Ausnahmen werfen* .
- Reflexivität: `A` immer gleich `A` (in manchen Systemen für `NULL` Werte möglicherweise nicht zutreffend).
- Transitvity: Wenn `A` gleich `B` ist und `B` gleich `C` , dann ist `A` gleich `C`
- Wenn `A` gleich `B` , haben `A` und `B` gleiche Hash-Codes.
- Vererbungsbaum Unabhängigkeit: Wenn `B` und `C` Fälle sind `Class2` geerbt von `Class1` : `Class1.Equals(A,B)` müssen den gleichen Wert wie der Aufruf immer wieder zurückkehren `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);
    }
}
```

Klassenmitgliedsoperatoren: Mitgliederzugang

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

Klassenmitgliedsoperatoren: Kein Zugriff auf bedingte Mitglieder

```
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;
```

Klassenmitgliedsoperatoren: Funktionsaufruf

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

Klassenelementoperatoren: Aggregierte Objektindizierung

```
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].
```

Klassenmitgliedsoperatoren: Keine bedingte Indexierung

```
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
```

"Exklusiv" oder "Operator"

Der Operator für ein "exklusives oder" (kurz XOR) lautet: ^

Dieser Operator gibt true zurück, wenn einer, aber nur einer der gelieferten Booler wahr ist.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

Bitverschiebungsoperatoren

Mit den Shift-Operatoren können Programmierer eine ganze Zahl anpassen, indem sie alle Bits nach links oder rechts verschieben. Das folgende Diagramm zeigt den Einfluss der Verschiebung eines Wertes um eine Ziffer nach links.

Linksverschiebung

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

Rechte Shifttaste

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

Implizite Besetzung und explizite Besetzung Operatoren

Mit C # können benutzerdefinierte Typen die Zuweisung und das Casting mithilfe der `explicit` und `implicit` Schlüsselwörter steuern. Die Signatur der Methode hat die Form:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

Die Methode kann keine weiteren Argumente aufnehmen und kann auch keine Instanzmethode sein. Es kann jedoch auf private Mitglieder des Typs zugreifen, in dem es definiert ist.

Ein Beispiel für eine `implicit` und `explicit` Besetzung:

```
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;
    }
}
```

Erlaube die folgende Cast-Syntax:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

Die Besetzungsoperatoren können auf beide Arten arbeiten, *von* Ihrem Typ *zu* Ihrem Typ:

```
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);
    }
}
```

Schließlich ist das `as` Schlüsselwort, das in einer Typhierarchie an der Umwandlung beteiligt sein kann, in dieser Situation **nicht** gültig. Selbst nachdem Sie eine `explicit` oder `implicit` Umwandlung definiert haben, können Sie Folgendes nicht tun:

```
ColorImage cm = myBinaryImage as ColorImage;
```

Es wird ein Kompilierungsfehler generiert.

Binäre Operatoren mit Zuordnung

C # verfügt über mehrere Operatoren, die mit einem `=` -Zeichen kombiniert werden können, um das Ergebnis des Operators auszuwerten und das Ergebnis der ursprünglichen Variablen zuzuweisen.

Beispiel:

```
x += y
```

ist das gleiche wie

```
x = x + y
```

Zuweisungsoperatoren:

- `+=`

- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `^=`
- `<<=`
- `>>=`

? : Ternärer Betreiber

Gibt einen von zwei Werten zurück, abhängig vom Wert eines booleschen Ausdrucks.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Beispiel:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

Der ternäre Operator ist rechtsassoziativ, so dass zusammengesetzte ternäre Ausdrücke verwendet werden können. Dies erfolgt durch Hinzufügen zusätzlicher ternärer Gleichungen an der wahren oder falschen Position einer ternären Elterngleichung. Es sollte darauf geachtet werden, dass die Lesbarkeit gewährleistet ist, dies kann jedoch unter Umständen kurz sein.

In diesem Beispiel wertet ein ternärer Verbindung Betrieb eine `clamp` und gibt den aktuellen Wert , wenn er innerhalb des Bereichs ist, der `min` - Wert , wenn es unterhalb dem Bereich ist, oder der `max` - Wert , wenn es über dem Bereich ist.

```
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;
}
```

Ternäre Operatoren können auch verschachtelt werden, z.

```
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
```

Beim Schreiben von zusammengesetzten ternären Anweisungen werden zur Verbesserung der Lesbarkeit üblicherweise Klammern oder Einrückungen verwendet.

Die Typen *expression_if_true* und *expression_if_false* müssen identisch sein, oder es muss eine implizite Konvertierung von einer zur anderen vorliegen.

```
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`.
```

Die Typ- und Konvertierungsanforderungen gelten auch für Ihre eigenen Klassen.

```
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`.
```

Art der

Ruft ein `System.Type` Objekt für einen Typ ab.

```
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]
```

Verwenden Sie die `GetType` Methode, um den `System.Type` der aktuellen Instanz `System.Type` .

Der Operator `typeof` nimmt einen `typeof` als Parameter an, der zur Kompilierzeit angegeben wird.

```
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
```

Standardoperator

Werttyp (wobei T: struct)

Die integrierten primitiven Datentypen wie `char` , `int` und `float` sowie benutzerdefinierte Typen, die mit `struct` oder `enum` deklariert wurden Ihr Standardwert ist `new T()` :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Referenztyp (wobei T: Klasse)

Alle `class` , `interface` , Arrays oder Delegattypen. Der Standardwert ist `null` :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

Name des Betreibers

Gibt eine Zeichenfolge zurück, die den nicht qualifizierten Namen einer `variable` , eines `type` oder eines `member` .

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

Der `nameof` Operator wurde in C # 6.0 eingeführt. Es wird zur Compile-Zeit und der zurückgegebene String - Wert wird inline durch die Compiler eingefügt ausgewertet, so kann es in den meisten Fällen verwendet werden , wo der konstante String verwendet werden kann (zum Beispiel der `case` Etikett in einer `switch` Anweisung, Attribute, etc .. .). Dies kann nützlich sein, wenn Ausnahmen, Attribute, MVC-Aktionslinks usw. ausgelöst und protokolliert werden.

. (Nullbedingter Operator)

6,0

In C # 6.0 wurde der nullbedingte Operator eingeführt `?.` gibt sofort `null` wenn der Ausdruck auf der linken Seite als `null` ausgewertet wird, anstatt eine `NullReferenceException` . Wenn seine linke Seite einen Wert ungleich `null` auswertet, wird er wie ein Normalwert behandelt . Operator. Beachten Sie, dass der Rückgabetyp immer einen nullfähigen Typ ist, da er möglicherweise `null` kann. Das bedeutet, dass es für einen Struktur- oder Primitivtyp in ein `Nullable<T>` .

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

Dies ist praktisch, wenn Sie Ereignisse abfeuern. Normalerweise müssten Sie den Ereignisaufruf in eine if -Anweisung einschließen, die nach `null` und das Ereignis anschließend auslösen, wodurch die Möglichkeit einer Race-Bedingung entsteht. Mit dem Null-Bedingungsoperator kann dies auf folgende Weise behoben werden:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Erhöhung und Abnahme von Postfix und Prefix

Postfix-Inkrement `x++` fügt `1` zu `x`

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix-Dekrement `x--` subtrahiert eins

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` wird als Präfix inkrement bezeichnet. Es erhöht den Wert von x und gibt dann x zurück, während `x++` den Wert von x zurückgibt und dann erhöht

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

während

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

beide werden häufig in for-Schleife verwendet

```
for(int i = 0; i < 10; i++)
{
```

```
}
```

=> Lambda-Operator

3,0

Der Operator `=>` hat dieselbe Priorität wie der Zuweisungsoperator `=` und ist rechtsassoziativ.

Es wird verwendet, um Lambda-Ausdrücke zu deklarieren, und es wird auch häufig bei LINQ-Abfragen verwendet :

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

Bei Verwendung in LINQ-Erweiterungen oder -Abfragen kann der Typ der Objekte normalerweise übersprungen werden, da er vom Compiler abgeleitet wird:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

Die allgemeine Form des Lambda-Operators ist die folgende:

```
(input parameters) => expression
```

Die Parameter des Lambda-Ausdrucks werden vor dem Operator `=>` angegeben, und der tatsächlich auszuführende Ausdruck / Anweisung / Block befindet sich rechts vom Operator:

```
// 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);
    }
```

Dieser Operator kann verwendet werden, um Delegaten einfach zu definieren, ohne eine explizite Methode zu schreiben:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

anstatt

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Zuweisungsoperator '='

Der Zuweisungsoperator `=` setzt den Wert des linken Operanden auf den Wert des rechten Operanden und gibt diesen Wert zurück:

```
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
```

?? Nullkoaleszenzoperator

Der Null-Koaleszenz-Operator `??` wird die linke Seite zurückgeben, wenn nicht null. Wenn es null ist, wird die rechte Seite zurückgegeben.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

Die `??` Bediener können verkettet werden , welche die Entfernung von ermöglicht , `if` überprüft.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Operatoren online lesen: https://riptutorial.com/de/csharp/topic/18/operatoren

Kapitel 111: Parallele LINQ (PLINQ)

Syntax

- ParallelEnumerable.Aggregate (func)
- ParallelEnumerable.Aggregate (Seed, func)
- ParallelEnumerable.Aggregate (Seed, updateAccumulatorFunc, CombineAccumulatorsFunc, ResultSelector)
- ParallelEnumerable.Aggregate (seedFactory, updateAccumulatorFunc, CombineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.All (Prädikat)
- ParallelEnumerable.Any ()
- ParallelEnumerable.Any (Prädikat)
- ParallelEnumerable.AsEnumerable ()
- ParallelEnumerable.AsOrdered ()
- ParallelEnumerable.AsParallel ()
- ParallelEnumerable.AsSequential ()
- ParallelEnumerable.AsUnordered ()
- ParallelEnumerable.Average (Auswahl)
- ParallelEnumerable.Cast ()
- ParallelEnumerable.Concat (Sekunde)
- ParallelEnumerable.Contains (Wert)
- ParallelEnumerable.Contains (Wert, Vergleicher)
- ParallelEnumerable.Count ()
- ParallelEnumerable.Count (Prädikat)
- ParallelEnumerable.DefaultIfEmpty ()
- ParallelEnumerable.DefaultIfEmpty (defaultValue)
- ParallelEnumerable.Distinct ()
- ParallelEnumerable.Distinct (Vergleicher)
- ParallelEnumerable.ElementAt (Index)
- ParallelEnumerable.ElementAtOrDefault (Index)
- ParallelEnumerable.Empty ()
- ParallelEnumerable.Except (Sekunde)
- ParallelEnumerable.Except (zweiter Vergleicher)
- ParallelEnumerable.First ()
- ParallelEnumerable.First (Prädikat)
- ParallelEnumerable.FirstOrDefault ()
- ParallelEnumerable.FirstOrDefault (Prädikat)
- ParallelEnumerable.ForAll (Aktion)
- ParallelEnumerable.GroupBy (keySelector)
- ParallelEnumerable.GroupBy (keySelector, Vergleicher)
- ParallelEnumerable.GroupBy (keySelector, elementSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, Vergleicher)
- ParallelEnumerable.GroupBy (keySelector, resultSelector)

- ParallelEnumerable.GroupBy (keySelector, resultSelector, Vergleicher)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector, Vergleicher)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector, Vergleicher)
- ParallelEnumerable.Intersect (second)
- ParallelEnumerable.Intersect (zweiter Vergleicher)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, Vergleicher)
- ParallelEnumerable.Last ()
- ParallelEnumerable.Last (Prädikat)
- ParallelEnumerable.LastOrDefault ()
- ParallelEnumerable.LastOrDefault (Prädikat)
- ParallelEnumerable.LongCount ()
- ParallelEnumerable.LongCount (Prädikat)
- ParallelEnumerable.Max ()
- ParallelEnumerable.Max (Auswahl)
- ParallelEnumerable.Min ()
- ParallelEnumerable.Min (Selektor)
- ParallelEnumerable.OfType ()
- ParallelEnumerable.OrderBy (keySelector)
- ParallelEnumerable.OrderBy (keySelector, Vergleicher)
- ParallelEnumerable.OrderByDescending (keySelector)
- ParallelEnumerable.OrderByDescending (keySelector, Vergleicher)
- ParallelEnumerable.Range (Start, Anzahl)
- ParallelEnumerable.Repeat (Element, Anzahl)
- ParallelEnumerable.Reverse ()
- ParallelEnumerable.Select (Auswahl)
- ParallelEnumerable.SelectMany (Selektor)
- ParallelEnumerable.SelectMany (collectionSelector, resultSelector)
- ParallelEnumerable.SequenceEqual (second)
- ParallelEnumerable.SequenceEqual (second, comparer)
- ParallelEnumerable.Single ()
- ParallelEnumerable.Single (Prädikat)
- ParallelEnumerable.SingleOrDefault ()
- ParallelEnumerable.SingleOrDefault (Prädikat)
- ParallelEnumerable.Skip (Anzahl)
- ParallelEnumerable.SkipWhile (Prädikat)
- ParallelEnumerable.Sum ()
- ParallelEnumerable.Sum (Auswahl)
- ParallelEnumerable.Take (Anzahl)
- ParallelEnumerable.TakeWhile (Prädikat)
- ParallelEnumerable.ThenBy (keySelector)
- ParallelEnumerable.ThenBy (keySelector, Vergleicher)

- ParallelEnumerable.ThenByDescending (keySelector)
- ParallelEnumerable.ThenByDescending (keySelector, Vergleicher)
- ParallelEnumerable.ToArray ()
- ParallelEnumerable.ToDictionary (keySelector)
- ParallelEnumerable.ToDictionary (keySelector, Vergleicher)
- ParallelEnumerable.ToDictionary (elementSelector)
- ParallelEnumerable.ToDictionary (elementSelector, Vergleicher)
- ParallelEnumerable.ToList ()
- ParallelEnumerable.ToLookup (keySelector)
- ParallelEnumerable.ToLookup (keySelector, Vergleicher)
- ParallelEnumerable.ToLookup (keySelector, elementSelector)
- ParallelEnumerable.ToLookup (keySelector, elementSelector, Vergleicher)
- ParallelEnumerable.Union (Sekunde)
- ParallelEnumerable.Union (zweiter Vergleicher)
- ParallelEnumerable.Where (Prädikat)
- ParallelEnumerable.WithCancellation (Annullierungstoken)
- ParallelEnumerable.WithDegreeOfParallelism (GradOfParallelism)
- ParallelEnumerable.WithExecutionMode (executionMode)
- ParallelEnumerable.WithMergeOptions (mergeOptions)
- ParallelEnumerable.Zip (zweiter resultSelector)

Examples

Einfaches Beispiel

Dieses Beispiel zeigt, wie PLINQ verwendet werden kann, um die geraden Zahlen zwischen 1 und 10.000 mit mehreren Threads zu berechnen. Beachten Sie, dass die Ergebnisliste nicht bestellt wird!

```
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

Der Parallelitätsgrad ist die maximale Anzahl gleichzeitig ausgeführter Aufgaben, die zur Verarbeitung der Abfrage verwendet werden.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

AsOrdered

Dieses Beispiel zeigt, wie PLINQ verwendet werden kann, um die geraden Zahlen zwischen 1 und 10.000 mit mehreren Threads zu berechnen. Die Reihenfolge wird in der Ergebnisliste beibehalten. `AsOrdered` jedoch, dass `AsOrdered` die Leistung einer großen Anzahl von Elementen `AsOrdered` kann. `AsOrdered` wird nach Möglichkeit eine `AsOrdered` Verarbeitung bevorzugt.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

AsUnordered

Geordnete Sequenzen können die Leistung beeinträchtigen, wenn Sie mit einer großen Anzahl von Elementen arbeiten. Um dies zu mildern, ist es möglich, `AsUnordered` wenn die Sequenzreihenfolge nicht mehr erforderlich ist.

```
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();
```

Parallele LINQ (PLINQ) online lesen: https://riptutorial.com/de/csharp/topic/3569/parallele-linq--plinq-

Kapitel 112: Polymorphismus

Examples

Ein anderes Polymorphismus-Beispiel

Polymorphismus ist eine der Säulen der OOP. Poly leitet sich von einem griechischen Begriff ab, der "mehrere Formen" bedeutet.

Unten ist ein Beispiel, das Polymorphismus zeigt. Die Klasse `Vehicle` nimmt mehrere Klassen als Basisklasse an.

Die abgeleiteten Klassen `Ducati` und `Lamborghini` erben von `Vehicle` und überschreiben die `Display()` Methode der Basisklasse, um ihre eigenen `NumberOfWheels` .

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
```

```
}
```

Unten ist der Code-Ausschnitt, in dem Polymorphismus gezeigt wird. Das Objekt wird für den Basistyp `Vehicle` mit einem variablen `vehicle` in Zeile 1 erstellt. Es ruft die Basisklassenmethode `Display()` in Zeile 2 auf und zeigt die Ausgabe wie gezeigt an.

```
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
}
```

In Zeile 3 zeigt das `vehicle` auf die abgeleitete Klasse `Ducati` und ruft seine `Display()` Methode auf, die die Ausgabe wie gezeigt anzeigt. Hier kommt das polymorphe Verhalten, obwohl das Objekt `vehicle` vom Typ `Vehicle` , ruft er die abgeleitete Klasse Methode `Display()` als Typ `Ducati` überschreibt die Basisklasse `Display()` Methode, da das `vehicle` Objekt in Richtung gerichtet ist `Ducati` .

Die gleiche Erklärung gilt, wenn die `Display()` Methode des `Lamborghini` Typs `Display()` wird.

Die Ausgabe wird unten gezeigt

```
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
```

Arten von Polymorphismus

Polymorphismus bedeutet, dass eine Operation auch auf Werte anderer Typen angewendet werden kann.

Es gibt mehrere Arten von Polymorphismus:

- **Ad-hoc-Polymorphismus:**
  enthält `function overloading` . Das Ziel ist, dass eine Methode mit verschiedenen Typen verwendet werden kann, ohne generisch zu sein.
- **Parametrischer Polymorphismus:**
  ist die Verwendung von generischen Typen. Siehe Generics
- **Subtyping:**
  hat das Ziel, eine Klasse zu vererben, um eine ähnliche Funktionalität zu verallgemeinern

---

Ad-hoc-Polymorphismus

Ziel des `Ad hoc polymorphism` ist die Erstellung einer Methode, die von verschiedenen Datentypen

aufgerufen werden kann, ohne dass eine Typumwandlung im Funktionsaufruf oder in den Generics erforderlich ist. Die folgenden Methoden `sumInt(par1, par2)` können mit verschiedenen Datentypen aufgerufen werden und haben für jede Kombination von Typen eine eigene Implementierung:

```
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);
}
```

Hier ein Beispiel für einen Anruf:

```
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
}
```

Subtyping

Subtyping ist die Verwendung von Vererbung von einer Basisklasse, um ein ähnliches Verhalten zu verallgemeinern:

```
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");
    }
}
```

Beide Klassen `NormalCar` und `ElectricCar` verfügen jetzt über eine Methode zum Auftanken, jedoch über eine eigene Implementierung. Hier ist ein Beispiel:

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

Die Ausgabe war wie folgt:

> Tanken mit Benzin
> Batterie aufladen

Polymorphismus online lesen: https://riptutorial.com/de/csharp/topic/1589/polymorphismus

Kapitel 113: Präprozessor-Anweisungen

Syntax

- #define *[Symbol]* // Definiert ein Compiler-Symbol.
- #undef *[Symbol]* // Definiert ein Compilersymbol.
- #warning *[* warn *message]* // Erzeugt eine Compiler-Warnung. Nützlich mit #if.
- #error *[Fehlermeldung]* // Erzeugt einen Compiler-Fehler. Nützlich mit #if.
- #line *[Zeilennummer] (Dateiname)* // Überschreibt die Compiler-Zeilennummer (und optional den Namen der Quelldatei). Wird mit T4-Textvorlagen verwendet .
- #pragma warning [disable | restore] *[Warnungsnummern]* // Deaktiviert / stellt Compiler-Warnungen wieder her.
- #pragma Checksumme " *[Dateiname]* " " *[Guid]* " " *[Prüfsumme]* " // Überprüft den Inhalt einer Quelldatei.
- #region *[* Regionsname *]* // Definiert einen reduzierbaren Codebereich.
- #endregion // Beendet einen Codebereichsblock.
- #if *[Bedingung]* // Führt den folgenden Code aus, wenn die Bedingung erfüllt ist.
- #else // Wird nach einem #if verwendet.
- #elif *[Bedingung]* // Wird nach einem #if verwendet.
- #endif // Beendet einen Bedingungsblock, der mit #if begonnen wurde.

Bemerkungen

Präprozessoranweisungen werden normalerweise verwendet, um Quellprogramme leicht zu ändern und in verschiedenen Ausführungsumgebungen leicht zu kompilieren. Direktiven in der Quelldatei weisen den Präprozessor an, bestimmte Aktionen auszuführen. Der Präprozessor kann beispielsweise Tokens im Text ersetzen, den Inhalt anderer Dateien in die Quelldatei einfügen oder die Kompilierung eines Teils der Datei unterdrücken, indem Textabschnitte entfernt werden. Präprozessorzeilen werden vor der Makroerweiterung erkannt und ausgeführt. Wenn also ein Makro in etwas erweitert wird, das wie ein Präprozessorbefehl aussieht, wird dieser Befehl vom Präprozessor nicht erkannt.

Präprozessoranweisungen verwenden denselben Zeichensatz wie Anweisungen für Quelldateien, mit der Ausnahme, dass Escape-Sequenzen nicht unterstützt werden. Der in Präprozessoranweisungen verwendete Zeichensatz ist derselbe wie der Ausführungszeichensatz. Der Präprozessor erkennt auch negative Zeichenwerte.

Bedingte Ausdrücke

Bedingte Ausdrücke (`#if` , `#elif` usw.) unterstützen eine begrenzte Teilmenge von Booleschen Operatoren. Sie sind:

- `==` und `!=` . Diese können nur zum Testen verwendet werden, ob das Symbol wahr (definiert) oder falsch (nicht definiert) ist.
- `&&`

`, || !`
- `()`

Zum Beispiel:

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

würde einen Code kompilieren, der "OK!" Wenn `DEBUG` nicht definiert ist, wird entweder `SOME_SYMBOL` oder `SOME_OTHER_SYMBOL` definiert und `RELEASE` definiert.

Anmerkung: Diese Ersetzungen werden *zur Kompilierzeit ausgeführt* und stehen daher zur Laufzeit nicht zur Überprüfung zur Verfügung. Code, der durch Verwendung von `#if` ist nicht Teil der Compiler-Ausgabe.

Siehe auch: C # -Prozessoranweisungen bei MSDN.

Examples

Bedingte Ausdrücke

Wenn das Folgende kompiliert wird, wird ein anderer Wert zurückgegeben, abhängig davon, welche Direktiven definiert sind.

```
// 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
}
```

Bedingte Ausdrücke werden normalerweise zum Protokollieren zusätzlicher Informationen für Debugbuilds verwendet.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
```

```
    }
}
```

Generieren von Compiler-Warnungen und -Fehlern

Compiler-Warnungen können mit der Direktive `#warning` generiert werden. Fehler können ebenfalls mit der Direktive `#error` generiert werden.

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

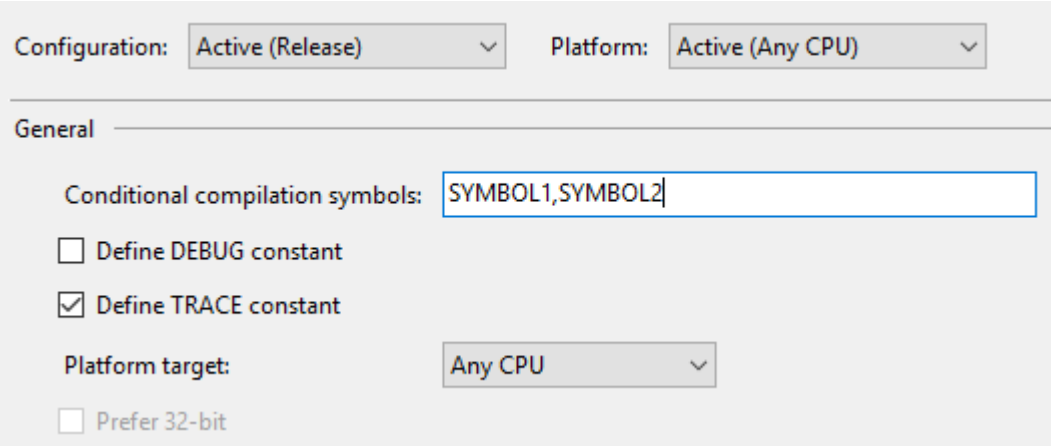
Definieren und Definieren von Symbolen

Ein Compilersymbol ist ein Schlüsselwort, das zur Kompilierzeit definiert wird und auf das überprüft werden kann, um bestimmte Codeabschnitte bedingt auszuführen.

Es gibt drei Möglichkeiten, ein Compiler-Symbol zu definieren. Sie können über Code definiert werden:

```
#define MYSYMBOL
```

Sie können in Visual Studio unter Projekteigenschaften> Erstellen> Bedingte Kompilierungssymbole definiert werden:



(Beachten Sie, dass `DEBUG` und `TRACE` eigene Kontrollkästchen haben und nicht explizit angegeben werden müssen.)

Oder sie können zur Kompilierzeit mit dem Schalter `/define:[name]` des C # -Compilers `csc.exe` .

Sie können auch undefinierte Symbole mit der Direktive `#undefine` .

Das häufigste Beispiel hierfür ist das `DEBUG` Symbol, das von Visual Studio definiert wird, wenn eine Anwendung im Debug-Modus (Versionsmodus) kompiliert wird.

```
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
    }
}
```

Wenn im obigen Beispiel ein Fehler in der Geschäftslogik der Anwendung auftritt und die Anwendung im Debug-Modus kompiliert wird (und das `DEBUG` Symbol gesetzt ist), wird der Fehler in das Ablaufverfolgungsprotokoll geschrieben und die Ausnahme wird erneut angezeigt -würfen zum Debuggen. Wenn die Anwendung jedoch im Freigabemodus kompiliert wird (und kein `DEBUG` Symbol festgelegt ist), wird ein Protokollierungsframework verwendet, um den Fehler unauffällig zu protokollieren, und dem Endbenutzer wird eine benutzerfreundliche Fehlermeldung angezeigt.

Regionsblöcke

Verwenden Sie `#region` und `#endregion` , um einen `#region #endregion` zu definieren.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

Diese Anweisungen sind nur dann von Nutzen, wenn zum Bearbeiten des Codes eine IDE verwendet wird, die reduzierbare Bereiche unterstützt (z. B. Visual Studio).

Andere Compiler-Anweisungen

Linie

`#line` steuert die Zeilennummer und den Dateinamen, die der Compiler bei der Ausgabe von Warnungen und Fehlern meldet.

```
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
}
```

Pragma Checksum

`#pragma checksum` ermöglicht die Angabe einer bestimmten Prüfsumme für eine generierte Programmdatenbank (PDB) zum Debuggen.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

Verwenden des Bedingungsattributs

Das Hinzufügen eines `Conditional` Attributs aus dem `System.Diagnostics` Namespace zu einer Methode ist eine saubere Methode, um zu steuern, welche Methoden in Ihren Builds aufgerufen werden und welche nicht.

```
#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() {...}
}
```

Deaktivieren und Wiederherstellen von Compiler-Warnungen

Sie können Compiler-Warnungen mit `#pragma warning disable` und mit `#pragma warning restore` :

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;
```

```
#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Durch Kommas getrennte Warnungsnummern sind zulässig:

```
#pragma warning disable CS0168, CS0219
```

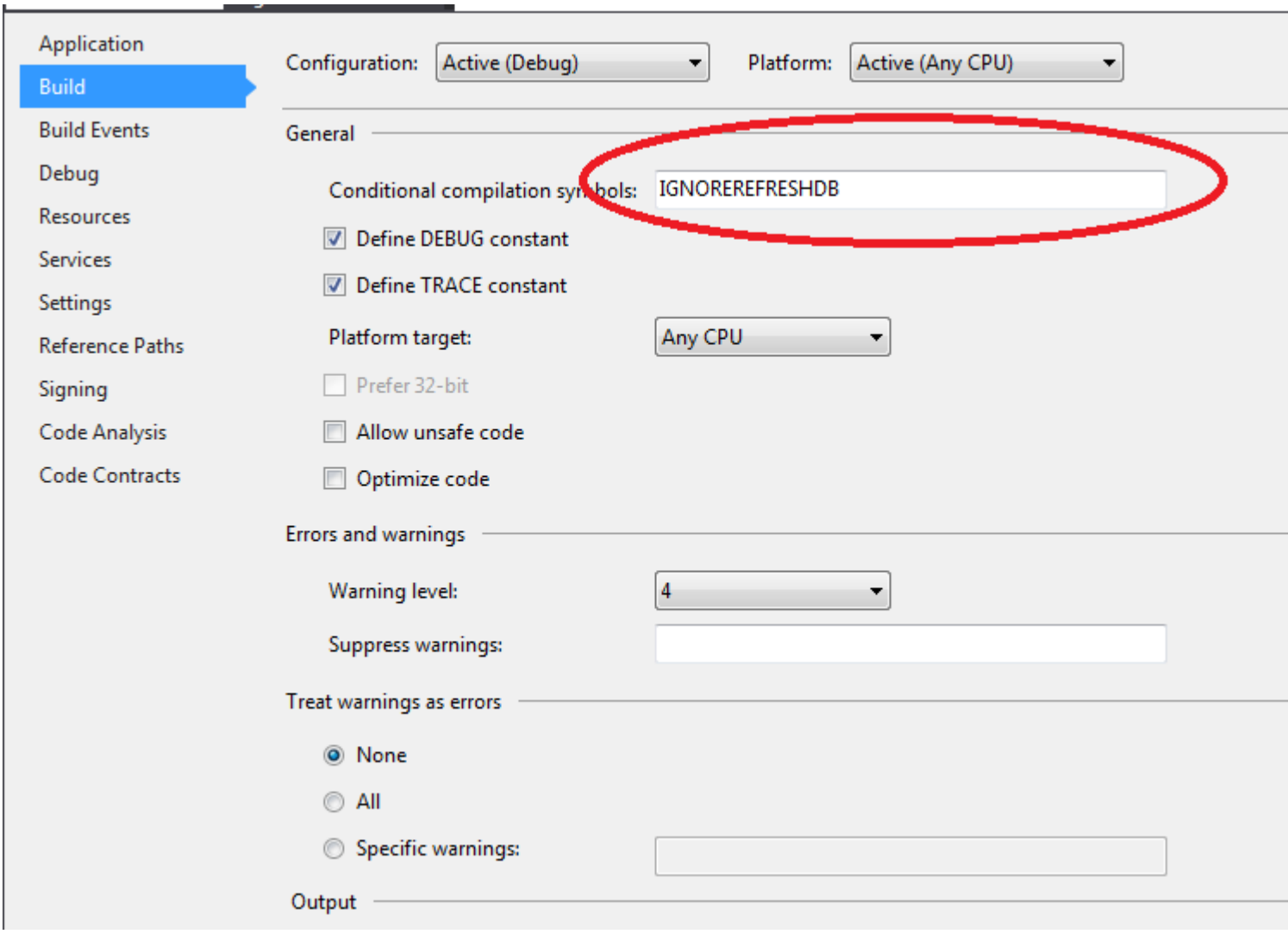
Das `CS` Präfix ist optional und kann sogar gemischt werden (dies ist jedoch keine bewährte Methode):

```
#pragma warning disable 0168, 0219, CS0414
```

Benutzerdefinierte Vorprozessoren auf Projektebene

Es ist praktisch, eine benutzerdefinierte bedingte Vorverarbeitung auf Projektebene festzulegen, wenn einige Aktionen übersprungen werden müssen, zum Beispiel für Tests.

Gehen Sie zum Projektmappen- `Solution Explorer` -> Klicken Sie mit der `rechten Maustaste` auf das Projekt, für das Sie die Variable festlegen möchten, auf -> `Properties` -> `Build` -> Suchen im Feld Feld für `Conditional compilation symbols` und geben Sie Ihre bedingte Variable hier ein



Code-Beispiel, das Code überspringt:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

Präprozessor-Anweisungen online lesen: https://riptutorial.com/de/csharp/topic/755/praprozessor-anweisungen

Kapitel 114: Reaktive Erweiterungen (Rx)

Examples

Beobachten des TextChanged-Ereignisses in einer TextBox

Ein Observable wird aus dem TextChanged-Ereignis der TextBox erstellt. Außerdem wird eine Eingabe nur ausgewählt, wenn sie sich von der letzten Eingabe unterscheidet und wenn innerhalb von 0,5 Sekunden keine Eingabe erfolgt. Die Ausgabe in diesem Beispiel wird an die Konsole gesendet.

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

Streaming von Daten aus der Datenbank mit Observable

Angenommen, eine Methode gibt `IEnumerable<T>` , z

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Erzeugt ein Observable und startet asynchron eine Methode. `SelectMany` die Sammlung und das Abonnement wird alle 200 Elemente über `Buffer SelectMany` .

```
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"));
```

Reaktive Erweiterungen (Rx) online lesen: https://riptutorial.com/de/csharp/topic/5770/reaktive-erweiterungen--rx-

Kapitel 115: Reflexion

Einführung

Reflection ist ein Mechanismus der C # -Sprache für den Zugriff auf dynamische Objekteigenschaften zur Laufzeit. Normalerweise werden über Reflection Informationen über den dynamischen Objekttyp und die Objektattributwerte abgerufen. In der REST-Anwendung kann Reflection beispielsweise verwendet werden, um durch das serialisierte Antwortobjekt zu iterieren.

Anmerkung: Gemäß den MS-Richtlinien sollte der kritische Code die Reflexion vermeiden. Siehe https://msdn.microsoft.com/de-de/library/ff647790.aspx

Bemerkungen

Mit Reflection kann der Code zur Laufzeit (Programmausführung) auf Informationen zu Baugruppen, Modulen und Typen zugreifen. Dies kann dann weiter verwendet werden, um Typen dynamisch zu erstellen, zu ändern oder auf sie zuzugreifen. Typen umfassen Eigenschaften, Methoden, Felder und Attribute.

Weiterführende Literatur:

Reflexion (C #)

Reflexion in .Net Framework

Examples

Holen Sie sich einen System.Type

Für eine Instanz eines Typs:

```
var theString = "hello";
var theType = theString.GetType();
```

Vom Typ selbst:

```
var theType = typeof(string);
```

Holen Sie sich die Mitglieder eines Typs

```
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.");
    }
  }
}
```

Ausgabe (*siehe Hinweis zur Ausgabereihenfolge weiter unten*):

```
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.
```

Wir können auch `GetMembers()` ohne `BindingFlags` . Dadurch werden *alle* öffentlichen Mitglieder dieses bestimmten Typs zurückgegeben.

Beachten Sie, dass `GetMembers` die Mitglieder nicht in einer bestimmten Reihenfolge zurückgibt. `GetMembers` Sie sich daher niemals auf die Reihenfolge, die `GetMembers` Ihnen zurückgibt.

Demo anzeigen

Holen Sie sich eine Methode und rufen Sie sie auf

Holen Sie sich die Instanzmethode und rufen Sie sie auf

```
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);
    }
}
```

Ausgabe:

Hölle

Demo anzeigen

Statische Methode abrufen und aufrufen

Wenn die Methode jedoch statisch ist, benötigen Sie keine Instanz, um sie aufzurufen.

```
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
```

Ausgabe:

7.38905609893065

Demo anzeigen

Eigenschaften abrufen und einstellen

Grundnutzung:

```
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);
```

Das Festlegen von schreibgeschützten, automatisch implementierten Eigenschaften kann über das Sicherungsfeld erfolgen (in .NET Framework heißt der Name des Sicherungsfelds "k__BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

Benutzerdefinierte Attribute

Suchen Sie nach Eigenschaften mit einem benutzerdefinierten Attribut - `MyAttribute`

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Finden Sie alle benutzerdefinierten Attribute einer bestimmten Eigenschaft

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Zählen Sie alle Klassen mit dem benutzerdefinierten Attribut `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
    }
}
```

Liest den Wert eines benutzerdefinierten Attributs zur Laufzeit

```
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);
        }
    }
```

Verwendungszweck

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
```

```
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Durchlaufen aller Eigenschaften einer Klasse

```
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));
}
```

Generische Argumente für Instanzen generischer Typen ermitteln

Wenn Sie über eine Instanz eines generischen Typs verfügen, den bestimmten Typ jedoch nicht kennen, möchten Sie möglicherweise die generischen Argumente ermitteln, die zum Erstellen dieser Instanz verwendet wurden.

Angenommen, jemand hat eine Instanz von `List<T>` und an eine Methode übergeben:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

wo `ShowGenericArguments` diese Signatur hat:

```
public void ShowGenericArguments(object o)
```

So haben Sie zum Zeitpunkt der Kompilierung keine Ahnung, mit welchen generischen Argumenten `o` . Reflection bietet eine Vielzahl von Methoden zur Überprüfung generischer Typen. Zuerst können wir feststellen, ob der Typ von `o` ein generischer Typ ist:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` gibt `true` zurück `true` wenn es sich um einen generischen Typ handelt, `Type.IsGenericType` `false` .

Aber das ist nicht alles, was wir wissen wollen. `List<>` selbst ist ebenfalls ein generischer Typ. Wir möchten jedoch nur Instanzen bestimmter *konstruierter generischer* Typen untersuchen. Ein konstruierter generischer Typ ist zum Beispiel eine `List<int>` , die einen bestimmten Typ *Argument* für alle generischen *Parameter* hat.

Die `Type` Klasse stellt zwei weitere Eigenschaften `IsConstructedGenericType` , `IsConstructedGenericType` und `IsGenericTypeDefinition` , um diese `IsGenericTypeDefinition` Typen von generischen Typdefinitionen zu unterscheiden:

```
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
```

Um die generischen Argumente einer Instanz `GetGenericArguments()` , können wir die `GetGenericArguments()` Methode verwenden, die ein `Type` Array zurückgibt, das die generischen `GetGenericArguments()` enthält:

```
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);
}
```

Der Aufruf von oben (`ShowGenericArguments(myList)`) führt also zu dieser Ausgabe:

```
Int32
```

Holen Sie sich eine generische Methode und rufen Sie sie auf

Nehmen wir an, Sie haben Klassen mit generischen Methoden. Und Sie müssen seine Funktionen mit Reflektion aufrufen.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Nehmen wir an, wir möchten GenericMethod mit dem Typ string aufrufen.

```
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
```

Für die statische Methode benötigen Sie keine Instanz. Daher ist das erste Argument auch null.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Erstellen Sie eine Instanz eines generischen Typs und rufen Sie die Methode auf

```
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[] { });
```

Instanziieren von Klassen, die eine Schnittstelle implementieren (z. B. Plugin-Aktivierung)

Wenn Sie möchten, dass Ihre Anwendung ein Plug-In-System unterstützt, z. B. zum Laden von Plug-Ins aus Assemblys im `plugins` Ordner:

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

Diese Klasse befindet sich in einer separaten DLL

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Der Plugin-Loader Ihrer Anwendung würde die DLL-Dateien finden, alle Typen in den Assemblys `IPlugin` , die `IPlugin` implementieren, und Instanzen davon erstellen.

```
    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>();
    }
```

Eine Instanz eines Typs erstellen

Der einfachste Weg ist die Verwendung der `Activator` Klasse.

Obwohl die Leistung von `Activator` seit .NET 3.5 verbessert wurde, ist die Verwendung von `Activator.CreateInstance()` wegen (relativ) geringer Leistung manchmal eine schlechte Option: Test 1 , Test 2 , Test 3 ...

Mit `Activator` Klasse

```
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
```

Sie können ein Objektarray an `Activator.CreateInstance` wenn Sie mehr als einen Parameter haben.

```
// 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);
```

Für einen generischen Typ

Die `MakeGenericType` Methode `MakeGenericType` einen offenen generischen Typ (wie `List<>`) in einen konkreten Typ (wie `List<string>`) um, indem `MakeGenericType` werden.

```
// 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);
```

Die `List<>` -Syntax ist außerhalb eines `typeof` Ausdrucks nicht zulässig.

Ohne `Activator` Klasse

Verwenden eines `new` Schlüsselworts (funktioniert für parameterlose Konstruktoren)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Invoke-Methode verwenden

```
// 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" });
```

Ausdrucksbäume verwenden

Ausdrucksbäume repräsentieren Code in einer baumartigen Datenstruktur, wobei jeder Knoten ein Ausdruck ist. Wie MSDN erklärt:

> Ausdruck ist eine Folge von einem oder mehreren Operanden und null oder mehr Operatoren, die für einen einzelnen Wert, ein Objekt, eine Methode oder einen Namespace ausgewertet werden können. Ausdrücke können aus einem Literalwert, einem Methodenaufruf, einem Operator und seinen Operanden oder einem einfachen Namen bestehen. Einfache Namen können der Name einer Variablen, eines Typmitglieds, eines Methodenparameters, eines Namespaces oder eines Typs sein.

```
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.");
        }
    }
```

Könnte so verwendet werden:

```
 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");
}
```

Verwenden von FormatterServices.GetUninitializedObject

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

Bei Verwendung von `FormatterServices.GetUninitializedObject` Konstruktoren und Feldinitialisierer nicht aufgerufen. Es ist für die Verwendung in Serialisierern und Remoting-Engines vorgesehen

Rufen Sie einen Typ mit Namen mit Namespace ab

Dazu benötigen Sie einen Verweis auf die Assembly, die den Typ enthält. Wenn Sie über einen anderen Typ verfügen, von dem Sie wissen, dass er sich in derselben Assembly befindet, wie Sie möchten, können Sie Folgendes tun:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- Dabei ist `typeName` der Name des `typeName` Typs (einschließlich des Namespaces) und `KnownType` der Typ, von dem Sie wissen, dass er sich in derselben Assembly befindet.

Weniger effizient, aber allgemeiner ist wie folgt:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Beachten Sie die Option, um das Scannen von System-Namespace-Assemblys auszuschließen, um die Suche zu beschleunigen. Wenn Ihr Typ tatsächlich ein CLR-Typ ist, müssen Sie diese beiden Zeilen löschen.

Wenn Sie den vollständig Assembly-qualifizierten Typnamen einschließlich der Assembly haben, können Sie diesen einfach mitbestellen

```
Type.GetType(fullyQualifiedName);
```

Erhalten Sie einen stark typisierten Delegierten über Reflection zu einer Methode oder Eigenschaft

Wenn Leistung ein Problem ist, ist das Aufrufen einer Methode über die Reflektion (dh über die `MethodInfo.Invoke` Methode) nicht ideal. Es ist jedoch relativ einfach, mit der Funktion `Delegate.CreateDelegate` einen leistungsfähigeren `Delegate.CreateDelegate` . Die Leistungseinbußen für die Verwendung von Reflektionen treten nur während des Delegierungserstellungsprozesses auf. Nachdem der Delegat erstellt wurde, gibt es wenig oder gar keine Leistungseinbußen, wenn er aufgerufen wird:

```
// 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));
```

Diese Technik kann auch auf Eigenschaften erweitert werden. Wenn wir eine Klasse namens `MyClass` mit einer `int` Eigenschaft namens `MyIntProperty` , `MyIntProperty` der Code zum `MyIntProperty` eines stark typisierten Getters (im folgenden Beispiel wird davon `MyIntProperty` , dass 'target' eine gültige Instanz von `MyClass`):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

... und dasselbe gilt für den Setter:

```
// 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);
```

Reflexion online lesen: https://riptutorial.com/de/csharp/topic/28/reflexion

Kapitel 116: Regeln der Namensgebung

Einführung

In diesem Thema werden einige grundlegende Namenskonventionen beschrieben, die beim Schreiben in der C # -Sprache verwendet werden. Wie alle Konventionen werden sie nicht vom Compiler durchgesetzt, sie sorgen jedoch für die Lesbarkeit zwischen Entwicklern.

Umfassende .NET Framework-Entwurfsrichtlinien finden Sie unter docs.microsoft.com/dotnet/standard/design-guidelines .

Bemerkungen

Wählen Sie leicht lesbare Bezeichnernamen

Beispielsweise ist eine Eigenschaft mit dem Namen HorizontalAlignment auf Englisch besser lesbar als AlignmentHorizontal.

Bevorzugung der Lesbarkeit gegenüber der Kürze

Der Eigenschaftsname `CanScrollHorizontally` ist besser als `ScrollableX` (eine obskure Referenz auf die X-Achse).

Vermeiden Sie die Verwendung von Unterstrichen, Bindestrichen oder anderen nicht-alphanumerischen Zeichen.

Verwenden Sie keine ungarische Schreibweise

In ungarischer Notation wird ein Präfix in Bezeichner `string strName` , um einige Metadaten über den Parameter zu kodieren, z.

Vermeiden Sie außerdem die Verwendung von Bezeichnern, die mit bereits in C # verwendeten Schlüsselwörtern in Konflikt stehen.

Abkürzungen und Akronyme

Im Allgemeinen sollten Sie keine Abkürzungen oder Akronyme verwenden. Dadurch werden Ihre Namen weniger lesbar. Ebenso ist es schwierig zu wissen, wann davon auszugehen ist, dass ein Akronym weithin bekannt ist.

Examples

Kapitalisierungskonventionen

Die folgenden Ausdrücke beschreiben verschiedene Arten von Fallbezeichnern.

Pascal Casing

Der erste Buchstabe des Bezeichners und der erste Buchstabe jedes nachfolgenden verketteten Wortes werden groß geschrieben. Sie können den Pascal-Fall für Bezeichner von drei oder mehr Zeichen verwenden. Zum Beispiel: `BackColor`

Kamelgehäuse

Der erste Buchstabe eines Bezeichners ist in Kleinbuchstaben und der erste Buchstabe jedes nachfolgenden verketteten Wortes wird großgeschrieben. Zum Beispiel: `backColor`

Großbuchstaben

Alle Buchstaben in der Kennung werden groß geschrieben. Zum Beispiel: `IO`

Regeln

Wenn ein Bezeichner aus mehreren Wörtern besteht, verwenden Sie keine Trennzeichen wie Unterstriche ("_") oder Bindestriche ("-") zwischen Wörtern. Verwenden Sie stattdessen ein Häkchen, um den Anfang jedes Wortes anzuzeigen.

In der folgenden Tabelle werden die Regeln für die Großschreibung von Bezeichnern zusammengefasst und Beispiele für die verschiedenen Arten von Bezeichnern gegeben:

Kennung	Fall	Beispiel
Lokale Variable	Kamel	carName
Klasse	Pascal	AppDomain
Aufzählungstyp	Pascal	ErrorLevel
Aufzählungswerte	Pascal	Fataler Fehler
Veranstaltung	Pascal	ValueChanged
Ausnahmeklasse	Pascal	WebException
Schreibgeschütztes statisches Feld	Pascal	Rotwert
Schnittstelle	Pascal	IDisposable
Methode	Pascal	ToString

Kennung	Fall	Beispiel
Namensraum	Pascal	System.Drawing
Parameter	Kamel	Modellname
Eigentum	Pascal	Hintergrundfarbe

Weitere Informationen finden Sie auf MSDN .

Schnittstellen

Schnittstellen sollten mit Substantiven oder Nominalphrasen oder Adjektiven benannt werden, die das Verhalten beschreiben. Beispielsweise verwendet `IComponent` ein beschreibendes Nomen, `ICustomAttributeProvider` eine Nominalphrase und `IPersistable` ein Adjektiv.

Schnittstellennamen sollte der Buchstabe `I` vorangestellt werden, um anzuzeigen, dass es sich bei dem Typ um eine Schnittstelle handelt, und es sollte der Fall Pascal verwendet werden.

Im Folgenden sind korrekt benannte Schnittstellen:

```
public interface IServiceProvider
public interface IFormatable
```

Private Felder

Es gibt zwei gängige Konventionen für private Felder: `camelCase` und `_camelCaseWithLeadingUnderscore` .

Kamel Fall

```
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;
    }
}
```

Kamelhülle mit Unterstrich

```
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;
    }
}
```

Namensräume

Das allgemeine Format für Namespaces lautet:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

Beispiele beinhalten:

```
Fabrikam.Math
Litware.Security
```

Wenn einem Namespace-Namen ein Firmenname vorangestellt wird, können Namespaces verschiedener Unternehmen nicht denselben Namen haben.

Aufzählungen

Verwenden Sie einen einzigen Namen für die meisten Enums

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

Verwenden Sie einen Plural-Namen für Aufzählungstypen, bei denen es sich um Bitfelder handelt

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Anmerkung: `FlagsAttribute` *einem* `FlagsAttribute` *des* `FlagsAttribute` *.*

Fügen Sie nicht "enum" als Suffix hinzu

```
public enum VolumeEnum // Incorrect
```

Verwenden Sie nicht den Aufzählungsnamen in jedem Eintrag

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

Ausnahmen

Fügen Sie "Ausnahme" als Suffix hinzu

Benutzerdefinierte Ausnahmename sollten mit "-Exception" versehen werden.

Im Folgenden sind korrekt benannte Ausnahmen:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Regeln der Namensgebung online lesen: https://riptutorial.com/de/csharp/topic/2330/regeln-der-namensgebung

Kapitel 117: Regex-Analyse

Syntax

- `new Regex(pattern);` *// Erzeugt eine neue Instanz mit einem definierten Muster.*
- `Regex.Match(input);` *// Startet die Suche und gibt die Übereinstimmung zurück.*
- `Regex.Matches(input);` *// Startet die Suche und gibt eine MatchCollection zurück*

Parameter

Name	Einzelheiten
Muster	Das `string` , das für die Suche verwendet werden muss. Weitere Informationen: msdn
RegexOptions *[Optional]*	Die üblichen Optionen hier sind `Singleline` und `Multiline` . Sie ändern das Verhalten von `NewLine` wie dem Punkt (.), Der eine `NewLine` (\ n) im `Multiline-Mode` aber nicht im `SingleLine-Mode` . Standardverhalten: msdn
Timeout *[optional]*	Wo Muster komplexer werden, kann die Suche mehr Zeit in Anspruch nehmen. Dies ist das verstrichene Zeitlimit für die Suche, wie es von der Netzwerkprogrammierung bekannt ist.

Bemerkungen

Benötigt mit

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

Schön zu haben

- Sie können Ihre Muster online testen, ohne Ihre Lösung zusammenstellen zu müssen, um Ergebnisse zu erhalten: Klicken Sie auf mich
- Regex101 Beispiel: Klicken Sie auf mich

Besonders Anfänger werden mit regex dazu neigen, ihre Aufgaben zu übertreffen, da sie sich mächtig anfühlen und sich für komplexere textbasierte Suchvorgänge an der richtigen Stelle befinden. Dies ist der Punkt, an dem Leute versuchen, Xml-Dokumente mit Regex zu parsen, ohne sich selbst zu fragen, ob es für diese Aufgabe eine bereits abgeschlossene Klasse wie `XmlDocument` *.*

Regex sollte die letzte Waffe sein, die sich gegen die Komplexität richtet. Vergessen Sie zumindest nicht, etwas zu unternehmen, um nach dem `right way` *zu suchen* `right way` *bevor Sie*

20 Zeilen mit Mustern aufschreiben.

Examples

Einziges Paar

```
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;
```

Ergebnis:

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

Mehrere Übereinstimmungen

```
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);
}
```

Ergebnis:

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

Regex-Analyse online lesen: https://riptutorial.com/de/csharp/topic/3774/regex-analyse

Kapitel 118: Rekursion

Bemerkungen

Beachten Sie, dass die Verwendung von Rekursion gravierenden Einfluss auf Ihren Code haben kann, da jeder rekursive Funktionsaufruf an den Stack angehängt wird. Bei zu vielen Aufrufen kann dies zu einer **StackOverflow-** Ausnahme führen. Die meisten „natürliche rekursive Funktionen“ kann als geschrieben werden `for` , `while` oder `foreach` Schleifenkonstrukt, und zwar nicht so **vornehm** oder **clever** suchen wird effizienter.

Denken Sie immer zweimal nach und verwenden Sie die Rekursion sorgfältig - wissen Sie, warum Sie sie verwenden:

- Rekursion sollte verwendet werden, wenn Sie wissen, dass die Anzahl der rekursiven Anrufe nicht zu *hoch ist*
    - *Übermäßig* bedeutet, es hängt davon ab, wie viel Speicher verfügbar ist
- Rekursion wird verwendet, weil sie klarer und sauberer ist und besser lesbar ist als eine iterative oder schleifenbasierte Funktion. Dies ist häufig der Fall, da der Code sauberer und kompakter ist (dh weniger Codezeilen).
    - Seien Sie sich jedoch bewusst, dass dies weniger effizient sein kann! Zum Beispiel wird bei der Fibonacci-Rekursion die Berechnungszeit exponentiell ansteigen, um die *n-te* Zahl in der Sequenz zu berechnen.

Wenn Sie mehr Theorie wünschen, lesen Sie bitte:

- https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

Examples

Rekursiv eine Objektstruktur beschreiben

Rekursion ist, wenn eine Methode sich selbst aufruft. Vorzugsweise tut dies so lange, bis eine bestimmte Bedingung erfüllt ist, und dann wird die Methode normal beendet, und es wird zu dem Punkt zurückgekehrt, von dem aus die Methode aufgerufen wurde. Wenn nicht, kann eine Stapelüberlaufausnahme aufgrund zu vieler rekursiver Aufrufe auftreten.

```
/// <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
    }
}
```

Rekursion im Klartext

Rekursion kann definiert werden als:

> Eine Methode, die sich selbst aufruft, bis eine bestimmte Bedingung erfüllt ist.

Ein hervorragendes und einfaches Beispiel für eine Rekursion ist eine Methode, die die Fakultät einer gegebenen Zahl erhält:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

In dieser Methode können wir sehen, dass die Methode ein Argument, `number` .

Schritt für Schritt:

In diesem Beispiel wird `Factorial(4)`

1. Ist die `number (4) == 1` ?
2. Nein? Rückkehr `4 * Factorial(number-1)` (3)
3. Da die Methode erneut aufgerufen wird, wiederholt sie nun den ersten Schritt mit `Factorial(3)` als neuem Argument.
4. Dies setzt sich fort, bis `Factorial(1)` ausgeführt wird und `number (1) == 1` zurückgibt.
5. Insgesamt "baut" die Berechnung `4 * 3 * 2 * 1` und gibt schließlich 24 zurück.

Der Schlüssel zum Verständnis der Rekursion ist, dass die Methode eine *neue Instanz* von sich selbst aufruft. Nach der Rückkehr wird die Ausführung der aufrufenden Instanz fortgesetzt.

Verwenden der Rekursion zum Abrufen der Verzeichnisstruktur

Eine der Anwendungen der Rekursion ist das Navigieren durch eine hierarchische Datenstruktur, wie z. B. eine Dateisystemverzeichnisstruktur, ohne zu wissen, wie viele Ebenen die Baumstruktur hat oder wie viele Objekte auf jeder Ebene vorhanden sind. In diesem Beispiel erfahren Sie, wie Sie Rekursion für eine Verzeichnisstruktur verwenden, um alle Unterverzeichnisse eines angegebenen Verzeichnisses zu finden und die gesamte Struktur auf der Konsole zu drucken.

```
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}");
        }
    }
}
```

Dieser Code ist etwas komplizierter als das Nötigste, um diese Aufgabe auszuführen, da er die Ausnahmebestimmung beinhaltet, um Probleme beim Abrufen der Verzeichnisse zu behandeln. Nachfolgend finden Sie eine Aufschlüsselung des Codes in kleinere Segmente mit Erläuterungen.

`Main` :

Die Hauptmethode nimmt eine Eingabe von einem Benutzer als Zeichenfolge an, die als Pfad zum Stammverzeichnis verwendet werden soll. Dann ruft sie die `PrintDirectoryTree` Methode mit dieser Zeichenfolge als Parameter auf.

`PrintDirectoryTree(string)` :

Dies ist die erste von zwei Methoden, die den tatsächlichen Verzeichnisbaumdruck verarbeiten. Diese Methode verwendet einen String, der den Pfad zum Stammverzeichnis als Parameter darstellt. Es wird geprüft, ob der Pfad ein tatsächliches Verzeichnis ist, und wenn nicht, wird eine `DirectoryNotFoundException` ausgelöst, die dann im catch-Block behandelt wird. Wenn der Pfad ein reales Verzeichnis ist, wird aus dem Pfad ein `DirectoryInfo` Objekt `rootDirectory` erstellt, und die zweite `PrintDirectoryTree` Methode wird mit dem `rootDirectory` Objekt und mit `RootLevel` `rootDirectory` `RootLevel` handelt es sich um eine Ganzzahlkonstante mit dem Wert null.

`PrintDirectoryTree(DirectoryInfo, int)` :

Diese zweite Methode übernimmt die Hauptlast der Arbeit. Als Parameter werden eine `DirectoryInfo` und eine Ganzzahl verwendet. `DirectoryInfo` ist das aktuelle Verzeichnis und die Ganzzahl ist die Tiefe des Verzeichnisses relativ zum Stamm. Um das Lesen zu erleichtern, ist die

Ausgabe für jede Ebene tief im aktuellen Verzeichnis eingerückt, sodass die Ausgabe folgendermaßen aussieht:

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

Sobald das aktuelle Verzeichnis gedruckt ist, werden seine Unterverzeichnisse abgerufen, und diese Methode wird dann für jedes Verzeichnis mit einem Tiefenwert von mehr als dem aktuellen aufgerufen. Dieser Teil ist die Rekursion: die Methode, die sich selbst aufruft. Das Programm wird auf diese Weise ausgeführt, bis jedes Verzeichnis in der Baumstruktur aufgerufen wurde. Wenn ein Verzeichnis ohne Unterverzeichnisse erreicht wurde, wird die Methode automatisch zurückgegeben.

Diese Methode fängt auch eine `UnauthorizedAccessException` , die ausgelöst wird, wenn eines der Unterverzeichnisse des aktuellen Verzeichnisses vom System geschützt wird. Die Fehlermeldung wird aus Konsistenzgründen auf der aktuellen Einrückungsebene gedruckt.

Die folgende Methode bietet einen grundlegenderen Ansatz für dieses Problem:

```
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);
    }
}
```

Dies schließt nicht die spezifische Fehlerprüfung oder Ausgabeformatierung des ersten Ansatzes ein, sondern führt tatsächlich dasselbe aus. Da es im Gegensatz zu `DirectoryInfo` nur Zeichenfolgen verwendet, kann es keinen Zugriff auf andere Verzeichniseigenschaften wie Berechtigungen gewähren.

Fibonacci-Folge

Sie können eine Zahl in der Fibonacci-Sequenz durch Rekursion berechnen.

Nach der mathematischen Theorie von F (n) = F (n-2) + F (n-1) für jedes i> 0

```
// Returns the i'th Fibonacci number
public int fib(int i) {
```

```
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

Fakultätsberechnung

Die Fakultät einer Zahl (mit!, Wie zum Beispiel 9!) Ist die Multiplikation dieser Zahl mit der Fakultät Eins. Also zum Beispiel 9! = 9 x 8! = 9 x 8 x 7! = 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1.

Also im Code wird das mit Rekursion:

```
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-Berechnung

Die Berechnung der Potenz einer gegebenen Zahl kann auch rekursiv erfolgen. Bei einer Basisnummer `n` und einem Exponenten `e` müssen wir sicherstellen, dass das Problem in Chunks aufgeteilt wird, indem der Exponent `e` verringert wird.

Theoretisches Beispiel:

- $2^2 = 2x2$
- $2^3 = 2 \cdot 2 \cdot 2$ oder $2^3 = 2^2 \cdot 2$
  Darin liegt das Geheimnis unseres rekursiven Algorithmus (siehe den Code unten). Hier geht es darum, das Problem in kleinere und einfacher zu lösende Brocken aufzulösen.
- **Anmerkungen**
  - Wenn die Basisnummer 0 ist, müssen wir uns bewusst machen, 0 als $0^3 = 0 \times 0 \times 0$ zurückzugeben
  - Wenn der Exponent 0 ist, müssen wir uns bewusst sein, immer 1 zurückzugeben, da dies eine mathematische Regel ist.

Code-Beispiel:

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,
aka: 2³ = 2 * 2² etc..
}
```

Tests in xUnit zur Überprüfung der Logik:
Obwohl dies nicht notwendig ist, sollten Sie immer Tests schreiben, um Ihre Logik zu überprüfen. Ich füge die hier im xUnit-Framework geschriebenen ein .

```
    [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 };
}
```

Rekursion online lesen: https://riptutorial.com/de/csharp/topic/2470/rekursion

Kapitel 119: Schlüsselwörter

Einführung

Schlüsselwörter sind vordefinierte, reservierte Bezeichner mit besonderer Bedeutung für den Compiler. Sie können in Ihrem Programm nicht ohne das `@` -Zeichen als Bezeichner verwendet werden. Beispielsweise ist `@if` ein gesetzlicher Bezeichner, aber nicht das Schlüsselwort `if` .

Bemerkungen

C # hat eine vordefinierte Sammlung von "Schlüsselwörtern" (oder reservierten Wörtern), die jeweils eine spezielle Funktion haben. Diese Wörter können nicht als Bezeichner (Namen für Variablen, Methoden, Klassen usw.) verwendet werden, sofern nicht das `@` vorangestellt ist.

- `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 (Direktive)
- using (Anweisung)
- virtual
- void
- volatile
- when
- while

Abgesehen davon verwendet C # auch einige Schlüsselwörter, um dem Code eine bestimmte Bedeutung zu geben. Sie werden als kontextabhängige Schlüsselwörter bezeichnet. Kontextbezogene Schlüsselwörter können als Bezeichner verwendet werden und müssen nicht mit `@` vorangestellt werden, wenn sie als Bezeichner verwendet werden.

- 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

Das Schlüsselwort `stackalloc` erstellt einen Speicherbereich auf dem Stapel und gibt einen Zeiger auf den Anfang dieses Speichers zurück. Der zugewiesene Stapelspeicher wird automatisch entfernt, wenn der Bereich, in dem er erstellt wurde, beendet wird.

```
//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;
...
```

Wird in einem unsicheren Kontext verwendet.

Wie bei allen Zeigern in C # gibt es keine Einschränkungen für Lesevorgänge und Zuweisungen. Das Lesen über die Grenzen des zugewiesenen Speichers hinaus hat unvorhersehbare Ergebnisse - es kann auf eine beliebige Stelle im Speicher zugegriffen werden oder es kann eine Zugriffsverletzung auftreten.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

Der zugewiesene Stapelspeicher wird automatisch entfernt, wenn der Bereich, in dem er erstellt wurde, beendet wird. Dies bedeutet, dass Sie den mit stackalloc erstellten Speicher niemals zurückgeben oder außerhalb der Gültigkeitsdauer des Bereichs speichern sollten.

```
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` kann nur beim Deklarieren *und* Initialisieren von Variablen verwendet werden. Folgendes ist *nicht* gültig:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Bemerkungen:

`stackalloc` sollte nur für Leistungsoptimierungen verwendet werden (entweder für die Berechnung oder für Interop). Dies liegt an der Tatsache, dass:

- Der Garbage Collector ist nicht erforderlich, da der Speicher auf dem Stack und nicht auf dem Heap reserviert wird. Der Speicher wird freigegeben, sobald die Variable den Gültigkeitsbereich verlässt
- Es ist schneller, Speicher auf dem Stapel als dem Heap zuzuweisen
- Erhöhen Sie die Wahrscheinlichkeit von Cache-Treffern auf der CPU aufgrund der Datenlokalität

flüchtig

Das Hinzufügen des `volatile` Schlüsselworts zu einem Feld zeigt dem Compiler an, dass der Feldwert von mehreren separaten Threads geändert werden kann. Der Hauptzweck des `volatile` Schlüsselworts besteht darin, Compiler-Optimierungen zu verhindern, die nur einen Single-Thread-Zugriff voraussetzen. Durch die Verwendung von `volatile` sichergestellt, dass der Wert des Felds der aktuellste verfügbare Wert ist und der Wert nicht den Zwischenspeichern unterliegt, die nichtflüchtige Werte sind.

Es empfiehlt sich, *jede Variable* , die von mehreren Threads verwendet werden kann, als `volatile` zu markieren, um unerwartetes Verhalten aufgrund von Optimierungen hinter den Kulissen zu verhindern. Betrachten Sie den folgenden Codeblock:

```
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);
    }
}
```

Im obigen Codeblock liest der Compiler die Anweisungen `x = 5` und `y = x + 10` und bestimmt, dass der Wert von `y` immer auf 15 endet. Daher optimiert er die letzte Anweisung als `y = 15` . Die Variable `x` ist jedoch tatsächlich ein `public` Feld, und der Wert von `x` kann zur Laufzeit durch einen anderen Thread geändert werden, der separat auf dieses Feld wirkt. Betrachten Sie nun diesen modifizierten Codeblock. Beachten Sie, dass das Feld `x` jetzt als `volatile` deklariert ist.

```
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);
    }
}
```

Nun sucht der Compiler für *Lese* Verwendungen des Feldes `x` und stellt sicher , dass der aktuelle Wert des Feldes immer abgerufen wird. Dadurch wird sichergestellt, dass der aktuelle Wert von `x` immer abgerufen wird, wenn mehrere Threads dieses Feld lesen und in dieses Feld schreiben.

`volatile` kann nur für Felder innerhalb von `class` oder `struct` . Folgendes ist *nicht* gültig :

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile` kann nur auf Felder folgender Typen angewendet werden:

- Referenztypen oder generische Typenparameter, die als Referenztypen bekannt sind
- primitive Typen wie `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `char` , `float` und `bool`
- Aufzählungstypen basierend auf `byte` , `sbyte` , `short` , `ushort` , `int` oder `uint`
- `IntPtr` und `UIntPtr`

Bemerkungen:

- Der `volatile` Modifikator wird normalerweise für ein Feld verwendet, auf das mehrere Threads zugreifen, ohne die Sperranweisung zum Serialisieren des Zugriffs zu verwenden.
- Das `volatile` Schlüsselwort kann auf Felder mit Referenztypen angewendet werden
- Das `volatile` Schlüsselwort kann nicht auf 64-Bit-Grundelementen auf einem 32-Bit-Plattformatom ausgeführt werden. Interlock-Vorgänge wie `Interlocked.Read` und `Interlocked.Exchange` müssen weiterhin für den sicheren Multithread-Zugriff auf diesen Plattformen verwendet werden.

Fest

Die feste Anweisung fixiert den Speicher an einem Ort. Objekte im Speicher bewegen sich normalerweise in der Nähe. Dies ermöglicht das Sammeln von Müll. Wenn wir jedoch unsichere Zeiger auf Speicheradressen verwenden, darf dieser Speicher nicht verschoben werden.

- Wir verwenden die fixed-Anweisung, um sicherzustellen, dass der Garbage Collector die Zeichenfolgendaten nicht verlagert.

Feste Variablen

```
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.
}
```

Wird in einem unsicheren Kontext verwendet.

Feste Array-Größe

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` kann nur für Felder in einer `struct` (muss auch in einem unsicheren Kontext verwendet werden).

Standard

Für Klassen, Interfaces, Delegate, Array, nullfähige (wie int?) Und `default(TheType)` gibt `default(TheType) null` :

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

Bei Strukturen und Aufzählungen gibt `default(TheType)` das gleiche wie das `new TheType()` :

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` kann besonders nützlich sein, wenn `T` ein generischer Parameter ist, für den keine Einschränkung vorhanden ist, um zu entscheiden, ob `T` ein Referenztyp oder ein Werttyp ist. Beispiel:

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

schreibgeschützt

Das Schlüsselwort `readonly` ist ein `readonly` . Wenn eine `readonly` einen `readonly` Modifizierer enthält, können Zuweisungen zu diesem Feld nur als Teil der Deklaration oder in einem Konstruktor in derselben Klasse erfolgen.

Das Schlüsselwort `readonly` unterscheidet sich vom Schlüsselwort `const` . Ein `const` Feld kann nur bei der Deklaration des Feldes initialisiert werden. Ein `readonly` Feld kann entweder bei der Deklaration oder in einem Konstruktor initialisiert werden. `readonly` Felder können daher abhängig vom verwendeten Konstruktor unterschiedliche Werte haben.

Das Schlüsselwort `readonly` wird häufig verwendet, wenn Abhängigkeiten `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
    }
}
```

> Hinweis: Das Deklarieren eines Felds als *Readonly* bedeutet keine *Unveränderlichkeit* . Wenn das Feld ein *Referenztyp* ist, kann der **Inhalt** des Objekts geändert werden. *Readonly* wird normalerweise verwendet, um zu verhindern, dass das Objekt nur während der **Instantiierung** dieses Objekts **überschrieben** und zugewiesen wird.
>
> Anmerkung: Innerhalb des Konstruktors kann ein Readonly-Feld neu zugewiesen werden

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

wie

Das `as` Schlüsselwort ist ein Operator, der einem *Cast* ähnelt. Wenn eine `InvalidCastException` nicht möglich ist, führt die Verwendung von `as` `null` und nicht zu einer `InvalidCastException` .

`expression as type` entspricht `expression is type ? (type)expression : (type)null` mit der Einschränkung, die `as` nur für Referenzkonvertierungen, nullfähige Konvertierungen und Boxkonvertierungen gültig ist. Benutzerdefinierte Konvertierungen werden *nicht* unterstützt. Stattdessen muss ein normaler Abguss verwendet werden.

Bei der obigen Erweiterung generiert der Compiler Code, sodass der `expression` nur einmal ausgewertet wird und eine dynamische Überprüfung des Typs verwendet (im Gegensatz zu den beiden im obigen Beispiel).

`as` kann nützlich sein, wenn ein Argument erwartet wird, um mehrere Typen zu vereinfachen.

Insbesondere räumt er die mehrere Optionen Benutzer - und nicht mit jeder Möglichkeit , die Überprüfung `is` vor dem Gießen, oder einfach nur Gießen und Ausnahmen zu kontrollieren. Es ist empfehlenswert, "as" beim Casting / Check eines Objekts zu verwenden, was nur eine Unboxing-Strafe verursacht. Unter Verwendung `is` zu überprüfen, dann Gießen zwei Unboxing Strafen führen.

Wenn erwartet wird, dass ein Argument eine Instanz eines bestimmten Typs ist, wird eine regelmäßige Besetzung bevorzugt, da der Zweck des Lesers für den Leser klarer ist.

Da ein Aufruf von `as null` , überprüfen Sie immer das Ergebnis, um eine `NullReferenceException` zu vermeiden.

Verwendungsbeispiel

```
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);
```

Live-Demo zu .NET-Geige

Gleichwertiges Beispiel ohne Verwendung `as` :

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

Dies ist hilfreich, wenn Sie die `Equals` Funktion in benutzerdefinierten Klassen überschreiben.

```
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
    }

}
```

ist

Überprüft, ob ein Objekt mit einem bestimmten Typ kompatibel ist, dh ob ein Objekt eine Instanz des Typs `BaseInterface` oder ein Typ ist, der von `BaseInterface` :

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

Wenn die Besetzung die Absicht hat, das Objekt zu verwenden, ist es am besten, das `as` Schlüsselwort zu verwenden. '

```
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
}
```

Ab C # 7 erweitert die `pattern matching` Funktion den Operator, um nach einem Typ zu suchen und gleichzeitig eine neue Variable zu deklarieren. Gleicher Codeteil mit C # 7:

7,0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

Art der

Gibt den `Type` eines Objekts zurück, ohne dass es instanziiert werden muss.

```
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` wird verwendet, um Werte darzustellen, **die sich** während der gesamten Lebensdauer des Programms **nicht ändern** . Ihr Wert ist ab der **Kompilierungszeit** konstant, im Gegensatz zum Schlüsselwort `readonly` , dessen Wert ab Laufzeit konstant ist.

Da sich beispielsweise die Lichtgeschwindigkeit niemals ändert, können wir sie konstant speichern.

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

Dies ist im Wesentlichen das gleiche wie die `return mass * 299792458 * 299792458` , da der Compiler direkt `c` durch seinen konstanten Wert ersetzt.

Daher kann `c` nach der Deklaration nicht mehr geändert werden. Folgendes wird einen Fehler bei der Kompilierung verursachen:

```
const double c = 299792458;  // Speed of light

c = 500;  //compile-time error
```

Einer Konstante können dieselben Zugriffsmodifizierer wie Methoden vorangestellt werden:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` Mitglieder sind von Natur aus `static` . Die Verwendung von `static` ist jedoch ausdrücklich nicht zulässig.

Sie können auch method-lokale Konstanten definieren:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

Diesen kann kein `private` oder `public` Schlüsselwort vorangestellt werden, da sie implizit lokal für die Methode sind, in der sie definiert sind.

In einer `const` Deklaration können nicht alle Typen verwendet werden. Die zulässigen Wertetypen sind die vordefinierten Typen `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` und alle `enum` . Der Versuch, `const` Member mit anderen Wertetypen (wie `TimeSpan` oder `Guid`) zu deklarieren, `TimeSpan` zur Kompilierzeit fehl.

Für den speziellen Referenztyp vordefinierte `string` , können Konstanten mit einem beliebigen Wert deklariert werden. Für alle anderen Referenztypen können Konstanten deklariert werden, müssen jedoch immer den Wert `null` .

Da `const` Werte zum Zeitpunkt der Kompilierung bekannt sind, werden sie als zulässig `case` Etikett in einer `switch` - Anweisung als Standardargumente für optionale Parameter als Argumente Spezifikationen zuzuschreiben, und so weiter.

Wenn `const` Werte in verschiedenen Baugruppen verwendet werden, muss bei der Versionsverwaltung sorgfältig vorgegangen werden. Wenn beispielsweise Assembly A eine `public const int MaxRetries = 3;` und Assembly B verwendet diese Konstante. Wenn der Wert von `MaxRetries` später in Assembly A auf `5` geändert wird (was dann erneut kompiliert wird), ist diese Änderung in Assembly B *nicht* wirksam, es *sei denn,* Assembly B wird ebenfalls neu kompiliert (mit ein Verweis auf die neue Version von A).

Aus diesem Grunde , wenn ein Wert könnte in zukünftigen Revisionen des Programms ändern, und wenn der Wert öffentlich sichtbar sein muss, nicht diesen Wert erklären `const` , es sei denn Sie wissen , dass alle abhängigen Baugruppen werden neu kompiliert werden , wenn etwas geändert wird. Die Alternative ist die Verwendung von `static readonly` anstelle von `const` , das zur Laufzeit aufgelöst wird.

Namensraum

Das `namespace` Schlüsselwort ist ein Organisationskonstrukt, das uns hilft, die Anordnung einer Codebasis zu verstehen. Namespaces in C # sind virtuelle Räume und nicht in einem physischen Ordner.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

Namensräume in C # können auch in verketteter Syntax geschrieben werden. Folgendes entspricht dem obigen:

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

versuchen, fangen, endlich werfen

`try` , `catch` , `finally` und `throw` erlauben es Ihnen, Ausnahmen in Ihrem Code zu behandeln.

```
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();
}
```

Hinweis: Das Schlüsselwort `return` kann im `try` Block verwendet werden, und der `finally` Block wird noch ausgeführt (unmittelbar vor der Rückkehr). Zum Beispiel:

```
try
```

```
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

Die Anweisung `connection.Close()` wird ausgeführt, bevor das Ergebnis von `connection.Get(query)` zurückgegeben wird.

fortsetzen

Übergeben Sie die Kontrolle sofort an die nächste Iteration des umgebenden Schleifenkonstrukts (for, for, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Ausgabe:

5
6
7
8
9

Live-Demo zu .NET-Geige

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Ausgabe:

ein
b
c
d

Live-Demo zu .NET-Geige

ref, raus

Die Schlüsselwörter `ref` und `out` bewirken, dass ein Argument als Verweis übergeben wird, nicht als Wert. Für Werttypen bedeutet dies, dass der Wert der Variablen vom Aufseher geändert werden kann.

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

Bei Referenztypen kann die Instanz in der Variablen nicht nur geändert werden (wie bei `ref`), sondern auch vollständig ersetzt werden:

```
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
```

Der Hauptunterschied zwischen dem `out` und `ref` - Schlüsselwort ist , dass `ref` die Variable erfordert vom Anrufer initialisiert wird, während `out` geht die Verantwortung an den Angerufenen.

Um einen `out` Parameter zu verwenden, müssen sowohl die Methodendefinition als auch die aufrufende Methode explizit das `out` Schlüsselwort verwenden.

```
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;
}
```

Live-Demo zu .NET-Geige

Folgendes wird *nicht* kompiliert, da für `out` Parameter ein Wert zugewiesen werden muss, bevor die Methode zurückgegeben wird (dies würde stattdessen mit `ref` kompiliert werden):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

out-Schlüsselwort als generischer Modifikator verwenden

`out` Schlüsselwort kann auch in generischen Typparametern verwendet werden, wenn generische Schnittstellen und Delegaten definiert werden. In diesem Fall gibt das Schlüsselwort `out` an, dass der Typparameter kovariant ist.

> Mit der Kovarianz können Sie einen stärker abgeleiteten Typ als den durch den generischen Parameter angegebenen verwenden. Dies ermöglicht die implizite Konvertierung von Klassen, die Variantenschnittstellen implementieren, und die implizite Konvertierung von Delegattypen. Kovarianz und Kontravarianz werden für Referenztypen unterstützt, für Werttypen jedoch nicht. - 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
```

geprüft, nicht geprüft

Die `checked` und `unchecked` Schlüsselwörter definieren, wie Operationen den mathematischen Überlauf behandeln. "Überlauf" im Kontext der `checked` und `unchecked` Schlüsselwörter liegt vor, wenn eine ganzzahlige arithmetische Operation zu einem Wert führt, dessen Größe größer ist, als der Zieldatentyp darstellen kann.

Wenn in einem `checked` Block ein Überlauf auftritt (oder wenn der Compiler so eingestellt ist, dass die überprüfte Arithmetik global verwendet wird), wird eine Ausnahme ausgelöst, um vor unerwünschtem Verhalten zu warnen. Währenddessen ist der Überlauf in einem `unchecked` Block stumm: Es werden keine Ausnahmen ausgelöst, und der Wert wird einfach an die gegenüberliegende Grenze verschoben. Dies kann zu subtilen, schwer zu findenden Fehlern führen.

Da die meisten Rechenoperationen für Werte ausgeführt werden, die nicht groß oder klein genug sind, um überzulaufen, ist es meistens nicht erforderlich, einen Block explizit als `checked` zu definieren. Bei der Arithmetik von unbegrenzten Eingaben, die zu einem Überlauf führen können, ist Vorsicht geboten, z. B. bei rekursiven Funktionen oder während der Benutzereingaben.

Weder `checked` noch `unchecked` wirken sich Gleitkomma-Rechenoperationen aus.

Wenn ein Block oder Ausdruck als `unchecked` deklariert ist, können alle darin enthaltenen Rechenoperationen überlaufen, ohne dass ein Fehler auftritt. Ein Beispiel, bei dem dieses Verhalten *gewünscht wird,* wäre die Berechnung einer Prüfsumme, bei der der Wert während der Berechnung "umlaufen" darf:

```
byte Checksum(byte[] data) {
```

```
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

Eine der häufigsten Anwendungen für `unchecked` ist das Implementieren einer benutzerdefinierten Überschreibung für `object.GetHashCode()` , eine Art von Prüfsumme. Sie sehen die Verwendung des Keywords in den Antworten auf diese Frage: Welches ist der beste Algorithmus für ein überschriebenes System.Object.GetHashCode? .

Wenn ein Block oder Ausdruck als `checked` deklariert wird, führt jede arithmetische Operation, die einen Überlauf verursacht, zur Auslösung einer `OverflowException` .

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Sowohl das Kontrollkästchen als auch das Kontrollkästchen können Block und Ausdruck sein.

Geprüfte und ungeprüfte Blöcke wirken sich nicht auf aufgerufene Methoden aus, sondern nur auf Operatoren, die in der aktuellen Methode direkt aufgerufen werden. Beispielsweise sind `Enum.ToObject()` , `Convert.ToInt32()` und benutzerdefinierte Operatoren nicht von benutzerdefinierten überprüften / ungeprüften Kontexten betroffen.

***Hinweis** : Das standardmäßige Überlauf-Standardverhalten (geprüft oder nicht markiert) kann in den **Projekteigenschaften** oder über die Befehlszeilenschalter **/ / [+ | -]** geändert werden. Üblicherweise werden standardmäßig Vorgänge für Debug-Builds und für Release-Builds nicht aktiviert. Die `checked` und `unchecked` Schlüsselwörter werden nur dann verwendet, wenn ein Standardansatz nicht angewendet wird und Sie ein explizites Verhalten benötigen, um die Korrektheit sicherzustellen.*

gehe zu

`goto` können Sie zu einer bestimmten Zeile innerhalb des Codes springen, die durch ein Label angegeben wird.

`goto` als ein:

Etikette:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
```

```
    goto sayHello;
}
```

Live-Demo zu .NET-Geige

Fallerklärung:

```
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
}
```

Live-Demo zu .NET-Geige

Dies ist besonders bei der Ausführung mehrerer Verhalten in einer switch-Anweisung hilfreich, da C # keine Fall-Through-Case-Blöcke unterstützt .

Ausnahme wiederholen

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

Live-Demo zu .NET-Geige

Ähnlich wie in vielen Sprachen wird von der Verwendung des goto-Keywords abgesehen von den unten aufgeführten Fällen abgeraten.

Gültige Verwendungen von `goto` die für C # gelten:

- Fallfall in switch-Anweisung.

- Mehrstufige Pause. LINQ kann stattdessen häufig verwendet werden, weist jedoch normalerweise eine schlechtere Leistung auf.
- Ressourcenfreigabe bei der Arbeit mit unverpackten Objekten auf niedriger Ebene. In C # sollten Low-Level-Objekte normalerweise in separaten Klassen eingeschlossen werden.
- Finite-State-Maschinen, zum Beispiel Parser; Wird vom Compiler intern verwendet und generiert async / await-Zustandsmaschinen.

enum

Das Schlüsselwort `enum` teilt dem Compiler mit, dass diese Klasse von der abstrakten Klasse `Enum` erbt, ohne dass der Programmierer sie explizit erben muss. `Enum` ist ein Nachkomme von `ValueType` , der für die Verwendung mit verschiedenen Konstanten benannter `ValueType` vorgesehen ist.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

Sie können optional einen bestimmten Wert für jeden (oder einige davon) angeben:

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

In diesem Beispiel habe ich einen Wert für 0 weggelassen. Dies ist normalerweise eine schlechte Praxis. Eine `enum` hat immer einen Standardwert, der durch explizite Konvertierung `(YourEnumType) 0` , wobei `YourEnumType` Ihnen angegebene `enume` . Wenn der Wert 0 nicht definiert ist, hat eine `enum` zu Beginn keinen definierten Wert.

Der standardmäßig zugrunde liegende Typ der `enum` ist `int` . Sie können den zugrunde liegenden Typ in einen beliebigen ganzzahligen Typ `sbyte` , einschließlich `byte` , `sbyte` , `short` , `ushort` , `int` , `uint` , `long` und `ulong` . Nachfolgend finden Sie eine Aufzählung mit dem zugrunde liegenden Typ- `byte` :

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Beachten Sie auch, dass Sie einfach mit einem Cast in den zugrunde liegenden Typ konvertieren

können:

```
int value = (int)NotableYear.EndOfWwI;
```

Aus diesen Gründen sollten Sie immer überprüfen, ob eine `enum` gültig ist, wenn Sie Bibliotheksfunktionen verfügbar machen:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

Base

Das **`base`** wird verwendet, um auf Mitglieder einer Basisklasse zuzugreifen. Sie wird häufig verwendet, um Basisimplementierungen von virtuellen Methoden aufzurufen oder um anzugeben, welcher Basiskonstruktor aufgerufen werden soll.

Einen Konstruktor auswählen

```
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);
    }
}
```

Aufruf der Basisimplementierung der virtuellen Methode

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

Es ist möglich, das Basisschlüsselwort zu verwenden, um eine Basisimplementierung von einer beliebigen Methode aufzurufen. Dadurch wird der Methodenaufruf direkt an die Basisimplementierung gebunden. Dies bedeutet, dass selbst wenn neue untergeordnete Klassen eine virtuelle Methode überschreiben, die Basisimplementierung weiterhin aufgerufen wird. Daher

muss diese mit Vorsicht verwendet werden.

```
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());
    }
}
```

für jeden

`foreach` wird verwendet, um die Elemente eines Arrays oder die Elemente in einer Auflistung zu `IEnumerable` die `IEnumerable` † implementiert.

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
```

```
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

Dies wird ausgegeben

> "Hallo Welt!"
> "Wie geht es dir heute?"
> "Auf Wiedersehen"

Live-Demo zu .NET-Geige

Sie können die `foreach` Schleife jederzeit mit dem Schlüsselwort break beenden oder mit dem Schlüsselwort continue zur nächsten Iteration übergehen.

```
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,
```

Live-Demo zu .NET-Geige

Beachten Sie, dass die Reihenfolge der Iteration *nur* für bestimmte Sammlungen wie Arrays und `List` garantiert wird, für viele andere Sammlungen jedoch **nicht** .

`IEnumerable` Während `IEnumerable` normalerweise zum `IEnumerable` aufzählbaren Sammlungen verwendet wird, erfordert `foreach` nur, dass die `object GetEnumerator()` öffentlich `object GetEnumerator()` Methode sollte ein Objekt zurückgeben, das die `bool MoveNext()` Methode `bool MoveNext()` und das `object Current { get; }` Eigenschaft.

Params

`params` kann ein Methodenparameter eine variable Anzahl von Argumenten erhalten, dh für diesen Parameter sind null, ein oder mehrere Argumente zulässig.

```
static int AddAll(params int[] numbers)
{
```

```
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

Diese Methode kann jetzt mit einer typischen Liste von `int` Argumenten oder einem Array von Ints aufgerufen werden.

```
AddAll(5, 10, 15, 20);               // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

`params` müssen höchstens einmal vorkommen. Wenn sie verwendet werden, muss sie die **letzte** in der Argumentliste sein, auch wenn der nachfolgende Typ sich vom Array unterscheidet.

Seien Sie vorsichtig beim Überladen von Funktionen, wenn Sie das Schlüsselwort `params` . C # zieht es vor, spezifischere Überladungen zu finden, bevor Sie versuchen, Überladungen mit `params` . Zum Beispiel, wenn Sie zwei Methoden haben:

```
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;
}
```

Dann hat die Überladung des spezifischen Arguments 2 Vorrang, bevor die `params` Überladung versucht wird.

```
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)
```

brechen

In einer Schleife (for, do, while) `break` die `break` Anweisung die Ausführung der innersten Schleife ab und kehrt zum Code zurück. Es kann auch mit `yield` in dem angegeben wird, dass ein Iterator beendet ist.

```
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.");
}
```

Live-Demo zu .NET-Geige

```
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);
}
```

Die break-Anweisung wird auch in Switch-Case-Konstrukten verwendet, um aus einem Case oder einem Standardsegment auszubrechen.

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

In switch-Anweisungen ist das Schlüsselwort 'break' am Ende jeder case-Anweisung erforderlich. Dies steht im Gegensatz zu einigen Sprachen, die das Durchfallen der nächsten Fallaussagen in der Serie ermöglichen. Problemumgehungen hierfür umfassen "goto" -Anweisungen oder das aufeinanderfolgende Stapeln der "case" -Anweisungen.

Der folgende Code gibt die Zahlen `0, 1, 2, ..., 9` und die letzte Zeile wird nicht ausgeführt. `yield break` bedeutet das Ende der Funktion (nicht nur eine Schleife).

```
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");
}
```

Live-Demo zu .NET-Geige

Beachten Sie, dass es im Gegensatz zu anderen Sprachen nicht möglich ist, einen bestimmten Bruch in C # zu kennzeichnen. Dies bedeutet, dass bei verschachtelten Schleifen nur die innerste Schleife angehalten wird:

```
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;
    }
}
```

Wenn Sie hier aus der *äußeren* Schleife ausbrechen möchten, können Sie verschiedene Strategien verwenden, z.

- Eine **goto-** Anweisung, um aus der gesamten Schleifenstruktur zu springen.
- Eine bestimmte Flag-Variable (`shouldBreak` im folgenden Beispiel " `shouldBreak`), die am Ende jeder Iteration der äußeren Schleife überprüft werden kann.
- Umgestaltung des Codes zur Verwendung einer `return` Anweisung im innersten Schleifenrumpf oder Vermeidung der gesamten geschachtelten Schleifenstruktur insgesamt.

```
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
```

abstrakt

Eine mit dem Keyword `abstract` gekennzeichnete Klasse kann nicht instanziiert werden.

Eine Klasse *muss* als abstrakt markiert sein, wenn sie abstrakte Elemente enthält oder wenn sie abstrakte Elemente erbt, die sie nicht implementiert. Eine Klasse *kann* als abstrakt markiert werden, auch wenn keine abstrakten Mitglieder beteiligt sind.

Abstrakte Klassen werden normalerweise als Basisklassen verwendet, wenn ein Teil der Implementierung von einer anderen Komponente angegeben werden muss.

```
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
```

Eine mit dem Schlüsselwort `abstract` gekennzeichnete Methode, Eigenschaft oder Ereignis zeigt an, dass die Implementierung für dieses Mitglied in einer Unterklasse bereitgestellt werden muss. Wie oben erwähnt, können abstrakte Member nur in abstrakten Klassen angezeigt werden.

```
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, doppelt, dezimal

schweben

`float` ist ein Alias für den .NET-Datentyp `System.Single` . Es ermöglicht die Speicherung von Gleitkommazahlen nach IEEE 754 mit einfacher Genauigkeit. Dieser Datentyp ist in der `mscorlib.dll` vorhanden, auf die jedes C # -Projekt beim Erstellen implizit verweist.

Ungefährer Bereich: -3,4 × 10^{38} bis 3,4 × 10^{38}

Dezimalgenauigkeit: 6-9 signifikante Stellen

Notation :

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> Es ist zu beachten, dass der `float` Typ häufig zu erheblichen Rundungsfehlern führt. Bei Anwendungen, bei denen Präzision wichtig ist, sollten andere Datentypen berücksichtigt werden.

---

doppelt

`double` ist ein Alias für den .NET-Datentyp `System.Double` . Es ist eine 64-Bit-Gleitkommazahl mit doppelter Genauigkeit. Dieser Datentyp ist in der `mscorlib.dll` vorhanden, auf die in einem C # -Projekt implizit verwiesen wird.

Bereich: ± 5,0 × 10^{-324} bis ± 1,7 × 10^{308}

Dezimalgenauigkeit: 15-16 signifikante Stellen

Notation :

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

Dezimal

`decimal` ist ein Alias für den .NET-Datentyp `System.Decimal` . Es stellt ein Schlüsselwort dar, das einen 128-Bit-Datentyp angibt. Im Vergleich zu Gleitkommatypen hat der Dezimaltyp eine höhere Genauigkeit und einen kleineren Bereich, wodurch er für finanzielle und monetäre Berechnungen geeignet ist. Dieser Datentyp ist in der `mscorlib.dll` vorhanden, auf die in einem C # -Projekt implizit verwiesen wird.

Bereich: -7,9 × 10^{28} bis 7,9 × 10^{28}

Dezimalgenauigkeit: 28-29 signifikante Stellen

Notation :

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

uint

Eine **vorzeichenlose Ganzzahl** oder **Uint** ist ein numerischer Datentyp, der nur positive Ganzzahlen enthalten kann. Wie der Name vermuten lässt, handelt es sich um eine vorzeichenlose 32-Bit-Ganzzahl. Das Schlüsselwort **uint** selbst ist ein Alias für den `System.UInt32` Typ `System.UInt32` . Dieser Datentyp ist in der `mscorlib.dll` , auf die jedes C # -Projekt beim Erstellen implizit verweist. Es belegt vier Byte Speicherplatz.

Vorzeichenlose Ganzzahlen können jeden Wert von 0 bis 4.294.967.295 enthalten.

Beispiele wie und jetzt nicht vorzeichenlose Ganzzahlen deklariert werden

```
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
```

Bitte beachten Sie: Laut Microsoft wird empfohlen, den **int-** Datentyp nach Möglichkeit zu verwenden, da der **Uint-** Datentyp nicht CLS-kompatibel ist.

diese

Das Schlüsselwort `this` bezieht sich auf die aktuelle Instanz von class (object). Auf diese Weise können zwei Variablen mit demselben Namen unterschieden werden, eine auf Klassenebene (ein Feld) und eine, die ein Parameter (oder eine lokale Variable) einer Methode ist.

```
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;
    }
}
```

Andere Verwendungen des Schlüsselworts sind das Verketten von nicht statischen Konstruktorüberladungen :

```
public MyClass(int arg) : this(arg, null)
{
}
```

und Indexer schreiben:

```
public string this[int idx1, string idx2]
{
    get { /* ... */ }
    set { /* ... */ }
}
```

und deklarieren von Erweiterungsmethoden :

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

Wenn es keinen Konflikt mit einer lokalen Variablen oder einem lokalen Parameter gibt, ist es eine Frage des Stils, ob `this` soll. In diesem Fall wäre also `this.MemberOfType` und `MemberOfType` gleichwertig. Siehe auch `base` .

Wenn eine Erweiterungsmethode für die aktuelle Instanz aufgerufen werden soll, ist `this` erforderlich. Wenn Sie sich beispielsweise in einer nicht statischen Methode einer Klasse befinden, die `IEnumerable<>` implementiert, und Sie die Erweiterung `Count` von vor aufrufen möchten, müssen Sie `IEnumerable<>` verwenden:

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

und `this` kann dort nicht ausgelassen werden.

zum

Syntax: `for (initializer; condition; iterator)`

- Die `for` Schleife wird häufig verwendet, wenn die Anzahl der Iterationen bekannt ist.
- Die Anweisungen im `initializer` werden nur einmal ausgeführt, bevor Sie in die Schleife gelangen.
- Der `condition` enthält einen booleschen Ausdruck, der am Ende jeder Schleifeniteration ausgewertet wird, um zu bestimmen, ob die Schleife beendet oder erneut ausgeführt werden soll.
- Der `iterator` Abschnitt definiert, was nach jeder Iteration des Schleifenkörpers passiert.

Dieses Beispiel zeigt, wie mit `for` die Zeichen einer Zeichenfolge durchlaufen werden kann:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Ausgabe:

H
e

I
I
O

Live-Demo zu .NET-Geige

Alle Ausdrücke, die eine `for` -Anweisung definieren, sind optional. Die folgende Anweisung wird beispielsweise zum Erstellen einer Endlosschleife verwendet:

```
for( ; ; )
{
    // Your code here
}
```

Der `initializer` kann mehrere Variablen enthalten, sofern sie vom gleichen Typ sind. Der `condition` kann aus einem beliebigen Ausdruck bestehen, der zu einem `bool` ausgewertet werden kann. Der `iterator` Abschnitt kann mehrere durch Kommas getrennte Aktionen ausführen:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Ausgabe:

Hallo
hallo1
hallo12

Live-Demo zu .NET-Geige

während

Der `while` Betreiber iteriert über einen Codeblock , bis die bedingte Abfrage gleich falsch oder der Code wird mit einer unterbrochen `goto` , `return` , `break` oder `throw` - Anweisung.

Syntax für `while` Schlüsselwort:

while (*Bedingung*) { *Codeblock;* }

Beispiel:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Ausgabe:

"While ist auf Loop Nummer 1."

"While ist auf Loop Nummer 2."
"While ist auf Loop Nummer 3."
"While ist auf Loop Nummer 4."
"While ist auf Loop Nummer 5."

Live-Demo zu .NET-Geige

Eine while-Schleife ist **Entry Controlled** , da die Bedingung **vor** der Ausführung des eingeschlossenen Codeblocks geprüft wird. Dies bedeutet, dass die while-Schleife ihre Anweisungen nicht ausführen würde, wenn die Bedingung falsch ist.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Wenn Sie eine `while` Bedingung angeben, ohne dass sie irgendwann falsch wird, führt dies zu einer Endlos- oder Endlosschleife. Dies sollte, soweit möglich, vermieden werden. Es können jedoch außergewöhnliche Umstände auftreten, wenn Sie dies benötigen.

Sie können eine solche Schleife wie folgt erstellen:

```
while (true)
{
//...
}
```

Beachten Sie, dass der C # -Compiler Schleifen wie transformiert

```
while (true)
{
// ...
}
```

oder

```
for(;;)
{
// ...
}
```

in

```
{
:label
// ...
goto label;
}
```

Beachten Sie, dass eine while-Schleife eine beliebige Bedingung haben kann, egal wie komplex

sie ist, solange sie einen booleschen Wert (bool) auswertet (oder zurückgibt). Es kann auch eine Funktion enthalten, die einen booleschen Wert zurückgibt (da eine solche Funktion denselben Typ wie ein Ausdruck wie "a == x" auswertet). Zum Beispiel,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

Rückkehr

> MSDN: Die return-Anweisung beendet die Ausführung der Methode, in der sie erscheint, und gibt die Kontrolle an die aufrufende Methode zurück. Es kann auch einen optionalen Wert zurückgeben. Wenn die Methode ein ungültiger Typ ist, kann die return-Anweisung weggelassen werden.

```
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
}
```

im

Das `in` Schlüsselwort hat drei Zwecke:

a) Als Teil der Syntax in einer `foreach` Anweisung oder als Teil der Syntax in einer LINQ-Abfrage

```
foreach (var member in sequence)
{
    // ...
}
```

b) Im Zusammenhang mit generischen Schnittstellen und generischen Delegattypen bedeutet dies eine *Kontravarianz* für den betreffenden Parameter Typ:

```
public interface IComparer<in T>
{
    // ...
}
```

c) Im Kontext von LINQ bezieht sich die Abfrage auf die abgefragte Sammlung

```
var query = from x in source select new { x.Name, x.ID, };
```

mit

Es gibt zwei Arten der `using` Schlüsselwörtern, `using statement` und `using directive` :

1. **mit Anweisung** :

   Das Schlüsselwort `using` sorgt dafür, dass Objekte, die die `IDisposable` Schnittstelle implementieren, nach der Verwendung ordnungsgemäß `IDisposable` werden. Es gibt ein separates Thema für die using-Anweisung

2. **unter Verwendung der Direktive**

   Die `using` Direktive hat drei Verwendungszwecke, die Using-Direktive finden Sie auf der Msdn-Seite . Es gibt ein separates Thema für die using-Direktive .

versiegelt

Bei der Anwendung auf eine Klasse verhindert der `sealed` Modifikator, dass andere Klassen von ihr erben.

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

Bei Verwendung auf eine `virtual` Methode (oder virtuelle Eigenschaft) verhindert der `sealed` Modifizierer, dass diese Methode (Eigenschaft) in abgeleiteten Klassen *überschrieben wird* .

```
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";
    }
}
```

Größe von

Wird verwendet, um die Größe in Byte für einen nicht verwalteten Typ zu ermitteln

```
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
```

statisch

Mit dem `static` Modifizierer wird ein statischer Member deklariert, der nicht instanziiert werden muss, um darauf zugegriffen zu werden, sondern wird einfach über seinen Namen, dh `DateTime.Now` , `DateTime.Now` .

`static` kann mit Klassen, Feldern, Methoden, Eigenschaften, Operatoren, Ereignissen und Konstruktoren verwendet werden.

Während eine Instanz einer Klasse eine separate Kopie aller Instanzfelder der Klasse enthält, gibt es von jedem statischen Feld nur eine Kopie.

```
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` entspricht der Gesamtzahl der Instanzen der Klasse `A`

Der statische Modifikator kann auch verwendet werden, um einen statischen Konstruktor für eine Klasse zu deklarieren, statische Daten zu initialisieren oder Code auszuführen, der nur einmal aufgerufen werden muss. Statische Konstruktoren werden aufgerufen, bevor die Klasse zum ersten Mal referenziert wird.

```
class A
{
    static public DateTime InitializationTime;
```

```
    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

Eine `static class` ist mit dem markierten `static` Schlüsselwort, und kann für eine Reihe von Methoden als vorteilhaft Behälter verwendet werden , die auf Parametern arbeiten, erfordert jedoch nicht unbedingt auf eine Instanz gebunden zu sein. Aufgrund der `static` Natur der Klasse kann sie nicht instanziiert werden, sie kann jedoch einen `static constructor` . Einige Funktionen einer `static class` umfassen:

- Kann nicht vererbt werden
- Kann nicht von etwas anderem als `Object` erben
- Kann einen statischen Konstruktor enthalten, jedoch keinen Instanzkonstruktor
- Kann nur statische Member enthalten
- Ist versiegelt

Der Compiler ist auch freundlich und teilt dem Entwickler mit, ob Instanzmitglieder in der Klasse vorhanden sind. Ein Beispiel wäre eine statische Klasse, die zwischen US-amerikanischen und kanadischen Metriken konvertiert:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

Wenn Klassen als statisch deklariert werden:

```
public static class Functions
{
  public static int Double(int value)
  {
    return value + value;
  }
}
```

Alle Funktionen, Eigenschaften oder Member innerhalb der Klasse müssen ebenfalls als statisch deklariert werden. Es kann keine Instanz der Klasse erstellt werden. Im Wesentlichen können Sie mit einer statischen Klasse Funktionspakete erstellen, die logisch zusammengefasst sind.

Seit C # 6 kann auch `static` verwendet `using` , um statische Member und Methoden zu importieren. Sie können dann ohne Klassennamen verwendet werden.

Alte Art, ohne `using static` :

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
Console
    }

}
```

Beispiel mit `using static`

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

Nachteile

Statische Klassen können zwar unglaublich nützlich sein, haben jedoch ihre eigenen Vorbehalte:

- Nach dem Aufruf der statischen Klasse wird die Klasse in den Arbeitsspeicher geladen und kann erst dann durch den Garbage Collector ausgeführt werden, wenn die AppDomain, in der sich die statische Klasse befindet, entladen wird.
- Eine statische Klasse kann keine Schnittstelle implementieren.

int

`int` ist ein Alias für `System.Int32` , bei dem es sich um einen Datentyp für 32-Bit-Ganzzahlen mit `System.Int32` handelt. Dieser Datentyp befindet sich in der `mscorlib.dll` die jedes C # -Projekt beim Erstellen implizit verweist.

Bereich: -2.147.483.648 bis 2.147.483.647

```
int int1 = -10007;
var int2 = 2132012521;
```

lange

Das **lange** Schlüsselwort wird zur Darstellung von 64-Bit-Ganzzahlen mit Vorzeichen verwendet. Es ist ein Alias für den `System.Int64` Datentyp in `mscorlib.dll` , die implizit von jedem C # Projekt verwiesen wird , wenn Sie sie erstellen.

*Jede **lange** Variable kann sowohl explizit als auch implizit deklariert werden:*

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

Eine **long-** Variable kann einen beliebigen Wert zwischen –9.223.372.036.854.775.808 und 9.223.372.036.854.775.807 enthalten und kann in Situationen nützlich sein, in denen eine Variable einen Wert enthalten muss, der die Grenzen dessen überschreitet, was andere Variablen (z. B. die **int-** Variable) halten können.

ulong

Schlüsselwort für vorzeichenlose 64-Bit-Ganzzahlen. Es stellt `System.UInt64` Datentyp gefunden `mscorlib.dll` , die implizit von jedem C # Projekt verwiesen wird , wenn Sie sie erstellen.

Bereich: 0 bis 18.446.744.073.709.551.615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 154467440636094513 15UL;
```

dynamisch

Das `dynamic` Schlüsselwort wird mit dynamisch typisierten Objekten verwendet . Als `dynamic` deklarierte Objekte verzichten auf statische Überprüfungen während der Kompilierung und werden stattdessen zur Laufzeit ausgewertet.

```
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
```

Im folgenden Beispiel wird `dynamic` mit der Bibliothek Json.NET von Newtonsoft verwendet, um Daten leicht aus einer deserialisierten JSON-Datei lesen zu können.

```
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
}
```

Mit dem dynamischen Schlüsselwort sind einige Einschränkungen verbunden. Eine davon ist die Verwendung von Erweiterungsmethoden. Im folgenden Beispiel wird eine Erweiterungsmethode für string `SayHello` : `SayHello` .

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

Der erste Ansatz besteht darin, es wie üblich (wie bei einer Zeichenfolge) aufzurufen:

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

Kein Kompilierungsfehler, aber zur Laufzeit erhalten Sie eine `RuntimeBinderException` . Um dieses Problem zu umgehen, rufen Sie die Erweiterungsmethode über die statische Klasse auf:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

virtuell, überschreiben, neu

virtuell und überschreiben

Mit dem `virtual` Schlüsselwort können eine Methode, eine Eigenschaft, ein Indexer oder ein Ereignis von abgeleiteten Klassen überschrieben werden und polymorphes Verhalten darstellen. (Mitglieder sind in C # standardmäßig nicht virtuell.)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

Um ein Element zu überschreiben, wird das `override` in den abgeleiteten Klassen verwendet. (Beachten Sie, dass die Unterschrift der Mitglieder identisch sein muss.)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
```

```
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

Das polymorphe Verhalten virtueller Member bedeutet, dass beim Aufruf das tatsächlich ausgeführte Member zur Laufzeit und nicht zur Kompilierzeit bestimmt wird. Das übergeordnete Element in der am meisten abgeleiteten Klasse, in der das betreffende Objekt eine Instanz ist, ist das ausgeführte Element.

Kurz gesagt, ein Objekt kann zur Kompilierzeit vom Typ `BaseClass` deklariert werden. Wenn es sich jedoch zur Laufzeit um eine Instanz von `DerivedClass` wird das überschriebene Member ausgeführt:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Das Überschreiben einer Methode ist optional:

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Neu

Da nur als `virtual` definierte Member überschreibbar und polymorph sind, kann eine abgeleitete Klasse, die ein nicht virtuelles Member neu definiert, zu unerwarteten Ergebnissen führen.

```
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!
```

In diesem Fall wird das ausgeführte Member immer zur Kompilierzeit basierend auf dem Objekttyp bestimmt.

- Wenn das Objekt vom Typ `BaseClass` (auch wenn zur Laufzeit eine abgeleitete Klasse ist), wird die Methode von `BaseClass` ausgeführt
- Wenn das Objekt vom Typ `DerivedClass` `DerivedClass` ist, wird die Methode von `DerivedClass` ausgeführt.

Dies ist normalerweise ein Unfall (wenn ein Mitglied zum Basistyp hinzugefügt wird, nachdem ein identischer zum abgeleiteten Typ hinzugefügt wurde) und eine Compiler-Warnung **CS0108** in diesen Szenarien generiert wird.

Wenn es beabsichtigt war, wird das `new` Schlüsselwort verwendet, um die Warnung des Compilers zu unterdrücken (und andere Entwickler über Ihre Absichten zu informieren!). Das Verhalten bleibt gleich, das `new` Schlüsselwort unterdrückt lediglich die Compiler-Warnung.

```
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!
```

Die Verwendung von Überschreiben ist *nicht* optional

Im Gegensatz zu C ++ ist die Verwendung des `override` *nicht* optional:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
```

```
    public void Foo() // Generates CS0108
    {
    }
}
```

Das obige Beispiel führt auch **CS0108** Warnung, weil `B.Foo()` wird nicht automatisch überschrieben `A.Foo()` . Fügen Sie `override` wenn die Basisklasse überschrieben werden soll und polymorphes Verhalten verursacht wird. Fügen Sie `new` wenn Sie nichtpolymorphes Verhalten wünschen, und lösen Sie den Aufruf mit dem statischen Typ auf. Letzteres sollte mit Vorsicht angewendet werden, da dies zu schweren Verwirrungen führen kann.

Der folgende Code führt sogar zu einem Fehler:

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

Abgeleitete Klassen können Polymorphismus einführen

Der folgende Code ist vollkommen gültig (obwohl selten):

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
        }
    }

    public class B : A
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

Alle Objekte mit einer statischen Referenz von B (und ihren Ableitungen) verwenden jetzt Polymorphismus, um `Foo()` aufzulösen, während Referenzen von A `A.Foo()` .

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

Virtuelle Methoden können nicht privat sein

Der C # -Compiler verhindert strikt sinnlose Konstrukte. Als `virtual` gekennzeichnete Methoden können nicht privat sein. Da eine private Methode von einem abgeleiteten Typ nicht gesehen werden kann, kann sie auch nicht überschrieben werden. Dies kann nicht kompiliert werden:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

async, warte ab

Das `await` Schlüsselwort wurde als Teil von C # 5.0 hinzugefügt, das ab Visual Studio 2012 unterstützt wird. Es nutzt die Task Parallel Library (TPL), die das Multithreading relativ vereinfacht. Die Schlüsselwörter `async` und `await` werden in der gleichen Funktion paarweise verwendet (siehe unten). Das `await` Schlüsselwort wird verwendet, um die Ausführung der aktuellen asynchronen Methode anzuhalten, bis die erwartete asynchrone Task abgeschlossen ist und / oder ihre Ergebnisse zurückgegeben werden. Um das `await` Schlüsselwort verwenden zu können, muss die Methode, die es verwendet, mit dem `async` Schlüsselwort gekennzeichnet sein.

Mit `async` mit `void` wird dringend abgeraten. Für weitere Informationen können Sie hier nachschauen.

Beispiel:

```
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;
}
```

Ausgabe:

> "Einen nutzlosen Prozess starten ..."
>
> ** ... 1 Sekunde Verzögerung ... **
>
> "Es dauerte 1000 Millisekunden, bis ein unbrauchbarer Prozess ausgeführt wurde."

Die Schlüsselwortpaare `async` und `await` können weggelassen werden, wenn eine Rückgabemethode `Task` oder `Task<T>` nur eine einzelne asynchrone Operation zurückgibt.

Lieber als das:

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

Es ist bevorzugt, dies zu tun:

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

5,0

In C # 5.0 kann `await` nicht in `catch` und `finally` .

6,0

Mit C # 6.0 `await` kann verwendet werden , `catch` und `finally` .

verkohlen

Ein Zeichen ist ein einzelner Buchstabe, der in einer Variablen gespeichert ist. Es handelt sich um einen integrierten Werttyp, der zwei Byte Speicherplatz beansprucht. Es stellt den in `mscorlib.dll` gefundenen `System.Char` Datentyp dar, auf den von jedem C # -Projekt implizit verwiesen wird, wenn Sie sie erstellen.

Dafür gibt es mehrere Möglichkeiten.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

Ein `ushort, int, uint, long, ulong, float, double,` kann implizit in `ushort, int, uint, long, ulong, float, double,` oder `decimal` konvertiert werden und gibt den ganzzahligen Wert dieses char zurück.

```
ushort u = c;
```

gibt 99 usw. zurück

Es gibt jedoch keine impliziten Konvertierungen von anderen Typen in Zeichen. Stattdessen musst du sie werfen.

```
ushort u = 99;
 char c = (char)u;
```

sperren

`lock` bietet Thread-Sicherheit für einen Codeblock, sodass nur ein Thread innerhalb desselben Prozesses auf ihn zugreifen kann. Beispiel:

```
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
```

Anwendungsfälle:

Wann immer Sie über einen Codeblock verfügen, der Nebeneffekte erzeugen kann, wenn er von

mehreren Threads gleichzeitig ausgeführt wird. Das Sperrschlüsselwort zusammen mit einem **gemeinsam genutzten Synchronisationsobjekt** (im Beispiel `_objLock`) kann verwendet werden, um dies zu verhindern.

Beachten Sie, dass `_objLock` nicht `null` und mehrere Threads, die den Code ausführen, dieselbe Objektinstanz verwenden müssen (indem Sie sie entweder zu einem `static` Feld machen oder für beide Threads dieselbe Klasseninstanz verwenden)

Auf der Compilerseite ist das Schlüsselwort lock ein syntaktischer Zucker, der durch `Monitor.Enter(_lockObj);` und `Monitor.Exit(_lockObj);` . Wenn Sie also die Sperre durch Umgeben des Codeblocks mit diesen beiden Methoden ersetzen, erhalten Sie dieselben Ergebnisse. Sie können den tatsächlichen Code in Syntactic Sugar im C # -Sperrbeispiel sehen

Null

Eine Variable eines Referenztyps kann entweder eine gültige Referenz auf eine Instanz oder eine Nullreferenz enthalten. Die Nullreferenz ist der Standardwert von Referenztypvariablen sowie von nullwertfähigen Werttypen.

`null` ist das Schlüsselwort, das eine Nullreferenz darstellt.

Als Ausdruck kann es verwendet werden, um die Nullreferenz den Variablen der oben genannten Typen zuzuweisen:

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

Nicht-nullwertfähige Werttypen können keine Nullreferenz zugewiesen werden. Alle folgenden Zuweisungen sind ungültig:

```
int a = null;
float b = null;
decimal c = null;
```

Die Nullreferenz sollte *nicht* mit gültigen Instanzen verschiedener Typen verwechselt werden, z.

- eine leere Liste (`new List<int>()`)
- eine leere Zeichenfolge (`""`)
- die Zahl Null (`0` , `0f` , `0m`)
- das Nullzeichen (`'\0'`)

Manchmal ist es sinnvoll zu prüfen, ob etwas entweder null oder ein leeres / Standardobjekt ist. Die System.String.IsNullOrEmpty (String) -Methode kann verwendet werden, um dies zu überprüfen, oder Sie können Ihre eigene gleichwertige Methode implementieren.

```
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 + "!");
    }
}
```

intern

Das `internal` Schlüsselwort ist ein Zugriffsmodifizierer für Typen und Typmember. Auf interne Typen oder Member kann **nur innerhalb von Dateien in derselben Assembly zugegriffen werden**

Verwendungszweck:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

Der Unterschied zwischen verschiedenen Zugriffsmodifizierern wird hier klargestellt

Zugriffsmodifikatoren

Öffentlichkeit

Auf den Typ oder das Member kann durch einen beliebigen anderen Code in derselben Assembly oder einer anderen Assembly, auf die sie verweist, zugegriffen werden.

Privatgelände

Auf den Typ oder Member kann nur durch Code in derselben Klasse oder Struktur zugegriffen werden.

geschützt

Auf den Typ oder Member kann nur durch Code in derselben Klasse oder Struktur oder in einer abgeleiteten Klasse zugegriffen werden.

intern

Auf den Typ oder das Member kann durch einen beliebigen Code in derselben Assembly zugegriffen werden, jedoch nicht von einer anderen Assembly.

intern geschützt

> Auf den Typ oder das Member kann durch einen beliebigen Code in derselben Assembly oder durch eine abgeleitete Klasse in einer anderen Assembly zugegriffen werden.

Wenn **kein Zugriffsmodifizierer festgelegt** ist, wird ein Standard-Zugriffsmodifizierer verwendet. Es gibt also immer eine Form von Zugriffsmodifizierer, auch wenn sie nicht gesetzt ist.

woher

`where` kann in C # zwei Zwecken dienen: Typeinschränkung in einem generischen Argument und Filtern von LINQ-Abfragen.

Betrachten wir in einer generischen Klasse

```
public class Cup<T>
{
    // ...
}
```

T wird als Typparameter bezeichnet. Die Klassendefinition kann Einschränkungen für die tatsächlichen Typen auferlegen, die für T angegeben werden können.

Die folgenden Arten von Einschränkungen können angewendet werden:

- Werttyp
- Referenztyp
- Standardkonstruktor
- Vererbung und Umsetzung

Werttyp

In diesem Fall können nur `struct` s (dazu gehören "primitive" Datentypen wie `int` , `boolean` usw.) angegeben werden

```
public class Cup<T> where T : struct
{
    // ...
}
```

Referenztyp

In diesem Fall können nur Klassenarten geliefert werden

```
public class Cup<T> where T : class
{
    // ...
}
```

Hybridwert / Referenztyp

Gelegentlich ist es wünschenswert, Typargumente auf die in einer Datenbank verfügbaren Argumente zu beschränken. Diese werden normalerweise Wertetypen und Zeichenfolgen zugeordnet. Da alle Typeinschränkungen erfüllt sein müssen, ist es nicht möglich anzugeben, `where T : struct or string` (dies ist keine gültige Syntax). Eine Problemumgehung besteht darin, Typargumente auf `IConvertible` zu beschränken, das die folgenden Typen enthält: "Boolean", "Boolean", "SByte", "Byte", "Int16", "IntInt16", "Int32", "Int32", "Int64", "Int64", "Int64", "Int64", "Single", "Double", Decimal, DateTime, Char und String. " Es ist möglich, dass andere Objekte IConvertible implementieren, obwohl dies in der Praxis selten ist.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

Standardkonstruktor

Es sind nur Typen zulässig, die einen Standardkonstruktor enthalten. Dazu gehören Werttypen und Klassen, die einen standardmäßigen (parameterlosen) Konstruktor enthalten

```
public class Cup<T> where T : new
{
    // ...
}
```

Vererbung und Umsetzung

Es können nur Typen angegeben werden, die von einer bestimmten Basisklasse erben oder eine bestimmte Schnittstelle implementieren.

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

Die Einschränkung kann sogar auf einen anderen Typparameter verweisen:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Für ein Typargument können mehrere Einschränkungen angegeben werden:

```
public class Cup<T> where T : class, new()
{
    // ...
```

```
}
```

Die vorherigen Beispiele zeigen generische Einschränkungen für eine Klassendefinition. Einschränkungen können jedoch überall dort verwendet werden, wo ein Typargument angegeben wird: Klassen, Strukturen, Schnittstellen, Methoden usw.

`where` kann auch eine LINQ-Klausel sein. In diesem Fall ist es analog zu `WHERE` in SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
    // prints 2 1 3 2 0
```

extern

Mit dem Schlüsselwort `extern` Methoden deklariert, die extern implementiert werden. Dies kann in Verbindung mit dem Attribut DllImport verwendet werden, um über Interop-Services nicht verwalteten Code aufzurufen. was in diesem Fall mit `static` Modifikator kommt

Zum Beispiel:

```
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);
    }
}
```

Hierbei wird die SetForegroundWindow-Methode aus der User32.dll-Bibliothek importiert

Dies kann auch verwendet werden, um einen externen Assembly-Alias zu definieren. So können wir auf unterschiedliche Versionen derselben Komponenten aus einer Baugruppe verweisen.

Um auf zwei Assemblys mit denselben vollständig qualifizierten Typnamen zu verweisen, muss an

der Eingabeaufforderung ein Alias wie folgt angegeben werden:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

Dadurch werden die externen Aliasnamen GridV1 und GridV2 erstellt. Um diese Aliasnamen innerhalb eines Programms zu verwenden, referenzieren Sie sie mit dem Schlüsselwort extern. Zum Beispiel:

```
extern alias GridV1;
extern alias GridV2;
```

bool

Schlüsselwort zum Speichern der booleschen Werte `true` und `false` . bool ist ein Alias von System.Boolean.

Der Standardwert eines bool ist false.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

Damit ein Bool Nullwerte zulässt, muss er als Bool initialisiert werden.

Der Standardwert eines bool? ist Null.

```
bool? a // default value is null
```

wann

Das `when` ist ein in **C # 6** hinzugefügtes Schlüsselwort und wird für die Ausnahmefilterung verwendet.

Vor der Einführung des Schlüsselworts `when` Sie für jeden Ausnahmetyp eine catch-Klausel haben können. Durch das Hinzufügen des Schlüsselworts ist jetzt eine feinere Kontrolle möglich.

A , `when` Ausdruck ist mit einem befestigten `catch` und nur dann , wenn die , `when` Bedingung ist `true` , der `catch` wird Klausel ausgeführt werden. Es ist möglich, mehrere `catch` Klauseln mit denselben Ausnahmeklassentypen und unterschiedlichen `when` Bedingungen zu verwenden.

```
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);
```

ungeprüft

Das `unchecked` Schlüsselwort verhindert, dass der Compiler nach Überläufen / Unterläufen sucht.

Zum Beispiel:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Ohne das `unchecked` Schlüsselwort wird keine der beiden Additionsoperationen kompiliert.

Wann ist das nützlich?

Dies ist hilfreich, da dies die Beschleunigung von Berechnungen beschleunigen kann, die definitiv nicht überlaufen, da die Überprüfung des Überlaufs Zeit erfordert oder wenn ein Überlauf / Unterlauf gewünscht wird (z. B. beim Generieren eines Hashcodes).

Leere

Das reservierte Wort `"void"` ist ein Alias des Typs `"void" System.Void` " und hat zwei Verwendungszwecke:

1. Deklarieren Sie eine Methode ohne Rückgabewert:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

Eine Methode mit einem Rückgabewert vom Typ void kann immer noch das Schlüsselwort `return` in seinem Körper enthalten. Dies ist nützlich, wenn Sie die Ausführung der Methode beenden und

den Fluss an den Aufrufer zurückgeben möchten:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Deklarieren Sie einen Zeiger auf einen unbekannten Typ in einem unsicheren Kontext.

In einem unsicheren Kontext kann ein Typ ein Zeigertyp, ein Werttyp oder ein Referenztyp sein. Eine `int* myInt` ist normalerweise `type* identifier` , wobei der Typ ein bekannter Typ ist - dh `int* myInt` , kann aber auch `void* identifier` , wobei der Typ unbekannt ist.

Beachten Sie, dass die Deklaration eines ungültigen Zeigertyps von Microsoft nicht empfohlen wird.

wenn, wenn ... sonst, wenn ... sonst wenn

Die `if` Anweisung wird verwendet, um den Programmfluss zu steuern. Eine `if` Anweisung gibt an, welche Anweisung basierend auf dem Wert eines `Boolean` Ausdrucks ausgeführt werden soll.

Bei einer einzelnen Anweisung sind die `braces` {} optional, werden jedoch empfohlen.

```
int a = 4;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

Die `if` Klausel kann auch eine `else` Klausel enthalten, die ausgeführt wird, wenn die Bedingung zu false ausgewertet wird:

```
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"
```

Mit dem `if` ... `else if` -Konstrukt können Sie mehrere Bedingungen angeben:

```
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"
```

Beachten Sie, dass das Steuerelement andere Bedingungen überspringt, wenn eine Bedingung erfüllt ist, und springt zum Ende des jeweiligen if-Konstrukts. Die *Reihenfolge* der Tests ist daher wichtig, wenn Sie if .. else if-Konstrukt verwenden

Boolesche C # -Ausdrücke verwenden eine Kurzschlussbewertung . Dies ist wichtig in Fällen, in denen die Beurteilung der Bedingungen Nebenwirkungen haben kann:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

Es kann nicht garantiert werden, dass `someOtherBooleanMethodWithSideEffects` tatsächlich ausgeführt wird.

Dies ist auch in Fällen wichtig, in denen frühere Bedingungen gewährleisten, dass spätere Bedingungen "sicher" sind. Zum Beispiel:

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

Die Reihenfolge ist in diesem Fall sehr wichtig, weil, wenn wir die Reihenfolge umkehren:

```
if (someCollection.Count > 0 && someCollection != null) {
```

es wird ein Wurf `NullReferenceException` , wenn `someCollection` ist `null` .

tun

Der do-Operator durchläuft einen Codeblock, bis eine bedingte Abfrage gleich false ist. Die do-while - Schleife kann auch durch eine unterbrochen werden `goto` , `return` , `break` oder `throw`

Aussage.

Die Syntax für das `do` Schlüsselwort lautet:

> do { *Codeblock;* } while (*Bedingung*);

Beispiel:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Ausgabe:

> "Do ist in Loop Nummer 1."
> "Do ist in Loop Nummer 2."
> "Do ist in Loop Nummer 3."
> "Do ist in Loop Nummer 4."
> "Do ist in Loop Nummer 5."

Im Gegensatz zur `while` Schleife ist die do-while-Schleife **Exit Controlled** . Dies bedeutet, dass die do-while-Schleife ihre Anweisungen mindestens einmal ausführt, selbst wenn die Bedingung beim ersten Mal fehlschlägt.

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

Operator

Die meisten integrierten Operatoren (einschließlich Konvertierungsoperatoren) können mit dem Schlüsselwort `operator` zusammen mit den Modifizierern `public` und `static` überladen werden.

Die Operatoren gibt es in drei Formen: unäre Operatoren, binäre Operatoren und Konvertierungsoperatoren.

Unäre und binäre Operatoren erfordern mindestens einen Parameter desselben Typs wie der enthaltende Typ, und einige erfordern einen zusätzlichen Übereinstimmungsoperator.

Konvertierungsoperatoren müssen in den oder den umgebenden Typ konvertieren.

```
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}}}";

}
```

Beispiel

```
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

Ein `struct` ist ein Wertetyp, der normalerweise zum Einkapseln kleiner Gruppen verwandter Variablen verwendet wird, z. B. der Koordinaten eines Rechtecks oder der Merkmale eines Elements in einem Inventar.

> Klassen sind Referenztypen, Strukturen sind Werttypen.

```
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();
        }
    }
}
```

Strukturen können auch Konstruktoren, Konstanten, Felder, Methoden, Eigenschaften, Indexer, Operatoren, Ereignisse und verschachtelte Typen enthalten. Wenn jedoch mehrere solcher Member erforderlich sind, sollten Sie in Betracht ziehen, Ihren Typ stattdessen als Klasse zu definieren.

Einige Vorschläge von MS, wann und wann Klassen verwendet werden sollen:

ERWÄGEN

Definieren einer Struktur anstelle einer Klasse, wenn Instanzen des Typs klein sind und häufig kurzlebig sind oder häufig in andere Objekte eingebettet sind.

VERMEIDEN

Definieren einer Struktur, sofern der Typ nicht alle der folgenden Merkmale aufweist:

- Es stellt logisch einen einzelnen Wert dar, ähnlich den primitiven Typen (int, double usw.).
- Es hat eine Instanzgröße unter 16 Byte.
- Es ist unveränderlich.
- Es muss nicht häufig verpackt werden.

Schalter

Die `switch` Anweisung ist eine Steueranweisung, die einen aus einer Kandidatenliste auszuführenden Schalterabschnitt auswählt. Eine switch-Anweisung enthält einen oder mehrere switch-Abschnitte. Jeder Schalter Abschnitt enthält eine oder mehr `case` Etiketten , gefolgt von einer oder mehrer Anweisungen. Wenn keine Fallbezeichnung einen übereinstimmenden Wert enthält, wird die Kontrolle an den `default` , sofern vorhanden. Der Fallfall wird in C # streng genommen nicht unterstützt. Wenn 1 oder mehrere jedoch `case` Etiketten leer sind, wird die Ausführung der Code des nächsten folgen `case` Block, der Code enthält. Dies ermöglicht Gruppierung mehrerer `case` Etiketten mit der gleichen Ausführung. Im folgende Beispiel, wenn `month` 12 entspricht, wird der Code in `case 2` wird ausgeführt werden , da der `case` Etikett `12` `1` und `2` sind gruppiert. Wenn ein `case` Block nicht leer ist, muss vor dem nächsten `case` Label eine `break` vorhanden sein, andernfalls weist der Compiler auf einen Fehler hin.

```
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;
}
```

Ein `case` kann nur durch einen Wert gekennzeichnet werden zum *Zeitpunkt* der *Kompilierung* bekannt (zB `1` , `"str"` , `Enum.A`), so dass eine `variable` ist kein gültiger `case` Etikett, aber ein `const` oder `Enum` Wert (sowie jede wörtlicher Wert).

Schnittstelle

Eine `interface` enthält die Signaturen von Methoden, Eigenschaften und Ereignissen. Die abgeleiteten Klassen definieren die Mitglieder, da das Interface nur die Deklaration der Mitglieder enthält.

Eine Schnittstelle wird mit dem `interface` deklariert.

```
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); } }
}
```

unsicher

Das `unsafe` Schlüsselwort kann in Typ- oder Methodendeklarationen oder zum Deklarieren eines Inline-Blocks verwendet werden.

Der Zweck dieses Schlüsselworts besteht darin, die Verwendung der *unsicheren Teilmenge* von C # für den betreffenden Block zu ermöglichen. Die unsichere Teilmenge enthält Funktionen wie Zeiger, Stapelzuordnung, C-artige Arrays usw.

Unsicherer Code ist nicht überprüfbar und wird daher nicht empfohlen. Das Kompilieren von unsicherem Code erfordert die Übergabe eines Wechsels an den C # -Compiler. Darüber hinaus erfordert die CLR, dass die ausgeführte Assembly volle Vertrauenswürdigkeit besitzt.

Trotz dieser Einschränkungen kann unsicherer Code dazu verwendet werden, einige Operationen performanter zu gestalten (z. B. Array-Indizierung) oder einfacher zu machen (z. B. Interop mit einigen nicht verwalteten Bibliotheken).

Als sehr einfaches Beispiel

```
// 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
   }
}
```

Bei der Arbeit mit Zeigern können wir die Werte von Speicherorten direkt ändern, anstatt sie mit Namen ansprechen zu müssen. Beachten Sie, dass dies häufig die Verwendung des festen Schlüsselworts erfordert, um mögliche Speicherbeschädigung zu verhindern, da der Garbage Collector die Dinge in Bewegung setzt (andernfalls erhalten Sie den Fehler CS0212). Da eine Variable, die "fixiert" wurde, nicht geschrieben werden kann, muss häufig auch ein zweiter Zeiger vorhanden sein, der auf dieselbe Position wie der erste zeigt.

```
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++;"
```

```
        }
    }
}
```

Ausgabe:

```
1
4
9
16
25
36
49
64
81
100
```

`unsafe` erlaubt auch die Verwendung von stackalloc, wodurch Speicherplatz wie _alloca in der C-Laufzeitbibliothek zugewiesen wird. Das obige Beispiel `stackalloc` wie folgt `stackalloc` , um `stackalloc` zu verwenden:

```
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++;
}
```

(Ausgabe ist wie oben)

implizit

Das `implicit` Schlüsselwort wird verwendet, um einen Konvertierungsoperator zu überladen. Sie können beispielsweise eine `Fraction` Klasse deklarieren, die bei Bedarf automatisch in ein `double` konvertiert werden soll und die automatisch aus `int` konvertiert werden kann:

```
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);
    }
}
```

wahr falsch

Die Schlüsselwörter `true` und `false` haben zwei Zwecke:

1. Als wörtliche boolesche Werte

```
var myTrueBool = true;
var myFalseBool = false;
```

2. Als Operatoren kann das überlastet werden

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Das Überladen des falschen Operators war vor C # 2.0 vor der Einführung von `Nullable` Typen `Nullable` .
Ein Typ, der den `true` Operator überlädt, muss auch den `false` Operator überladen.

Schnur

`string` ist ein Alias für den .NET-Datentyp `System.String` , der das `System.String` Text (Zeichenfolgen) ermöglicht.

Notation:

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Jedes Zeichen in der Zeichenfolge ist in UTF-16 codiert. Dies bedeutet, dass jedes Zeichen mindestens 2 Byte Speicherplatz benötigt.

ushort

Ein numerischer Typ, der zum Speichern von positiven 16-Bit-Ganzzahlen verwendet wird. `ushort` ist ein Alias für `System.UInt16` und belegt 2 Byte Speicher.

Gültiger Bereich ist `0` bis `65535` .

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

sbyte

Ein numerischer Typ zum Speichern von *vorzeichenbehafteten* 8-Bit-Ganzzahlen. `sbyte` ist ein Alias für `System.SByte` und belegt 1 Byte Speicherplatz. Verwenden Sie für das vorzeichenlose Äquivalent `byte` .

Gültiger Bereich ist `-127` bis `127` (der Rest wird zum Speichern des Zeichens verwendet).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

var

Eine implizit typisierte lokale Variable, die stark typisiert ist, als ob der Benutzer den Typ deklariert hätte. Im Gegensatz zu anderen Variablendeklarationen bestimmt der Compiler den Variablentyp, den dieser darstellt, basierend auf dem ihm zugewiesenen Wert.

```
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).
```

Im Gegensatz zu anderen Variablentypen müssen Variablendefinitionen mit diesem Schlüsselwort bei der Deklaration initialisiert werden. Dies liegt daran, dass das Schlüsselwort **var** eine implizit typisierte Variable darstellt.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

Mit dem Schlüsselwort **var** können auch neue Datentypen im laufenden Betrieb erstellt werden. Diese neuen Datentypen werden als *anonyme Typen bezeichnet* . Sie sind sehr nützlich, da sie es einem Benutzer ermöglichen, einen Satz von Eigenschaften zu definieren, ohne zuvor einen Objekttyp explizit deklarieren zu müssen.

Normaler anonymer Typ

```
var a = new { number = 1, text = "hi" };
```

LINQ-Abfrage, die einen anonymen Typ zurückgibt

```
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);
}
```

Sie können das var-Schlüsselwort in der foreach-Anweisung verwenden

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

delegieren

Delegaten sind Typen, die einen Verweis auf eine Methode darstellen. Sie werden verwendet, um Methoden als Argumente an andere Methoden zu übergeben.

Delegierte können statische Methoden, Instanzmethoden, anonyme Methoden oder Lambda-Ausdrücke enthalten.

```
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;
    }
}
```

Bei der Zuweisung einer Methode zu einem Delegierten ist zu beachten, dass die Methode denselben Rückgabetyp sowie dieselben Parameter aufweisen muss. Dies unterscheidet sich von einer "normalen" Methodenüberladung, bei der nur die Parameter die Signatur der Methode definieren.

Ereignisse werden auf Delegierten aufgebaut.

Veranstaltung

Ein `event` ermöglicht es dem Entwickler, ein Benachrichtigungsmuster zu implementieren.

Einfaches Beispiel

```
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-Referenz

teilweise

Das Schlüsselwort `partial` kann während der Typdefinition von Klassen, Strukturen oder Schnittstellen verwendet werden, um die Typdefinition in mehrere Dateien aufzuteilen. Dies ist nützlich, um neue Funktionen in automatisch generierten Code zu integrieren.

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;}
    }
}
```

Hinweis: Eine Klasse kann in beliebig viele Dateien aufgeteilt werden. Alle Deklarationen müssen sich jedoch unter demselben Namespace und derselben Assembly befinden.

Methoden können auch mit dem Schlüsselwort `partial` deklariert werden. In diesem Fall enthält

eine Datei nur die Methodendefinition und eine andere Datei die Implementierung.

> Bei einer Teilmethode ist ihre Signatur in einem Teil eines Teiltyps definiert und ihre Implementierung in einem anderen Teil des Typs definiert. Mit partiellen Methoden können Klassenentwickler, ähnlich wie Ereignisbehandlungsroutinen, Methoden-Hooks bereitstellen, die Entwickler möglicherweise implementieren oder nicht. Wenn der Entwickler keine Implementierung bereitstellt, entfernt der Compiler die Signatur zur Kompilierzeit. Für Teilmethoden gelten folgende Bedingungen:
>
> - Unterschriften in beiden Teilen des Teiltyps müssen übereinstimmen.
> - Die Methode muss ungültig sein.
> - Es sind keine Zugriffsmodifizierer zulässig. Teilmethoden sind implizit privat.
>
> - 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);
        }
    }
}
```

Anmerkung: Der Typ, der die partielle Methode enthält, muss auch als partiell deklariert werden.

Schlüsselwörter online lesen: https://riptutorial.com/de/csharp/topic/26/schlusselworter

Kapitel 120: Schnittstellen

Examples

Eine Schnittstelle implementieren

Eine Schnittstelle wird verwendet, um das Vorhandensein einer Methode in jeder Klasse zu erzwingen, die sie "implementiert". Das Interface wird mit dem Schlüsselwort `interface` und kann von einer Klasse 'implementiert' werden, indem nach dem Klassennamen `: InterfaceName` hinzugefügt wird. Eine Klasse kann mehrere Schnittstellen implementieren, indem jede Schnittstelle durch ein Komma getrennt wird.
`: 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";
    }
}
```

Da sie `INoiseMaker` implementieren, müssen sowohl `cat` als auch `dog` die `string MakeNoise()` -Methode enthalten und können ohne sie nicht kompiliert werden.

Implementierung mehrerer Schnittstellen

```
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";
    }
}
```

Explizite Schnittstellenimplementierung

Eine explizite Schnittstellenimplementierung ist erforderlich, wenn Sie mehrere Schnittstellen implementieren, die eine gemeinsame Methode definieren. Abhängig davon, welche Schnittstelle zum Aufrufen der Methode verwendet wird, sind verschiedene Implementierungen erforderlich eine gemeinsame Implementierung ist möglich).

```
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...
```

Die Implementierung kann von keiner anderen Stelle aus aufgerufen werden, es sei denn, Sie verwenden die Schnittstelle:

```
public class Golfer : IGolfPlayer
{
```

```
    string IGolfPlayer.Drive()
    {
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

Aus diesem Grund kann es vorteilhaft sein, komplexen Implementierungscode einer explizit implementierten Schnittstelle in einer separaten, privaten Methode zu speichern.

Eine explizite Schnittstellenimplementierung kann natürlich nur für Methoden verwendet werden, die für diese Schnittstelle tatsächlich vorhanden sind:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

Ebenso führt die Verwendung einer expliziten Schnittstellenimplementierung ohne Deklaration dieser Schnittstelle für die Klasse ebenfalls zu einem Fehler.

Hinweis:

Das explizite Implementieren von Schnittstellen kann auch verwendet werden, um toten Code zu vermeiden. Wenn eine Methode nicht mehr benötigt wird und von der Schnittstelle entfernt wird, beschwert sich der Compiler über jede noch vorhandene Implementierung.

Hinweis:

Programmierer erwarten, dass der Vertrag unabhängig vom Kontext des Typs gleich ist, und die explizite Implementierung sollte beim Aufruf kein anderes Verhalten zeigen. Im Gegensatz zum obigen Beispiel sollten `IGolfPlayer.Drive` und `Drive` dasselbe tun, wenn dies möglich ist.

Warum verwenden wir Schnittstellen?

Eine Schnittstelle ist eine Definition eines Vertrages zwischen dem Benutzer der Schnittstelle und der Klasse, die sie implementiert. Eine Möglichkeit, sich eine Schnittstelle vorzustellen, ist die Deklaration, dass ein Objekt bestimmte Funktionen ausführen kann.

Nehmen wir an, wir definieren eine Schnittstelle `IShape` , um verschiedene Arten von Formen `IShape` Wir erwarten, dass eine Form einen Bereich hat. `IShape` definieren wir eine Methode, mit der die Schnittstellenimplementierungen gezwungen werden, ihren Bereich zurückzugeben:

```
public interface IShape
{
    double ComputeArea();
}
```

Nehmen wir an, wir haben die folgenden zwei Formen: ein `Rectangle` und einen `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;
    }
}
```

Jeder von ihnen hat seine eigene Definition seiner Fläche, aber beide sind Formen. Daher ist es nur logisch, sie als `IShape` in unserem Programm zu sehen:

```
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
```

Schnittstellen-Grundlagen

Die Funktion einer Schnittstelle wird als "Vertrag" der Funktionalität bezeichnet. Das bedeutet, dass Eigenschaften und Methoden deklariert, aber nicht implementiert werden.

So anders als Klassen Interfaces:

- Kann nicht instanziiert werden
- Kann keine Funktionalität haben
- Kann nur Methoden enthalten * *(Eigenschaften und Ereignisse sind Methoden intern)*
- Das Erben einer Schnittstelle wird als "Implementieren" bezeichnet.
- Sie können von einer Klasse erben, Sie können jedoch mehrere Schnittstellen "implementieren"

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Dinge zu beachten:

- Das Präfix "I" ist eine Namenskonvention für Schnittstellen.
- Der Funktionsrumpf wird durch ein Semikolon ";" ersetzt.
- Eigenschaften sind auch zulässig, da sie intern auch Methoden sind

```
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
```

Dies ist besonders nützlich, wenn Sie mit UI-Frameworks wie WinForms oder WPF arbeiten, da es zwingend erforderlich ist, von einer Basisklasse zu erben, um ein Benutzersteuerelement zu erstellen, und Sie die Möglichkeit verlieren, Abstraktionen über verschiedene Steuerelementtypen zu erstellen. Ein Beispiel? In Kürze:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

Das vorgeschlagene Problem ist, dass beide ein Konzept von "Text" enthalten, die Eigenschaftsnamen jedoch unterschiedlich sind. Sie können keine *abstrakten Basisklassen* erstellen, da sie obligatorisch auf zwei verschiedene Klassen vererbt werden. Eine Schnittstelle kann das erleichtern

```
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; }
}
```

Jetzt ist MyButton und MyTextBlock austauschbar.

```
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


}
```

Mitglieder mit expliziter Implementierung "ausblenden"

Hassen Sie es nicht, wenn Schnittstellen Ihre Klasse mit zu vielen Mitgliedern verschmutzen, für die Sie sich nicht interessieren? Nun, ich habe eine Lösung bekommen! Explizite Implementierungen

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

Normalerweise würden Sie die Klasse so implementieren.

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

Jedes Mitglied ist öffentlich.

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Antwort: ich nicht. Es sollte also auch nicht als öffentlich deklariert werden, sondern durch die Angabe der Mitglieder als privat wird der Compiler einen Fehler auslösen

Die Lösung ist die explizite Implementierung:

```
public class MyObjectWithMessages : IMessageService{
```

```
    void IMessageService.OnMessageRecieve() {

    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

Jetzt haben Sie die Mitglieder nach Bedarf implementiert und sie werden keine Mitglieder als öffentlich ausweisen.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

Wenn Sie ernsthaft weiterhin auf das Member zugreifen möchten, obwohl es explizit implementiert ist, müssen Sie das Objekt nur in die Schnittstelle umwandeln und Sie können es tun.

```
((IMessageService)obj).OnMessageRecieve();
```

Vergleichbar als Beispiel für die Implementierung einer Schnittstelle

Schnittstellen können abstrakt erscheinen, bis Sie sie in der Praxis erscheinen. `IComparable` und `IComparable<T>` sind hervorragende Beispiele dafür, warum Schnittstellen für uns hilfreich sein können.

Nehmen wir an, in einem Programm für einen Online-Shop haben wir verschiedene Artikel, die Sie kaufen können. Jeder Artikel hat einen Namen, eine ID-Nummer und einen Preis.

```
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

}
```

Wir haben unsere `Item` in einer `List<Item>` gespeichert, und in unserem Programm möchten wir unsere Liste nach ID-Nummern von klein nach groß sortieren. Anstatt unseren eigenen Sortieralgorithmus zu schreiben, können wir stattdessen die `Sort()` Methode verwenden, die `List<T>` bereits hat. Da unsere `Item` Klasse jedoch jetzt ist, kann die `List<T>` nachvollziehen, in welcher Reihenfolge die Liste sortiert wird. Hier kommt die `IComparable` Schnittstelle ins `IComparable` .

Um die `CompareTo` Methode korrekt zu implementieren, sollte `CompareTo` eine positive Zahl zurückgeben, wenn der Parameter "kleiner als" der aktuelle ist, Null, wenn sie gleich sind, und eine negative Zahl, wenn der Parameter "größer als" ist.

```
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
```

Hier ist das Beispiel `Item` Implementierung der Schnittstelle s‘:

```
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

}
```

Auf Oberflächenebene gibt die `CompareTo` Methode in unserem Artikel einfach die Differenz in ihren ID-Nummern zurück. Was macht das oben in der Praxis?

Nun, wenn wir rufen `Sort()` auf einer `List<Item>` Objekt, die `List` wird die automatisch aufrufen `Item` ,s `CompareTo` Methode , wenn es zu bestimmen , muss in welcher Reihenfolge in Objekte zu setzen. Außerdem neben `List<T>` , alle andere Objekte Wenn zwei Objekte miteinander verglichen werden müssen, funktioniert das mit dem `Item` da wir die Fähigkeit definiert haben, zwei verschiedene `Item` zu vergleichen.

Schnittstellen online lesen: https://riptutorial.com/de/csharp/topic/2208/schnittstellen

Kapitel 121: Singleton-Implementierung

Examples

Statisch initialisiertes Singleton

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

Diese Implementierung ist threadsicher , da in diesem Fall `instance` Objekt im statischen Konstruktor initialisiert. Die CLR stellt bereits sicher, dass alle statischen Konstruktoren threadsicher ausgeführt werden.

Das Ändern der `instance` ist keine Thread-sichere Operation, daher garantiert das `readonly` Attribut die Unveränderlichkeit nach der Initialisierung.

Faules, fadensicheres Singleton (mit Double Checked Locking)

Diese Thread-sichere Version eines Singleton wurde in den frühen .NET-Versionen benötigt, bei denen die `static` Initialisierung nicht garantiert Thread-sicher war. In moderneren Versionen des Frameworks wird normalerweise ein statisch initialisierter Singleton bevorzugt, da es sehr einfach ist, Implementierungsfehler im folgenden Muster zu machen.

```
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;
```

```
        }
    }
}
```

Beachten Sie, dass die Prüfung `if (instance == null)` zweimal durchgeführt wird: einmal vor dem Sperren und einmal danach. Diese Implementierung wäre auch ohne die erste Nullprüfung Thread-sicher. Dies würde jedoch bedeuten, dass *jedes Mal, wenn* die Instanz angefordert wird, eine Sperre erlangt *wird* , was die Performance beeinträchtigen würde. Die erste Nullprüfung wird hinzugefügt, damit die Sperre nicht erfasst wird, sofern dies nicht erforderlich ist. Die zweite Nullüberprüfung stellt sicher, dass nur der erste Thread, der die Sperre erhält, die Instanz erstellt. Die anderen Threads finden die zu füllende Instanz und springen voraus.

Faules, fadensicheres Singleton (mit Lazy)

Der .NET 4.0-Typ Lazy garantiert eine Thread-sichere Objektinitialisierung, sodass dieser Typ für Singletons verwendet werden könnte.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

Mit `Lazy<T>` wird sichergestellt, dass das Objekt nur instanziiert wird, wenn es irgendwo im aufrufenden Code verwendet wird.

Eine einfache Verwendung wird wie folgt sein:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Live-Demo zu .NET-Geige

Fauler, threadsicherer Singleton (für .NET 3.5 oder älter, alternative Implementierung)

Da Sie in .NET 3.5 und älter keine `Lazy<T>` -Klasse haben, verwenden Sie das folgende Muster:

```
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();
    }
}
```

Dies ist inspiriert von Jon Skeets Blogpost .

Da die `Nested` Klasse verschachtelt und privat ist, wird die Instantiierung der Singleton-Instanz nicht durch den Zugriff auf andere Mitglieder der `Sigleton` Klasse (z. B. eine öffentliche readonly-Eigenschaft) ausgelöst.

Entsorgen der Singleton-Instanz, wenn sie nicht mehr benötigt wird

Die meisten Beispiele zeigen, wie ein `LazySingleton` Objekt instanziiert und gehalten wird, bis die `LazySingleton` beendet wurde, selbst wenn dieses Objekt von der Anwendung nicht mehr benötigt wird. Eine Lösung hierfür ist, `IDisposable` zu implementieren und die Objektinstanz wie folgt auf null zu setzen:

```
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;
}
```

Der obige Code gibt die Instanz vor der Anwendungsbeendigung frei, aber nur, wenn die Verbraucher nach jeder Verwendung `Dispose()` für das Objekt aufrufen. Da es keine Garantie dafür gibt, dass dies geschehen wird, gibt es auch keine Garantie, dass die Instanz jemals beseitigt wird. Wenn diese Klasse jedoch intern verwendet wird, ist es einfacher sicherzustellen, dass die `Dispose()` -Methode nach jeder Verwendung aufgerufen wird. Ein Beispiel folgt:

```
public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}
```

Bitte beachten Sie, dass dieses Beispiel **nicht threadsicher ist** .

Singleton-Implementierung online lesen: https://riptutorial.com/de/csharp/topic/1192/singleton-implementierung

Kapitel 122: Stacktraces lesen und verstehen

Einführung

Ein Stack-Trace ist eine große Hilfe beim Debuggen eines Programms. Sie erhalten eine Stack-Ablaufverfolgung, wenn Ihr Programm eine Ausnahme auslöst und manchmal, wenn das Programm abnormal beendet wird.

Examples

Stack-Trace für eine einfache NullReferenceException in Windows Forms

Lassen Sie uns ein kleines Stück Code erstellen, das eine Ausnahme auslöst:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

Wenn wir das ausführen, erhalten wir die folgende Ausnahme und Stack-Trace:

```
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)
```

Die Stapelverfolgung geht so weiter, aber dieser Teil wird für unsere Zwecke ausreichen.

Am oberen Rand des Stack-Trace sehen wir die Zeile:

> at WindowsFormsApplication1.Form1.button1_Click (Objektsender, EventArgs e) in F: \ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.cs: Zeile 29

Dies ist der wichtigste Teil. Es zeigt uns die *genaue* Zeile, in der die Exception aufgetreten ist: Zeile 29 in Form1.cs.
Hier beginnen Sie Ihre Suche.

Die zweite Zeile ist

> bei System.Windows.Forms.Control.OnClick (EventArgs e)

Dies ist die Methode, die `button1_Click` aufgerufen `button1_Click` . Nun wissen wir, dass `button1_Click` , wo der Fehler aufgetreten ist, von `System.Windows.Forms.Control.OnClick` aufgerufen wurde.

Wir können so weitermachen; Die dritte Zeile ist

> bei System.Windows.Forms.Button.OnClick (EventArgs e)

Dies ist wiederum der Code, der `System.windows.Forms.Control.OnClick` aufgerufen `System.windows.Forms.Control.OnClick` .

Der Stack-Trace ist die Liste der Funktionen, die aufgerufen wurden, bis der Code auf die Ausnahme stieß. Wenn Sie dem folgen, können Sie herausfinden, welchen Ausführungspfad Ihr Code folgte, bis er in Schwierigkeiten geriet!

Beachten Sie, dass die Stack-Trace Aufrufe vom .Net-System enthält. Normalerweise müssen Sie nicht alle `System.Windows.Forms` Microsofts `System.Windows.Forms` befolgen, um herauszufinden, was schiefgegangen ist, sondern nur den Code, der zu Ihrer eigenen Anwendung gehört.

Warum wird dies als "Stack-Trace" bezeichnet?
Jedes Mal, wenn ein Programm eine Methode aufruft, verfolgt es den Ort, an dem es sich befand. Es hat eine Datenstruktur namens "Stack", in der der letzte Speicherort abgelegt wird.
Wenn die Ausführung der Methode abgeschlossen ist, wird auf dem Stack nach dem Ort gesucht, an dem sie sich vor dem Aufruf der Methode befand - und von dort aus fortgesetzt.

Der Stack teilt dem Computer also mit, wo er aufgehört hat, bevor er eine neue Methode aufruft.

Es dient aber auch zur Fehlersuche. Wie ein Detektiv, der die Schritte verfolgt, die ein Krimineller bei der Begehung seiner Straftat unternommen hat, kann ein Programmierer den Stapel verwenden, um die Schritte zu verfolgen, die ein Programm ausgeführt hat, bevor es abgestürzt ist.

Stacktraces lesen und verstehen online lesen:
https://riptutorial.com/de/csharp/topic/8923/stacktraces-lesen-und-verstehen

Kapitel 123: Statische Klassen

Examples

Statisches Schlüsselwort

Das statische Schlüsselwort bedeutet zwei Dinge:

1. Dieser Wert ändert sich nicht von Objekt zu Objekt, sondern ändert sich für eine Klasse als Ganzes
2. Statische Eigenschaften und Methoden erfordern keine Instanz.

```
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

    }
}
```

Statische Klassen

Das Schlüsselwort "statisch" hat beim Verweisen auf eine Klasse drei Auswirkungen:

1. Sie **können keine** Instanz einer statischen Klasse erstellen (dadurch wird sogar der Standardkonstruktor entfernt).

2. Alle Eigenschaften und Methoden in der Klasse **müssen ebenfalls** statisch sein.
3. Eine `static` Klasse ist eine `sealed` Klasse, dh sie kann nicht vererbt werden.

```
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
    }
}
```

Statische Klassenlebensdauer

Eine `static` Klasse wird beim Member-Zugriff träge initialisiert und ist für die Dauer der Anwendungsdomäne gültig.

```
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");
    }
}
```

Statische Klassen online lesen: https://riptutorial.com/de/csharp/topic/1653/statische-klassen

Kapitel 124: Stoppuhren

Syntax

- stopWatch.Start () - Startet die Stoppuhr.
- stopWatch.Stop () - Stoppt die Stoppuhr.
- stopWatch.Elapsed - Ruft die vom aktuellen Intervall gemessene Gesamtzeit ab.

Bemerkungen

Stoppuhren werden häufig in Benchmarking-Programmen verwendet, um Zeitcode einzugeben und zu sehen, wie optimal unterschiedliche Codesegmente für die Ausführung sind.

Examples

Instanz einer Stoppuhr erstellen

Eine Stoppuhr-Instanz kann die abgelaufene Zeit über mehrere Intervalle hinweg messen, wobei die gesamte abgelaufene Zeit alle einzelnen Intervalle addiert. Dies gibt eine zuverlässige Methode zum Messen der verstrichenen Zeit zwischen zwei oder mehr Ereignissen.

```
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` ist in `System.Diagnostics` Sie müssen also `using System.Diagnostics;` hinzufügen `using System.Diagnostics;` in Ihre Datei.

IsHighResolution

- Die IsHighResolution-Eigenschaft gibt an, ob der Timer auf einem Leistungsindikator mit hoher Auflösung oder auf der DateTime-Klasse basiert.
- Dieses Feld ist schreibgeschützt.

```
// 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

Der von der Stoppuhrklasse verwendete Timer hängt von der Systemhardware und dem Betriebssystem ab. IsHighResolution ist true, wenn der Stoppuhr-Timer auf einem Leistungsindikator mit hoher Auflösung basiert. Andernfalls ist IsHighResolution false, was bedeutet, dass der Stoppuhr-Timer auf dem Systemzeitgeber basiert.

Ticks in Stoppuhr sind maschinen- / betriebssystemabhängig. Daher sollten Sie sich nicht darauf verlassen, dass Stoppuhr-Ticks in Sekunden zwischen zwei Systemen gleich sind und nach einem Neustart möglicherweise sogar auf demselben System. Daher können Sie niemals darauf zählen, dass Stoppuhr-Ticks dasselbe Intervall wie DateTime / TimeSpan-Ticks sind.

Um systemunabhängige Zeiten zu erhalten, verwenden Sie die Eigenschaften Elapsed oder ElapsedMilliseconds der Stoppuhr, bei denen die Stoppuhr.Frequenz (Ticks pro Sekunde) bereits berücksichtigt wird.

Die Stoppuhr sollte immer über Datetime für Timing-Prozesse verwendet werden, da sie leichter ist und mit Dateime arbeitet, wenn kein Leistungszähler mit hoher Auflösung verwendet werden kann.

Quelle

Stoppuhren online lesen: https://riptutorial.com/de/csharp/topic/3676/stoppuhren

Kapitel 125: String Escape-Sequenzen

Syntax

- \ '- einfaches Anführungszeichen (0x0027)
- "- doppelte Anführungszeichen (0x0022)
- \\ - Backslash (0x005C)
- \ 0 - null (0x0000)
- \ a - Warnung (0x0007)
- \ b - Rücktaste (0x0008)
- \ f - Formularvorschub (0x000C)
- \ n - neue Zeile (0x000A)
- \ r - Wagenrücklauf (0x000D)
- \ t - horizontale Registerkarte (0x0009)
- \ v - vertikale Registerkarte (0x000B)
- \ u0000 - \ uFFFF - Unicode-Zeichen
- \ x0 - \ xFFFF - Unicode-Zeichen (Code mit variabler Länge)
- \ U00000000 - \ U0010FFFF - Unicode-Zeichen (zum Erzeugen von Ersatzzeichen)

Bemerkungen

String-Escape-Sequenzen werden zur **Kompilierzeit** in das entsprechende Zeichen umgewandelt. Gewöhnliche Zeichenfolgen, die zufällige umgekehrte Schrägstriche enthalten, werden **nicht** transformiert.

Beispielsweise werden die Zeichenfolgen `notEscaped` und `notEscaped2` nicht in ein Zeilenvorschubzeichen umgewandelt, sondern bleiben zwei verschiedene Zeichen (`'\'` und `'n'`).

```
string escaped = "\n";
string notEscaped = "\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

Examples

Unicode-Zeichen-Escape-Sequenzen

```
string sqrt = "\u221A";      // √
string emoji = "\U0001F601"; // 
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

Sonderzeichen in Zeichenliteralen übergehen

Apostrophe

```
char apostrophe = '\'';
```

Backslash

```
char oneBackslash = '\\';
```

Sonderzeichen in String-Literalen übergehen

Backslash

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

Das zweite Beispiel verwendet ein wörtliches Stringliteral , das den Backslash nicht als Fluchtsymbol behandelt.

Zitate

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @"""Hello World!"", said the quick brown fox.";
```

Beide Variablen enthalten den gleichen Text.

> "Hallo Welt!", Sagte der schnelle braune Fuchs.

Zeilenumbrüche

Verbatim-String-Literale können Zeilenumbrüche enthalten:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Beide Variablen enthalten den gleichen Text.

Nicht erkannte Escape-Sequenzen erzeugen Fehler bei der Kompilierung

Die folgenden Beispiele werden nicht kompiliert:

```
string s = "\c";
char c = '\c';
```

Stattdessen erzeugen sie zur Kompilierungszeit die Fehler `Unrecognized escape sequence` .

Verwenden von Escape-Sequenzen in Bezeichnern

Escape-Sequenzen sind nicht auf `string` und `char` Literale beschränkt.

Angenommen, Sie müssen eine Drittanbieter-Methode überschreiben:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

Angenommen, das Zeichen `Œ` ist in der Zeichencodierung, die Sie für Ihre C # `Œ` nicht verfügbar. Sie Glück, ist es erlaubt , entweicht von der Art zu verwenden `\u####` oder `\U########` in ***Identifikatoren*** im Code. Es ist also legal zu schreiben:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

und der C # -Compiler weiß, dass `Œ` und `\u0152` dasselbe Zeichen sind.

(Es empfiehlt sich jedoch, auf UTF-8 oder eine ähnliche Kodierung umzuschalten, die alle Zeichen verarbeiten kann.)

String Escape-Sequenzen online lesen: https://riptutorial.com/de/csharp/topic/39/string-escape-sequenzen

Kapitel 126: String Interpolation

Syntax

- $ "content {expression} content"
- $ "content {expression: format} content"
- $ "Inhalt {Ausdruck} {{Inhalt in geschweiften Klammern}} Inhalt}"
- $ "Inhalt {Ausdruck: Format} {{Inhalt in Klammern}} Inhalt}"

Bemerkungen

String-Interpolation ist eine Abkürzung für die `string.Format()` -Methode, die das Erstellen von Strings mit Variablen- und Ausdruckswerten erleichtert.

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);  // returns "Hello, World"
var newWay = $"Hello, {name}!";                   // returns "Hello, World"
```

Examples

Ausdrücke

Vollständige Ausdrücke können auch in interpolierten Strings verwendet werden.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Live-Demo zu .NET-Geige

Datumsangaben in Strings formatieren

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

Sie können auch die `DateTime.ToString` Methode verwenden, um das `DateTime` Objekt zu formatieren. Dies erzeugt dieselbe Ausgabe wie der obige Code.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Ausgabe:

Es ist der 11. November 2015, wünsch dir was!

Live-Demo zu .NET-Geige

Live-Demo mit DateTime.ToString

> **Hinweis:** `MM` steht für Monate und `mm` für Minuten. Seien Sie vorsichtig, wenn Sie diese verwenden, da Fehler Fehler verursachen können, die möglicherweise schwer zu entdecken sind.

Einfache Verwendung

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

Hinter den Kulissen

Innerlich das

```
$"Hello, {name}!"
```

Wird wie folgt kompiliert:

```
string.Format("Hello, {0}!", name);
```

Auffüllen der Ausgabe

String kann so formatiert werden, dass ein Auffüllparameter akzeptiert wird, der angibt, wie viele Zeichenpositionen der eingefügte String verwenden soll:

```
${value, padding}
```

> **HINWEIS:** Positive Auffüllwerte geben die linke Auffüllung an und negative Auffüllwerte geben die rechte Auffüllung an.

Linke Polsterung

Ein linker Abstand von 5 (fügt vor dem Wert von number 3 Leerzeichen hinzu, sodass insgesamt 5 Zeichenpositionen in der resultierenden Zeichenfolge verwendet werden.)

```
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);
```

Ausgabe:

```
The answer to life, the universe and everything is    42.
```

Live-Demo zu .NET-Geige

Rechte Polsterung

Die rechte Auffüllung, die einen negativen Auffüllwert verwendet, fügt Leerzeichen am Ende des aktuellen Werts hinzu.

```
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);
```

Ausgabe:

```
The answer to life, the universe and everything is 42    .
```

Live-Demo zu .NET-Geige

Auffüllen mit Formatangaben

Sie können vorhandene Formatierungsbezeichner auch in Verbindung mit dem Auffüllen verwenden.

```
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 ";
//                                                          ^^^^^
```

Live-Demo zu .NET-Geige

Zahlen in Strings formatieren

Sie können einen Doppelpunkt und die numerische Standardformatsyntax verwenden, um zu steuern, wie Zahlen formatiert werden.

```
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"
```

Live-Demo zu .NET-Geige

String Interpolation online lesen: https://riptutorial.com/de/csharp/topic/22/string-interpolation

Kapitel 127: String.Format

Einführung

Die `Format` Methoden sind eine Gruppe von Überladungen in der `System.String` Klasse, die zum Erstellen von Zeichenfolgen verwendet werden, die Objekte in bestimmten Zeichenfolgendarstellungen kombinieren. Diese Informationen können auf `String.Format` , verschiedene `WriteLine` Methoden sowie andere Methoden im .NET-Framework angewendet werden.

Syntax

- string.Format (Zeichenkettenformat, params object [] args)
- string.Format (IFormatProvider-Anbieter, Zeichenfolgenformat, Params-Objekt [] args)
- $ "string {text} blablabla" // Seit C # 6

Parameter

Parameter	Einzelheiten
Format	Eine zusammengesetzte Formatzeichenfolge , die definiert, wie die *Argumente* in einer Zeichenfolge kombiniert werden sollen.
args	Eine Folge von Objekten, die zu einem String zusammengefasst werden sollen. Da dies ein `params` Argument verwendet, können Sie entweder eine durch Kommas getrennte Liste von Argumenten oder ein tatsächliches Objektarray verwenden.
Anbieter	Eine Sammlung von Möglichkeiten zum Formatieren von Objekten in Strings. Typische Werte sind CultureInfo.InvariantCulture und CultureInfo.CurrentCulture .

Bemerkungen

Anmerkungen:

- `String.Format()` verarbeitet `null` Argumente, ohne eine Ausnahme `String.Format()` .
- Es gibt Überladungen, die den Parameter `args` durch einen, zwei oder drei Objektparameter ersetzen.

Examples

Orte, an denen String.Format im Framework "eingebettet" ist

Es gibt mehrere Orte, an denen Sie `String.Format` *indirekt verwenden können* : Das Geheimnis besteht darin, die Überladung mit dem Signaturzeichenfolgenformat `string format, params object[] args` zu suchen, z. B .:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Kann durch eine kürzere Version ersetzt werden:

```
Console.WriteLine("{0} - {1}", name, value);
```

Es gibt andere Methoden, die auch `String.Format` zB:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

Benutzerdefiniertes Zahlenformat verwenden

`NumberFormatInfo` kann zum Formatieren von Ganzzahl- und `NumberFormatInfo` verwendet werden.

```
// 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);
```

Erstellen Sie einen benutzerdefinierten Formatanbieter

```
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;
    }
}
```

Verwendungszweck:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Ausgabe:

```
-> dlroW olleH <-
```

Links / rechts ausrichten, mit Leerzeichen auffüllen

Der zweite Wert in den geschweiften Klammern gibt die Länge der Ersatzzeichenfolge an. Durch Einstellen des zweiten Werts auf positiv oder negativ kann die Ausrichtung der Zeichenfolge geändert werden.

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Ausgabe:

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

Numerische Formate

```
// 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);
```

Ausgabe:

```
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%
```

Währungsformatierung

Der Formatbezeichner "c" (oder Währung) konvertiert eine Zahl in eine Zeichenfolge, die einen

Währungsbetrag darstellt.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

Präzision

Die Standardeinstellung ist 2. Verwenden Sie c1, c2, c3 usw., um die Genauigkeit zu steuern.

```
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
```

Währungszeichen

1. `CultureInfo` Instanz, um ein benutzerdefiniertes `CultureInfo` zu verwenden.

```
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. Verwenden Sie eine beliebige Zeichenfolge als Währungssymbol. Verwenden Sie `NumberFormatInfo` , um das Währungssymbol anzupassen.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

Position des Währungssymbols

Verwenden Sie CurrencyPositivePattern für positive Werte und CurrencyNegativePattern für negative Werte.

```
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 $
```

Negative Musterverwendung ist dasselbe wie positives Muster. Viel mehr Anwendungsfälle finden Sie im Original-Link.

Benutzerdefiniertes Dezimaltrennzeichen

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

Seit C # 6.0

6,0

Seit C # 6.0 ist es möglich, anstelle von `String.Format` String-Interpolation zu verwenden.

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Hallo John Doe!

Weitere Beispiele hierzu unter dem Thema C # 6.0-Features: Zeichenketteninterpolation .

Lockige geschweifte Klammern innerhalb eines String.Format () - Ausdrucks

```
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"
```

Datumsformatierung

```
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}");
```

Ausgabe :

```
06
Лхагва
06
```

Bezeichner	Bedeutung	Probe	Ergebnis
d	Datum	`{0:d}`	7/6/2016
dd	Tag, null gepolstert	`{0:dd}`	06
ddd	Kurzer Tagesname	`{0:ddd}`	Heiraten
dddd	Ganztägiger Name	`{0:dddd}`	Mittwoch
D	Langes Datum	`{0:D}`	Mittwoch, 6. Juli 2016
f	Vollständiges Datum und Uhrzeit, kurz	`{0:f}`	Mittwoch, 6. Juli 2016 18:30 Uhr
ff	Zweite Brüche, 2-stellig	`{0:ff}`	20
F f f	Zweite Brüche, 3-stellig	`{0:fff}`	201
ffff	Zweite Brüche, 4-stellig	`{0:ffff}`	2016
F	Volles Datum und volle Zeit	`{0:F}`	Mittwoch, 6. Juli 2016 18:30:14 PM
G	Standarddatum und -uhrzeit	`{0:g}`	06.07.2016 18:30 Uhr
gg	Epoche	`{0:gg}`	ANZEIGE
hh	Stunde (2-stellig, 12H)	`{0:hh}`	06
HH	Stunde (2-stellig, 24H)	`{0:HH}`	18
M	Monat und Tag	`{0:M}`	6. Juli
mm	Minuten, null aufgefüllt	`{0:mm}`	30
MM	Monat, null aufgefüllt	`{0:MM}`	07
MMM	3-stelliger Monatsname	`{0:MMM}`	Jul
MMMM	Vollständiger Monatsname	`{0:MMMM}`	Juli
ss	Sekunden	`{0:ss}`	14
r	RFC1123 Datum	`{0:r}`	Mi, 06. Jul 2016 18:30:14 GMT
s	Sortierbare Datumszeichenfolge	`{0:s}`	2016-07-06T18: 30: 14
t	Kurze Zeit	`{0:t}`	6:30 ABENDS
T	Lange Zeit	`{0:T}`	18:30:14 Uhr

Bezeichner	Bedeutung	Probe	Ergebnis
tt	AM / PM	`{0:tt}`	PM
u	Universelle sortierbare Ortszeit	`{0:u}`	2016-07-06 18: 30: 14Z
U	Universal GMT	`{0:U}`	Mittwoch, 6. Juli 2016 09:30:14 vormittags
Y	Monat und Jahr	`{0:Y}`	Juli 2016
yy	2 stelliges Jahr	`{0:yy}`	16
yyyy	4 stelliges Jahr	`{0:yyyy}`	2016
zz	2-stelliger Zeitzonen-Offset	`{0:zz}`	+09
zzz	Full Time Zone Offset	`{0:zzz}`	+09: 00

ToString ()

Die ToString () -Methode ist für alle Referenzobjekttypen vorhanden. Dies ist darauf zurückzuführen, dass alle Referenztypen von Object abgeleitet sind, das die ToString () - Methode enthält. Die ToString () -Methode der Objektbasisklasse gibt den Typnamen zurück. Das Fragment unten gibt "User" auf der Konsole aus.

```
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());
```

Die Klasse User kann jedoch ToString () überschreiben, um die zurückgegebene Zeichenfolge zu ändern. Das folgende Codefragment gibt "Id: 5, Name: User1" an die Konsole aus.

```
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());
```

Beziehung zu ToString ()

Während die `String.Format()` -Methode sicherlich zum Formatieren von Daten als Strings nützlich ist, kann es oft etwas übertrieben sein, insbesondere wenn ein einzelnes Objekt behandelt wird, wie unten gezeigt:

```
String.Format("{0:C}", money);  // yields "$42.00"
```

Ein einfacherer Ansatz besteht darin, einfach die `ToString()` Methode zu verwenden, die für alle Objekte in C # verfügbar ist. Es unterstützt alle dieselben Standard- und benutzerdefinierten Formatierungszeichenfolgen , erfordert jedoch nicht die erforderliche Parameterzuordnung, da es nur ein einziges Argument gibt:

```
money.ToString("C");  // yields "$42.00"
```

Einschränkungen und Formatierungsbeschränkungen

Während dieser Ansatz in einigen Szenarien möglicherweise einfacher ist, ist der `ToString()` Ansatz in Bezug auf das Hinzufügen von linkem oder rechtem Abstand wie in der `String.Format()` -Methode beschränkt:

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

Um dasselbe Verhalten mit der `ToString()` -Methode zu erreichen, müssen Sie eine andere Methode wie `PadLeft()` bzw. `PadRight()` verwenden:

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

String.Format online lesen: https://riptutorial.com/de/csharp/topic/79/string-format

Kapitel 128: StringBuilder

Examples

Was ist ein StringBuilder und wann wird er verwendet?

Ein `StringBuilder` repräsentiert eine Reihe von Zeichen, die im Gegensatz zu einer normalen Zeichenfolge veränderbar sind. Oft müssen Strings, die wir bereits erstellt haben, geändert werden, das Standard-String-Objekt kann jedoch nicht geändert werden. Dies bedeutet, dass jedes Mal, wenn eine Zeichenfolge geändert wird, ein neues Zeichenfolgenobjekt erstellt, kopiert und erneut zugewiesen werden muss.

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

Im obigen Beispiel hat `myString` anfangs nur den Wert `"Apples"` . Wenn wir jedoch "sind meine Lieblingsfrucht" verketten, muss die String-Klasse intern Folgendes tun:

- Erstellen eines neuen Array von Zeichen, das der Länge von `myString` und der neuen Zeichenfolge entspricht, die wir anhängen.
- Kopieren Sie alle Zeichen von `myString` an den Anfang unseres neuen Arrays und die neue Zeichenfolge an das Ende des Arrays.
- Erstellen Sie ein neues String-Objekt im Speicher und `myString` Sie es erneut `myString` .

Für eine einzelne Verkettung ist dies relativ trivial. Was aber, wenn nötig, um viele Anfügeoperationen in einer Schleife auszuführen?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

Durch das wiederholte Kopieren und Erstellen von Objekten wird die Leistung unseres Programms erheblich beeinträchtigt. Wir können dies vermeiden, indem wir stattdessen einen `StringBuilder` .

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Wenn nun dieselbe Schleife ausgeführt wird, ist die Leistung und Geschwindigkeit der Ausführungszeit des Programms wesentlich schneller als bei Verwendung einer normalen Zeichenfolge. Um den `StringBuilder` wieder in einen normalen String `ToString()` , können Sie einfach die `ToString()` Methode von `StringBuilder` aufrufen.

Dies ist jedoch nicht die einzige Optimierung, die `StringBuilder` bietet. Um die Funktionen weiter zu optimieren, können wir andere Eigenschaften nutzen, um die Leistung zu verbessern.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

Wenn wir im Voraus wissen, wie lange unser `StringBuilder` muss, können wir seine Größe vorab angeben, um zu verhindern, dass das Zeichen-Array intern geändert werden muss.

```
sb.Append('k', 2000);
```

Die Verwendung von `StringBuilder` zum Anhängen ist zwar viel schneller als eine Zeichenfolge, kann jedoch noch schneller ausgeführt werden, wenn Sie nur ein einziges Zeichen oft hinzufügen müssen.

Sobald Sie Ihren Zeichenfolge Bau abgeschlossen haben, können Sie die Verwendung `ToString()` Methode auf dem `StringBuilder` es zu einer grundlegenden konvertieren `string` . Dies ist häufig erforderlich, da die `StringBuilder` Klasse nicht von `string` erbt.

Zum Beispiel können Sie einen `StringBuilder` , um einen `string` zu erstellen:

```
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();
}
```

Fazit: `StringBuilder` sollte anstelle von string verwendet werden, wenn viele Änderungen an einer Zeichenfolge im Hinblick auf die Leistung vorgenommen werden müssen.

Verwenden Sie StringBuilder zum Erstellen von Zeichenfolgen aus einer großen Anzahl von Datensätzen

```
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();
}
```

StringBuilder online lesen: https://riptutorial.com/de/csharp/topic/4675/stringbuilder

Kapitel 129: String-Manipulation

Examples

Ändern der Groß- / Kleinschreibung von Zeichen in einem String

Die `System.String` Klasse unterstützt eine Reihe von Methoden zum Konvertieren zwischen Groß- und Kleinbuchstaben in einer Zeichenfolge.

- `System.String.ToLowerInvariant` wird verwendet, um ein in Kleinbuchstaben konvertiertes String-Objekt zurückzugeben.
- `System.String.ToUpperInvariant` wird verwendet, um ein in Großbuchstaben konvertiertes String-Objekt zurückzugeben.

Anmerkung: Der Grund für die Verwendung der *invarianten* Versionen dieser Methoden besteht darin, die Erzeugung unerwarteter kulturspezifischer Buchstaben zu verhindern. Dies wird hier ausführlich erklärt .

Beispiel:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Beachten Sie, dass Sie beim Konvertieren in Klein- und Großbuchstaben eine bestimmte **Kultur** angeben *können* , indem Sie die Methoden String.ToLower (CultureInfo) und String.ToUpper (CultureInfo) verwenden.

Einen String innerhalb eines Strings finden

Mit dem `System.String.Contains` Sie herausfinden, ob eine bestimmte Zeichenfolge in einer Zeichenfolge vorhanden ist. Die Methode gibt einen booleschen Wert zurück, true, wenn die Zeichenfolge vorhanden ist, andernfalls false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Mit der Methode `System.String.IndexOf` können Sie die Startposition einer Teilzeichenfolge innerhalb einer vorhandenen Zeichenfolge ermitteln.
Beachten Sie, dass die zurückgegebene Position nullbasiert ist. Wenn der Teilstring nicht gefunden wird, wird der Wert -1 zurückgegeben.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

Verwenden Sie die Methode `System.String.LastIndexOf` um den ersten Speicherort am ***Ende*** einer Zeichenfolge zu finden:

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Leerzeichen aus einer Zeichenfolge entfernen (entfernen)

Mit der `System.String.Trim` Methode können alle führenden und `System.String.Trim` Leerzeichen aus einer Zeichenfolge entfernt werden:

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

In Ergänzung:

- Um Leerzeichen nur vom *Anfang* eines Strings zu entfernen, verwenden Sie: `System.String.TrimStart`
- Um Leerzeichen nur vom *Ende* eines Strings zu entfernen, verwenden Sie: `System.String.TrimEnd`

Teilstring, um einen Teil einer Zeichenfolge zu extrahieren.

Mit der Methode `System.String.Substring` kann ein Teil der Zeichenfolge extrahiert werden.

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

Ersetzen eines Strings innerhalb eines Strings

Mit der Methode `System.String.Replace` können Sie einen Teil einer Zeichenfolge durch eine andere Zeichenfolge ersetzen.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Alle Vorkommen der Suchzeichenfolge werden ersetzt:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` kann auch zum *Entfernen eines* Teils einer Zeichenfolge verwendet werden, indem eine leere Zeichenfolge als Ersetzungswert angegeben wird:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Aufteilen einer Zeichenfolge mit einem Trennzeichen

Verwenden Sie die `System.String.Split` Methode, um ein Zeichenfolgenarray zurückzugeben, das `System.String.Split` der ursprünglichen Zeichenfolge enthält, die basierend auf einem angegebenen Trennzeichen aufgeteilt wird:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Ausgabe:

Ein
Zwei
Drei
Vier

Verketten Sie ein String-Array zu einem String

Die `System.String.Join` Methode ermöglicht die Verkettung aller Elemente in einem String-Array unter Verwendung eines angegebenen Trennzeichens zwischen jedem Element:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

String-Verkettung

Zeichenfolgenverkettung kann mithilfe der Methode `System.String.Concat` oder (viel einfacher) mit dem Operator `+` werden:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

String-Manipulation online lesen: https://riptutorial.com/de/csharp/topic/3599/string-manipulation

Kapitel 130: String-Verkettung

Bemerkungen

Wenn Sie eine dynamische Zeichenfolge erstellen, empfiehlt es sich, die `StringBuilder` Klasse zu `Concat` , anstatt die Zeichenfolgen mithilfe der + oder `Concat` Methode zu verbinden, da jedes + / `Concat` bei jeder `Concat` ein neues Zeichenfolgenobjekt erstellt.

Examples

+ Operator

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Verketten Sie Zeichenfolgen mit System.Text.StringBuilder

Das Verketten von Zeichenfolgen mit einem StringBuilder kann Leistungsvorteile gegenüber der einfachen Verkettung von Zeichenfolgen mit `+` bieten. Dies liegt an der Speicherzuordnung. Zeichenfolgen werden bei jeder Verkettung neu zugewiesen, StringBuilders reservieren Speicher nur in Blöcken, wenn der aktuelle Block erschöpft ist. Dies kann einen großen Unterschied machen, wenn viele kleine Verkettungen durchgeführt werden.

```
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 "
```

Aufrufe von `Append()` können verkettet werden, da ein Verweis auf den `StringBuilder` :

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

Concat-String-Array-Elemente mit String.Join

Mit der `String.Join` Methode können mehrere Elemente aus einem String-Array verkettet werden.

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
```

```
Console.WriteLine(result);
```

Erzeugt die folgende Ausgabe: "Orange, Traube"

In diesem Beispiel wird die `String.Join(String, String[], Int32, Int32)` , die den `String.Join(String, String[], Int32, Int32)` und die Anzahl über dem Trennzeichen und Wert angibt.

Wenn Sie den startIndex und die Anzahl der Überladungen nicht verwenden möchten, können Sie alle angegebenen Zeichenfolgen verknüpfen. So was:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

die produzieren wird;

Apfel, Orange, Traube, Birne

Verkettung zweier Zeichenketten mit $

$ bietet eine einfache und prägnante Methode zum Verketten mehrerer Zeichenfolgen.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

String-Verkettung online lesen: https://riptutorial.com/de/csharp/topic/3616/string-verkettung

Kapitel 131: Strom

Examples

Streams verwenden

Ein Stream ist ein Objekt, das eine einfache Übertragung von Daten ermöglicht. Sie selbst fungieren nicht als Datencontainer.

Die Daten, mit denen wir arbeiten, haben die Form eines Byte-Arrays (`byte []`). Die Funktionen zum Lesen und Schreiben sind alle byteorientiert, zB `WriteByte()` .

Es gibt keine Funktionen für den Umgang mit ganzen Zahlen, Strings usw. Dies macht den Stream sehr universell, aber weniger einfach zu handhaben, wenn Sie beispielsweise nur Text übertragen möchten. Streams können besonders hilfreich sein, wenn Sie mit großen Datenmengen arbeiten.

Wir müssen verschiedene Arten von Streams verwenden, von denen aus geschrieben / gelesen werden muss (z. B. der Hintergrundspeicher). Wenn die Quelle beispielsweise eine Datei ist, müssen wir `FileStream` :

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

In ähnlicher Weise wird `MemoryStream` verwendet, wenn der Sicherungsspeicher Arbeitsspeicher ist:

```
// 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...
}
```

In ähnlicher Weise wird `System.Net.Sockets.NetworkStream` für den Netzwerkzugriff verwendet.

Alle Streams werden von der generischen Klasse `System.IO.Stream` . Daten können nicht direkt aus Streams gelesen oder geschrieben werden. Das .NET Framework bietet `StreamReader` wie `StreamReader` , `StreamWriter` , `BinaryReader` und `BinaryWriter` , die zwischen `BinaryWriter` Typen und der Low-Level-Stream-Schnittstelle konvertieren und die Daten für Sie in den oder aus dem Stream übertragen.

Das Lesen und Schreiben von Streams kann über `StreamReader` und `StreamWriter` . Beim Schließen sollte man vorsichtig sein. Standardmäßig wird auch der enthaltene Stream geschlossen und für die weitere Verwendung unbrauchbar. Dieses Standardverhalten kann geändert werden, indem ein Konstruktor verwendet wird, der über `bool leaveOpen` Parameter `bool leaveOpen` und seinen Wert auf `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();
}
```

Da Klassen `Stream` , `StreamReader` , `StreamWriter` usw. die `IDisposable` Schnittstelle implementieren, können wir die `Dispose()` -Methode für Objekte dieser Klassen aufrufen.

Strom online lesen: https://riptutorial.com/de/csharp/topic/3114/strom

Kapitel 132: Structs

Bemerkungen

Im Gegensatz zu Klassen ist eine `struct` ein `struct` , *der standardmäßig* auf dem lokalen Stapel und nicht auf dem verwalteten Heap erstellt *wird* . Dies bedeutet, dass die `struct` wird, sobald der bestimmte Stapel den Gültigkeitsbereich `struct` . Enthaltene Referenztypen von nicht zugeordneten `struct` werden ebenfalls überstrichen, sobald der GC feststellt, dass sie nicht mehr von der `struct` referenziert werden.

`struct` s können nicht erben und können keine Grundlage für die Vererbung sein, sie sind implizit versiegelt und können auch keine `protected` Mitglieder enthalten. Eine `struct` kann jedoch eine Schnittstelle wie Klassen implementieren.

Examples

Struktur deklarieren

```
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` Instanzfelder können über einen parametrisierten Konstruktor oder einzeln nach der `struct` werden.
- Private Member können nur vom Konstruktor initialisiert werden.
- `struct` definiert einen versiegelten Typ, der implizit von System.ValueType erbt.
- Strukturen können nicht von einem anderen Typ erben, sie können jedoch Schnittstellen implementieren.
- Strukturen werden bei der Zuweisung kopiert. Das bedeutet, dass alle Daten in die neue Instanz kopiert werden und Änderungen an einer Instanz nicht von der anderen Instanz übernommen werden.

- Eine Struktur kann nicht `null` , obwohl es als ein Nullable Type verwendet *werden:*

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Strukturen können mit oder ohne Verwendung des `new` Operators instanziiert werden.

```
//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;
```

  Der `new` Operator muss jedoch verwendet werden, um einen Initialisierer verwenden zu können:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

Eine Struktur kann alles deklarieren, was eine Klasse mit wenigen Ausnahmen deklarieren kann:

- Eine Struktur kann keinen parameterlosen Konstruktor deklarieren. `struct` Instanzfelder können über einen parametrisierten Konstruktor oder einzeln nach der `struct` werden. Private Member können nur vom Konstruktor initialisiert werden.
- Eine Struktur kann Mitglieder nicht als geschützt deklarieren, da sie implizit versiegelt ist.
- Strukturfelder können nur initialisiert werden, wenn sie const oder static sind.

Strukturverwendung

Mit Konstruktor:

```
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);
```

Ohne Konstruktor:

```
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;
```

Wenn wir eine Struktur mit ihrem Konstruktor verwenden, haben wir keine Probleme mit einem nicht zugewiesenen Feld (jedes nicht zugewiesene Feld hat einen Nullwert).

Im Gegensatz zu Klassen muss eine Struktur nicht erstellt werden. Das neue Schlüsselwort muss also nicht verwendet werden, es sei denn, Sie müssen einen der Konstruktoren aufrufen. Eine Struktur erfordert kein neues Schlüsselwort, da es sich um einen Werttyp handelt und daher nicht null sein kann.

Struktur implementierende Schnittstelle

```
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;
    }
}
```

Strukturen werden bei der Zuweisung kopiert

Sinse-Strukturen sind Werttypen, bei denen alle Daten bei der Zuweisung *kopiert* werden. Durch Änderungen an der neuen Kopie werden die Daten für die ursprüngliche Kopie nicht geändert. Der nachstehende `p1` zeigt, dass `p1` in `p2` *kopiert wird* und Änderungen an `p1` die `p2` Instanz nicht beeinflussen.

```
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
```

Structs online lesen: https://riptutorial.com/de/csharp/topic/778/structs

Kapitel 133: Strukturelle Entwurfsmuster

Einführung

Strukturelle Entwurfsmuster sind Muster, die beschreiben, wie Objekte und Klassen kombiniert werden können, um eine große Struktur zu bilden, und die das Design erleichtern, indem eine einfache Methode zum Verwalten von Beziehungen zwischen Entitäten identifiziert wird. Es werden sieben strukturelle Muster beschrieben. Sie lauten wie folgt: Adapter, Brücke, Composite, Dekorateur, Fassade, Fliegengewicht und Stellvertreter

Examples

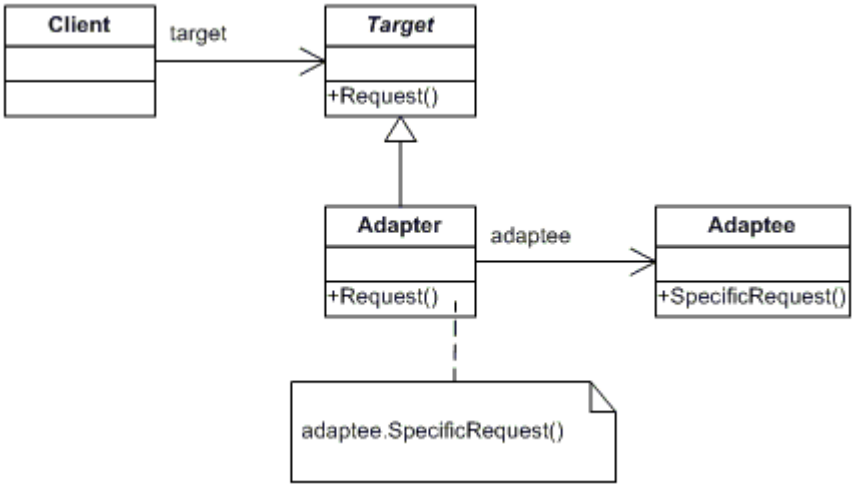
Adapter Design Pattern

> **„Adapter“ ist,** wie der Name schon sagt, das Objekt, mit dem zwei miteinander inkompatible Schnittstellen miteinander kommunizieren können.
>
> **Zum Beispiel:** Wenn Sie ein Iphone 8 (oder ein anderes Produkt von Apple) kaufen, benötigen Sie viele Adapter. Weil die Standardschnittstelle kein Audio oder USB unterstützt. Mit diesen Adaptern können Sie Kopfhörer mit Kabel oder ein normales Ethernet-Kabel verwenden. *"Zwei miteinander inkompatible Schnittstellen kommunizieren also miteinander"* .
>
> **In technischer Hinsicht bedeutet dies:** Konvertieren Sie die Schnittstelle einer Klasse in eine andere Schnittstelle, die ein Client erwartet. Adapter lassen Klassen zusammenarbeiten, die sonst wegen inkompatibler Schnittstellen nicht möglich wären. Die Klassen und Objekte, die an diesem Muster teilnehmen, sind:

Das Adaptermuster verlässt 4 Elemente

1. **ITarget:** Dies ist die Schnittstelle, über die der Client die Funktionalität erreicht.
2. **Anpassung:** Dies ist die Funktionalität, die der Client wünscht, seine Schnittstelle ist jedoch nicht mit dem Client kompatibel.
3. **Client:** Dies ist die Klasse, die mit dem Code des Adaptees einige Funktionalität erreichen möchte.
4. **Adapter:** Dies ist die Klasse, die ITarget implementiert und den Adaptee-Code aufrufen würde, den der Client aufrufen möchte.

UML



Erstes Code-Beispiel (Theoretisches Beispiel) .

```
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();
    }
}
```

Zweites Codebeispiel (Realimitation)

```
/// <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();
    }
}
```

Wann verwenden?

- Zulassen, dass ein System Klassen eines anderen Systems verwendet, die nicht mit ihm kompatibel sind.
- Erlauben Sie die Kommunikation zwischen neuen und bereits bestehenden Systemen, die voneinander unabhängig sind
- Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter sind das beste Beispiel für das Adaptermuster.

Strukturelle Entwurfsmuster online lesen: https://riptutorial.com/de/csharp/topic/9764/strukturelle-entwurfsmuster

Kapitel 134: Synchronisierungskontext in Async-Await

Examples

Pseudocode für async / await-Schlüsselwörter

Betrachten Sie eine einfache asynchrone Methode:

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

Vereinfachend können wir sagen, dass dieser Code eigentlich Folgendes bedeutet:

```
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;
}
```

Das bedeutet, dass `async` / `await` Schlüsselwörter den aktuellen Synchronisationskontext verwenden, sofern vorhanden. Sie können also Bibliothekscode schreiben, der in UI-, Web- und Konsolenanwendungen ordnungsgemäß funktionieren würde.

Quellartikel .

Synchronisierungskontext deaktivieren

Um den Synchronisationskontext zu deaktivieren, rufen Sie die `ConfigureAwait` Methode auf:

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

. . .
```

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

> Mit ConfigureAwait können Sie das standardmäßige Aufnahmeverhalten von SynchronizationContext vermeiden. Durch das Übergeben von false für den Parameter flowContext wird verhindert, dass SynchronizationContext verwendet wird, um die Ausführung nach dem Erwarten wieder aufzunehmen.

Zitat von Es ist alles über den SynchronizationContext .

Warum ist SynchronizationContext so wichtig?

Betrachten Sie dieses Beispiel:

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

Diese Methode friert die UI-Anwendung ein, bis `RunTooLong` abgeschlossen ist. Die Anwendung reagiert nicht.

Sie können versuchen, den inneren Code asynchron auszuführen:

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

Dieser Code wird jedoch nicht ausgeführt, da der innere Körper möglicherweise auf einem Thread außerhalb der Benutzeroberfläche ausgeführt wird und die Eigenschaften der Benutzeroberfläche nicht direkt geändert werden sollten :

```
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;
    });
}
```

Vergessen Sie nicht, immer dieses Muster zu verwenden. Oder versuchen Sie es mit `SynchronizationContext.Post` , das es für Sie machen wird:

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

```
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
        }, null);
    });
}
```

Synchronisierungskontext in Async-Await online lesen:
https://riptutorial.com/de/csharp/topic/7381/synchronisierungskontext-in-async-await

Kapitel 135: System.DirectoryServices.Protocols.LdapConnec

Examples

Authentifizierte SSL-LDAP-Verbindung, SSL-Zertifikat stimmt nicht mit Reverse-DNS überein

Richten Sie einige Konstanten für die Server- und Authentifizierungsinformationen ein. Angenommen, LDAPv3, aber das lässt sich leicht ändern.

```
// 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";
```

Erstellen Sie die Verbindung tatsächlich mit drei Teilen: einem LdapDirectoryIdentifier (dem Server) und NetworkCredentials.

```
// 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));
```

Verwenden Sie den LDAP-Server, z. B. suchen Sie nach einer Person für alle objectClass-Werte. Die objectClass dient zur Veranschaulichung einer zusammengesetzten Suche: Das kaufmännische Und ist der Boolesche Operator "und" für die beiden Abfrageklauseln.

```
 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
}
```

Super einfaches anonymes LDAP

Angenommen, LDAPv3, aber das lässt sich leicht ändern. Dies ist eine anonyme, unverschlüsselte LDAPv3-LdapConnection-Erstellung.

```
private const string TargetServer = "ldap.example.com";
```

Erstellen Sie die Verbindung tatsächlich mit drei Teilen: einem LdapDirectoryIdentifier (dem Server) und NetworkCredentials.

```
// 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
    }
};
```

Um die Verbindung zu nutzen, würde so etwas Leute mit dem Nachnamen Smith bekommen

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

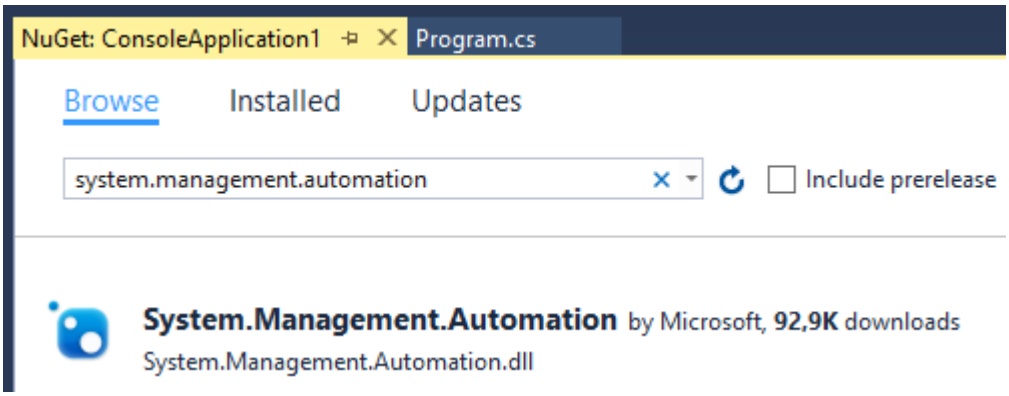
System.DirectoryServices.Protocols.LdapConnection online lesen:
https://riptutorial.com/de/csharp/topic/5177/system-directoryservices-protocols-ldapconnection

Kapitel 136: System.Management.Automation

Bemerkungen

> Der *System.Management.Automation*- Namespace ist der *Stammnamespace* für Windows PowerShell.

System.Management.Automation ist eine Erweiterungsbibliothek von Microsoft. Sie kann Visual Studio-Projekten über den NuGet-Paket-Manager oder die Paket-Manager-Konsole hinzugefügt werden.



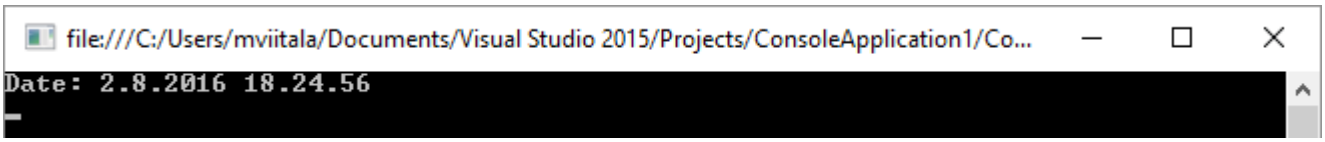
```
PM> Install-Package System.Management.Automation
```

Examples

Einfache synchrone Pipeline aufrufen

Holen Sie sich das aktuelle Datum und die aktuelle Uhrzeit.

```
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();
    }
}
```



System.Management.Automation online lesen: https://riptutorial.com/de/csharp/topic/4988/system-management-automation

Kapitel 137: T4-Codegenerierung

Syntax

- **T4-Syntax**
- `<#@...#>` // Deklarieren von Eigenschaften einschließlich Vorlagen, Assemblys und Namespaces und der Sprache, die die Vorlage verwendet
- `Plain Text` // Deklariert Text, der für die generierten Dateien durchgeschleift werden kann
- `<#=...#>` // Deklarieren von Skripten
- `<#+...#>` // Scriptlets deklarieren
- `<#...#>` // Textblöcke deklarieren

Examples

Laufzeitcode-Generierung

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

T4-Codegenerierung online lesen: https://riptutorial.com/de/csharp/topic/4824/t4-codegenerierung

Kapitel 138: Task Parallele Bibliothek

Examples

Parallel.ForEach

Ein Beispiel, das die Parallel.ForEach-Schleife verwendet, um ein bestimmtes Array von Website-URLs zu pingen.

```
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.für

Ein Beispiel, in dem die Parallel.For-Schleife zum Pingen eines angegebenen Arrays von Website-URLs verwendet wird.

```
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.Einruf

Paralleles Aufrufen von Methoden oder Aktionen (Paralleler Bereich)

```
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));
    }
}
```

Eine async-stornierbare Abrufaufgabe, die zwischen Iterationen wartet

```
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);
        }
    }
}
```

Eine stornierbare Abfrageaufgabe mit der CancellationTokenSource

```
public class Foo
{
    private CancellationTokenSource _cts;

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

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

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

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

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

Async-Version von PingUrl

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

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

        var result = await ping.SendPingAsync(url);

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

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

        return response;
    }
```

Task Parallele Bibliothek online lesen: https://riptutorial.com/de/csharp/topic/1010/task-parallele-bibliothek

Kapitel 139: Teilklasse und Methoden

Einführung

Partial classes bietet uns die Möglichkeit, Klassen in mehrere Teile und in mehrere Quelldateien aufzuteilen. Alle Teile werden während der Kompilierzeit zu einer einzigen Klasse zusammengefasst. Alle Teile sollten das Schlüsselwort `partial` und sollten die gleiche Zugänglichkeit haben. Alle Teile sollten in derselben Assembly vorhanden sein, damit sie während der Kompilierzeit aufgenommen werden kann.

Syntax

- public **partial** class MyPartialClass {}

Bemerkungen

- Teilklassen müssen in derselben Assembly und in demselben Namensraum als die Klasse definiert werden, die sie erweitern.
- Alle Teile der Klasse müssen das `partial` Schlüsselwort verwenden.
- Alle Teile der Klasse müssen dieselbe Zugänglichkeit haben. `public` / `protected` / `private` etc ..
- Wenn ein Teil das `abstract` Schlüsselwort verwendet, wird der kombinierte Typ als abstrakt betrachtet.
- Wenn ein Teil das `sealed` Schlüsselwort verwendet, gilt der kombinierte Typ als versiegelt.
- Wenn ein Teil einen Basistyp verwendet, erbt der kombinierte Typ von diesem Typ.
- Der kombinierte Typ erbt alle für alle Teilklassen definierten Schnittstellen.

Examples

Teilklassen

Partielle Klassen bieten die Möglichkeit, Klassendeklarationen aufzuteilen (normalerweise in separate Dateien). Ein häufiges Problem, das mit partiellen Klassen gelöst werden kann, besteht darin, dass Benutzer automatisch generierten Code ändern können, ohne befürchten zu müssen, dass ihre Änderungen überschrieben werden, wenn der Code erneut generiert wird. Auch mehrere Entwickler können mit derselben Klasse oder denselben Methoden arbeiten.

```
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."
        }
    }
}
```

Teilmethoden

Die Teilmethode besteht aus der Definition in einer Teilklassendeklaration (als allgemeines Szenario - in der automatisch generierten) und der Implementierung in einer anderen Teilklassendeklaration.

```
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."
```

```
        }
    }
}
```

Teilklassen, die von einer Basisklasse erben

Bei der Vererbung von einer Basisklasse muss nur für eine Teilklasse die Basisklasse angegeben werden.

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

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

Sie *können dieselbe* Basisklasse in mehreren Teilklassen angeben. Es wird von einigen IDE-Tools als redundant gekennzeichnet, aber es wird korrekt kompiliert.

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

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

Sie *können keine unterschiedlichen* Basisklassen in mehreren Teilklassen angeben. Dies führt zu einem Compiler-Fehler.

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

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

Teilklasse und Methoden online lesen: https://riptutorial.com/de/csharp/topic/3674/teilklasse-und-methoden

Kapitel 140: Timer

Syntax

- `myTimer.Interval` - `myTimer.Interval` wie oft das "Tick" `myTimer.Interval` (in Millisekunden) aufgerufen wird
- `myTimer.Enabled` - boolescher Wert, mit dem der Timer aktiviert / deaktiviert wird
- `myTimer.Start()` - Startet den Timer.
- `myTimer.Stop()` - Stoppt den Timer.

Bemerkungen

Bei Verwendung von Visual Studio können Timer aus der Toolbox direkt als Steuerelement Ihrem Formular hinzugefügt werden.

Examples

Multithread-Timer

`System.Threading.Timer` - Einfachster Multithread-Timer. Enthält zwei Methoden und einen Konstruktor.

Beispiel: Ein Timer ruft die DataWrite-Methode auf, die nach Ablauf von fünf Sekunden "Multithread ausgeführt ..." schreibt, und danach jede Sekunde, bis der Benutzer die Eingabetaste drückt:

```
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..."
  }
}
```

Hinweis: Wird einen separaten Abschnitt zum Entsorgen von Multithread-Timern bereitstellen.

`Change` - Diese Methode kann aufgerufen werden, wenn Sie das Timerintervall ändern möchten.

`Timeout.Infinite` - Wenn Sie nur einmal feuern möchten. Geben Sie dies im letzten Argument des Konstruktors an.

`System.Timers` - Eine andere von .NET Framework bereitgestellte `System.Timers` . Es `System.Threading.Timer` den `System.Threading.Timer` .

Eigenschaften:

- `IComponent` - Ermöglicht das `IComponent` in der Komponentenleiste des Designer von Visual Studio
- `Interval` statt einer `Change` Methode
- `Elapsed event` statt eines Callback - `delegate`
- `Enabled` Eigenschaft zum Starten und Stoppen des Timers (`default value = false`)
- `Start` & `Stop` Methoden für den Fall, dass Sie durch die `Enabled` Eigenschaft (über Punkt) verwirrt werden
- `AutoReset` - zum Anzeigen eines wiederkehrenden Ereignisses (`default value = true`)
- `SynchronizingObject` Eigenschaft mit `Invoke` und `BeginInvoke` Methoden für die sichere Methoden auf WPF - Elemente aufrufen und Windows Forms - Steuerelemente

Beispiel für alle oben genannten Funktionen:

```
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` - Verwenden Sie den Thread-Pool, um einigen Threads zu ermöglichen, viele Timer zu bedienen. Das bedeutet, dass die Callback-Methode oder das `Elapsed` Ereignis bei jedem Aufruf eines anderen Threads ausgelöst werden kann.

`Elapsed` - Dieses Ereignis wird immer rechtzeitig `Elapsed` - unabhängig davon, ob die Ausführung des vorherigen `Elapsed` Ereignisses abgeschlossen ist. Aus diesem Grund müssen Rückrufe oder Ereignishandler threadsicher sein. Die Genauigkeit von Multithread-Timern hängt vom

Betriebssystem ab und beträgt normalerweise 10–20 ms.

`interop` - Wenn Sie eine höhere Genauigkeit benötigen, verwenden Sie diese `interop` und rufen Sie den Windows-Multimedia-Timer auf. Dies hat eine Genauigkeit von bis zu 1 ms und ist in `winmm.dll` definiert.

`timeBeginPeriod` - Rufen Sie diese `timeBeginPeriod` zuerst auf, um das Betriebssystem darüber zu informieren, dass Sie eine hohe Timing-Genauigkeit benötigen

`timeSetEvent` - Rufen Sie nach `timeBeginPeriod` diese `timeBeginPeriod` auf, um einen Multimedia-Timer zu starten.

`timeKillEvent` - rufe dies auf, wenn du fertig bist, und der Timer wird `timeKillEvent`

`timeEndPeriod` - Rufen Sie dies auf, um das Betriebssystem darüber zu informieren, dass Sie keine hohe Timing-Genauigkeit mehr benötigen.

Sie finden vollständige Beispiele im Internet, die den Multimedia-Timer verwenden, indem Sie nach den Schlüsselwörtern `dllimport winmm.dll timesetevent` .

Instanz eines Timers erstellen

Zeitgeber werden verwendet, um Aufgaben in bestimmten Zeitabständen auszuführen (Do X alle Y Sekunden). Nachfolgend ein Beispiel zum Erstellen einer neuen Instanz eines Zeitgebers.

HINWEIS : Dies gilt für Zeitgeber, die WinForms verwenden. Wenn Sie WPF verwenden, möchten Sie vielleicht einen Blick auf `DispatcherTimer` werfen

```
    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();
        }

    }
```

Zuweisen des "Tick" -Ereignishandlers zu einem Timer

Alle Aktionen, die in einem Timer ausgeführt werden, werden im Ereignis "Tick" behandelt.

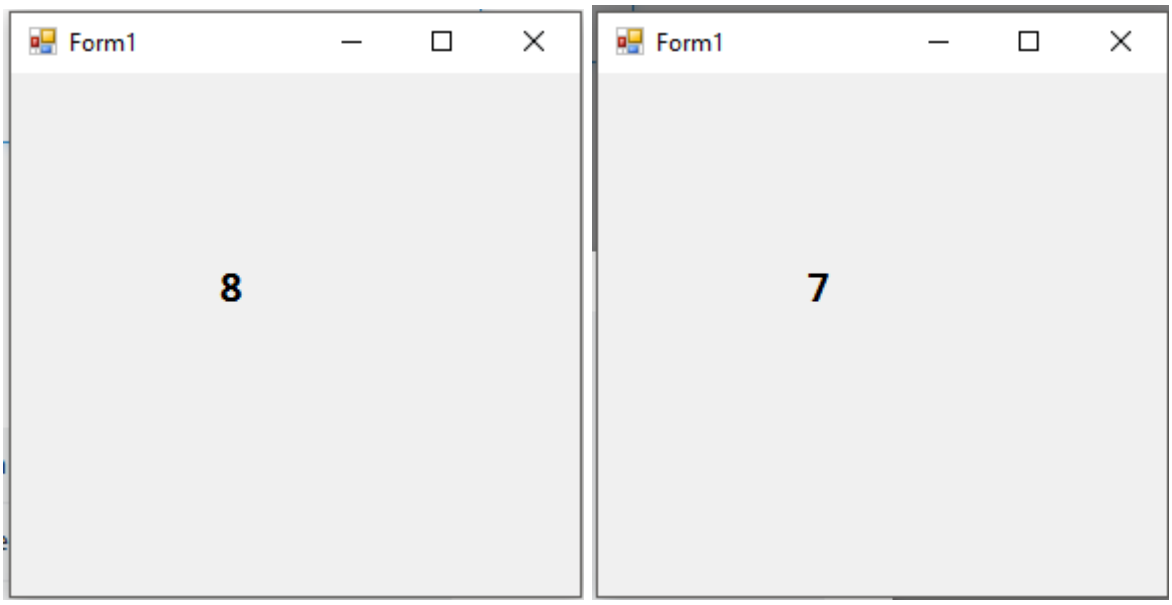
```
public partial class Form1 : Form
{

    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.
    }
}
```

Beispiel: Einen Timer verwenden, um einen einfachen Countdown auszuführen.

```
    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();
            }
        }
    }
```

Ergebnisse in ...



Und so weiter...

Timer online lesen: https://riptutorial.com/de/csharp/topic/3829/timer

Kapitel 141: Tuples

Examples

Tupel erstellen

Tupel werden mit den generischen Typen `Tuple<T1>` - `Tuple<T1,T2,T3,T4,T5,T6,T7,T8>` . Jeder der Typen stellt ein Tupel dar, das 1 bis 8 Elemente enthält. Elemente können unterschiedlichen Typs sein.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Tupel können auch mit statischen `Tuple.Create` Methoden erstellt werden. In diesem Fall werden die Typen der Elemente vom C # -Compiler abgeleitet.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7,0

Seit C # 7.0 können Tupel einfach mit ValueTuple erstellt werden.

```
var tuple = ("foo", 123, true, new MyClass());
```

Elemente können zur leichteren Zerlegung benannt werden.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Zugriff auf Tupel-Elemente

Um auf Tupel-Elemente `Item1` `Item8` Eigenschaften von `Item1` - `Item8` . Es werden nur die Eigenschaften verfügbar sein, deren Indexnummer kleiner oder gleich der `Item3` (dh man kann nicht auf `Item3` Eigenschaft `Item3` in `Tuple<T1,T2>` zugreifen).

```
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()
```

Vergleich und Sortierung von Tupeln

Tupel können anhand ihrer Elemente verglichen werden.

Beispielsweise kann eine Aufzählungsliste, deren Elemente vom Typ `Tuple` sind, anhand von

Vergleichsoperatoren sortiert werden, die für ein bestimmtes Element definiert sind:

```
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)
```

Oder um die Sortierung umzukehren:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Gibt mehrere Werte aus einer Methode zurück

Tupel können verwendet werden, um mehrere Werte aus einer Methode zurückzugeben, ohne out-Parameter zu verwenden. Im folgenden Beispiel wird `AddMultiply` verwendet, um zwei Werte (Summe, Produkt) zurückzugeben.

```
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);
}
```

Ausgabe:

53
700

Jetzt bietet C # 7.0 eine alternative Methode, um mehrere Werte aus Methoden mithilfe von `ValueTuple` . Weitere Informationen zur `ValueTuple` Struktur .

Tuples online lesen: https://riptutorial.com/de/csharp/topic/838/tuples

Kapitel 142: Typumwandlung

Bemerkungen

Die Typkonvertierung konvertiert einen Datentyp in einen anderen Typ. Es wird auch als Type Casting bezeichnet. In C # hat Typ Casting zwei Formen:

Implizite Typkonvertierung - Diese Konvertierungen werden von C # typsicher durchgeführt. Beispielsweise sind Konvertierungen von kleineren zu größeren ganzzahligen Typen und Konvertierungen von abgeleiteten Klassen in Basisklassen.

Explizite Typkonvertierung - Diese Konvertierungen werden explizit von Benutzern mit vordefinierten Funktionen ausgeführt. Explizite Konvertierungen erfordern einen Besetzungsoperator.

Examples

Beispiel für einen impliziten MSDN-Operator

```
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();
    }
}
```

Ausgabe:

Ziffer für doppelte implizite Konvertierung aufgerufen
double to digit implict Konvertierung aufgerufen
num = 7 dig2 = 12

Live-Demo zu .NET-Geige

Explizite Typkonvertierung

```
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();
      }
   }
}
```

Typumwandlung online lesen: https://riptutorial.com/de/csharp/topic/3489/typumwandlung

Kapitel 143: Überlastauflösung

Bemerkungen

Der Prozess der Überlastauflösung wird in der C # -Spezifikation , Abschnitt 7.5.3 beschrieben. Ebenfalls relevant sind die Abschnitte 7.5.2 (Typeninferenz) und 7.6.5 (Aufrufausdrücke).

Wie die Überladungslösung funktioniert, wird wahrscheinlich in C # 7 geändert. Die Designhinweise weisen darauf hin, dass Microsoft ein neues System einführen wird, um festzustellen, welche Methode die bessere ist (in komplizierten Szenarien).

Examples

Beispiel für ein grundlegendes Überladen

Dieser Code enthält eine überladene Methode namens **Hello** :

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Wenn die **Main-** Methode aufgerufen wird, wird sie gedruckt

```
int
double
```

Wenn der Compiler zum Zeitpunkt der Kompilierung den Methodenaufruf `Hello(0)` , findet er alle Methoden mit dem Namen `Hello` . In diesem Fall findet sie zwei davon. Dann versucht sie herauszufinden, welche der Methoden *besser ist* . Der Algorithmus zum Bestimmen, welche Methode besser ist, ist komplex, aber es läuft darauf hinaus, "möglichst wenig implizite Konvertierungen wie möglich zu machen".

Im Fall von `Hello(0)` ist daher für die Methode `Hello(int)` keine Konvertierung erforderlich, für die Methode `Hello(double)` ist jedoch eine implizite numerische Konvertierung erforderlich. Daher wird

die erste Methode vom Compiler ausgewählt.

Im Fall von `Hello(0.0)` gibt es keine Möglichkeit, `0.0` implizit in ein `int` zu konvertieren, sodass die Methode `Hello(int)` nicht einmal für die Überlastauflösung in Betracht gezogen wird. Nur die Methode bleibt übrig und wird vom Compiler ausgewählt.

"params" wird nicht erweitert, sofern dies nicht erforderlich ist.

Das folgende Programm:

```
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);
    }
}
```

wird drucken:

```
5
two
3
```

Die `Method(objectArray)` kann auf zwei Arten interpretiert werden: Ein einzelnes `Object` Argument, das zufällig ein Array ist (das Programm würde also `1` ausgeben, da dies die Anzahl der Argumente oder ein Array von Argumenten wäre, die in der angegeben werden Normalform, als ob die Methode `Method` nicht das Schlüsselwort `params` In diesen Situationen hat die normale, nicht expandierte Form immer Vorrang. Das Programm gibt also `5` .

Im zweiten Ausdruck, `Method(objectArray, objectArray)` , sind sowohl die erweiterte Form der ersten als auch die traditionelle zweite Methode anwendbar. Auch in diesem Fall haben nicht expandierte Formulare Vorrang, daher druckt das Programm `two` .

Im dritten Ausdruck, `Method(objectArray, objectArray, objectArray)` , besteht die einzige Option darin, die erweiterte Form der ersten Methode zu verwenden. Das Programm druckt also `3` .

Übergabe von null als eines der Argumente

Wenn Sie haben

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

und aus irgendeinem Grund müssen Sie die erste Überladung von `F1` aufrufen, aber mit `x = null` , dann einfach

```
F1(null);
```

wird nicht kompiliert, da der Aufruf mehrdeutig ist. Um dem entgegenzuwirken, können Sie tun

```
F1(null as MyType1);
```

Überlastauflösung online lesen: https://riptutorial.com/de/csharp/topic/77/uberlastauflosung

Kapitel 144: Überlauf

Examples

Ganzzahlüberlauf

Es gibt eine maximale Kapazität, die eine ganze Zahl speichern kann. Und wenn Sie dieses Limit überschreiten, wird es zur negativen Seite zurückkehren. Für `int` ist es `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
```

Verwenden Sie für alle Integer-Werte außerhalb dieses Bereichs den Namespace System.Numerics mit dem Datentyp BigInteger. Unter dem Link finden Sie weitere Informationen unter https://msdn.microsoft.com/de-de/library/system.numerics.biginteger(v=vs.110).aspx

Überlauf während des Betriebs

Überlauf tritt auch während des Betriebs auf. Im folgenden Beispiel ist x ein `int` , 1 ist standardmäßig ein `int` . Daher ist der Zusatz ein `int` Zusatz. Und das Ergebnis wird ein `int` . Und es wird überlaufen.

```
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
```

Sie können dies verhindern, indem Sie 1L verwenden. Jetzt wird 1 `long` und die Addition wird `long`

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1L;                    //It will be OK
Console.WriteLine(y);               //Will print 2147483648
```

Bestellung zählt

Im folgenden Code gibt es einen Überlauf

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

Während im folgenden Code kein Überlauf vorliegt

```
int x = int.MaxValue;
```

```
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

Dies liegt an der Reihenfolge der Operationen von links nach rechts. Im ersten Codefragment läuft `x + x` über und wird danach `long` . Auf der anderen Seite wird `x + 1L` `long` und danach wird `x` zu diesem Wert addiert.

Überlauf online lesen: https://riptutorial.com/de/csharp/topic/3303/uberlauf

Kapitel 145: Unsicherer Code in .NET

Bemerkungen

- Um das `unsafe` Schlüsselwort in einem .NET-Projekt verwenden zu können, müssen Sie in Projekteigenschaften => Erstellen "Nicht sicheren Code zulassen" aktivieren
- Die Verwendung von unsicherem Code kann die Leistung verbessern, geht jedoch auf Kosten der Codesicherheit (daher der Begriff `unsafe`).

Wenn Sie beispielsweise eine for-Schleife eines Arrays verwenden,

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

.NET Framework stellt sicher, dass Sie die Grenzen des Arrays nicht überschreiten. Eine `IndexOutOfRangeException` wenn der Index die Grenzen überschreitet.

Wenn Sie jedoch unsicheren Code verwenden, können Sie die Grenzen des Arrays wie folgt überschreiten:

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

Examples

Unsicherer Array-Index

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

Beim Ausführen dieses Codes wird ein Array der Länge 3 erstellt, es wird jedoch versucht, das

fünfte Element (Index 4) abzurufen. Auf meinem Rechner wurde dieser `1910457872` gedruckt, aber das Verhalten ist nicht definiert.

Ohne den `unsafe` Block können Sie keine Zeiger verwenden und daher nicht auf Werte über das Ende eines Arrays hinaus zugreifen, ohne dass eine Ausnahme ausgelöst wird.

Verwendung von Arrays mit unsicher

Wenn auf Arrays mit Zeigern `IndexOutOfRangeException` wird, werden keine Begrenzungen `IndexOutOfRangeException` , und daher wird keine `IndexOutOfRangeException` ausgelöst. Dies macht den Code schneller.

Zuweisen von Werten zu einem Array mit einem Zeiger:

```
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
                }
            }
        }
    }
}
```

Das sichere und normale Gegenstück wäre:

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

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

Der unsichere Teil ist im Allgemeinen schneller und der Leistungsunterschied kann abhängig von der Komplexität der Elemente im Array sowie der auf jedes Element angewandten Logik variieren. Obwohl es schneller sein kann, sollte es mit Vorsicht verwendet werden, da es schwieriger zu warten und leichter zu brechen ist.

Verwendung von Zeichenketten mit unsicher

```
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"
```

Unsicherer Code in .NET online lesen: https://riptutorial.com/de/csharp/topic/81/unsicherer-code-in--net

Kapitel 146: Unveränderlichkeit

Examples

System.String-Klasse

In C # (und .NET) wird eine Zeichenfolge durch die Klasse System.String dargestellt. Das `string` Schlüsselwort ist ein Alias für diese Klasse.

Die System.String-Klasse ist unveränderlich, dh ihr Zustand kann nach ihrer Erstellung nicht mehr geändert werden.

Alle Vorgänge, die Sie an einer Zeichenfolge ausführen, wie z. B. Unterzeichenfolge, Entfernen, Ersetzen, Verkettung mit dem Operator `+` , usw., erstellen eine neue Zeichenfolge und geben diese zurück.

Siehe folgendes Programm zur Demonstration -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

Dadurch werden `string` und `mystring` .

Saiten und Unveränderlichkeit

Unveränderbare Typen sind Typen, die bei einer Änderung eine neue Version des Objekts im Speicher erstellen, anstatt das vorhandene Objekt im Speicher zu ändern. Das einfachste Beispiel hierfür ist der integrierte `string` .

Mit folgendem Code wird "world" an das Wort "Hello" angehängt.

```
string myString = "hello";
myString += " world";
```

Was in diesem Fall im Speicher geschieht, ist, dass ein neues Objekt erstellt wird, wenn Sie an die `string` in der zweiten Zeile anhängen. Wenn Sie dies als Teil einer großen Schleife ausführen, besteht die Möglichkeit, dass dies zu Leistungsproblemen in Ihrer Anwendung führt.

Das veränderbare Äquivalent für einen `string` ist ein `StringBuilder`

Nehmen Sie den folgenden Code

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

Wenn Sie dies ausführen, ändern Sie das `StringBuilder` Objekt selbst im Arbeitsspeicher.

Unveränderlichkeit online lesen: https://riptutorial.com/de/csharp/topic/1863/unveranderlichkeit

Kapitel 147: Veranstaltungen

Einführung

Ein Ereignis ist eine Benachrichtigung, dass ein Ereignis aufgetreten ist (z. B. ein Mausklick) oder in einigen Fällen unmittelbar bevorsteht (z. B. Preisänderung).

Klassen können Ereignisse definieren und ihre Instanzen (Objekte) können diese Ereignisse auslösen. Eine Schaltfläche kann beispielsweise ein Click-Ereignis enthalten, das ausgelöst wird, wenn ein Benutzer darauf geklickt hat.

Event-Handler sind dann Methoden, die aufgerufen werden, wenn das entsprechende Ereignis ausgelöst wird. Ein Formular kann beispielsweise einen Clicked-Ereignishandler für jede Schaltfläche enthalten, die es enthält.

Parameter

Parameter	Einzelheiten
EventArgsT	Der von EventArgs abgeleitete Typ, der die Ereignisparameter enthält.
Veranstaltungsname	Der Name der Veranstaltung
HandlerName	Der Name des Event-Handlers.
SenderObject	Das Objekt, das das Ereignis aufruft.
EventArguments	Eine Instanz des EventArgsT-Typs, die die Ereignisparameter enthält.

Bemerkungen

Wenn Sie eine Veranstaltung auslösen:

- Überprüfen Sie immer, ob der Delegat `null` . Ein null-Delegat bedeutet, dass das Ereignis keine Abonnenten hat. Das Auslösen eines Ereignisses ohne Abonnenten führt zu einer `NullReferenceException` .

6,0

- Kopieren Sie den Delegaten (z. B. `EventName`) in eine lokale Variable (z. B. `eventName`), bevor Sie auf null überprüfen / das Ereignis `eventName` . Dadurch werden Race-Bedingungen in Umgebungen mit mehreren Threads vermieden:

Falsch :

```
    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.
```

Richtig :

```
    // 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

- Verwenden Sie den nullbedingten Operator (?.), `EventName?.Invoke(SenderObject, new EventArgsT());` anstatt den Delegaten in einer `if EventName?.Invoke(SenderObject, new EventArgsT());` null zu überprüfen: `EventName?.Invoke(SenderObject, new EventArgsT());`
- Bei der Verwendung von Aktion <> zum Deklarieren von Delegattypen muss die anonyme Methoden- / Ereignishandler-Signatur mit dem deklarierten anonymen Delegattyp in der Ereignisdeklaration übereinstimmen.

Examples

Ereignisse deklarieren und anheben

Ein Ereignis deklarieren

Sie können ein Ereignis für jede `class` oder `struct` mit der folgenden Syntax deklarieren:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

Es gibt eine erweiterte Syntax für die Deklaration von Ereignissen, bei der Sie eine private Instanz des Ereignisses halten und eine öffentliche Instanz mithilfe von `add` und `set` Accessoren definieren. Die Syntax ist den C # -Eigenschaften sehr ähnlich. In allen Fällen sollte die oben dargestellte Syntax bevorzugt werden, da der Compiler Code ausgibt, um sicherzustellen, dass mehrere Threads Ereignishandler sicher zu dem Ereignis in Ihrer Klasse hinzufügen und entfernen können.

Das Ereignis auslösen

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);

}
```

Beachten Sie, dass Ereignisse nur vom deklarierenden Typ ausgelöst werden können. Kunden können nur abonnieren / abbestellen.

Für C # -Versionen vor 6.0, bei denen `EventName?.Invoke` nicht unterstützt wird, `EventName?.Invoke` es sich, das Ereignis vor dem Aufruf einer temporären Variablen zuzuweisen (`EventName?.Invoke` gewährleistet die Thread-Sicherheit in Fällen, in denen mehrere Threads das gleiche ausführen Code. Andernfalls kann in bestimmten Fällen, in denen mehrere Threads dieselbe Objektinstanz verwenden, eine `NullReferenceException` ausgelöst werden. In C # 6.0 gibt der Compiler Code aus, der dem im Codebeispiel für C # 6 gezeigten ähnlich ist.

Standard-Ereigniserklärung

Ereigniserklärung:

```
public event EventHandler<EventArgsT> EventName;
```

Event-Handler-Deklaration:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

Anmeldung zum Event:

Dynamisch:

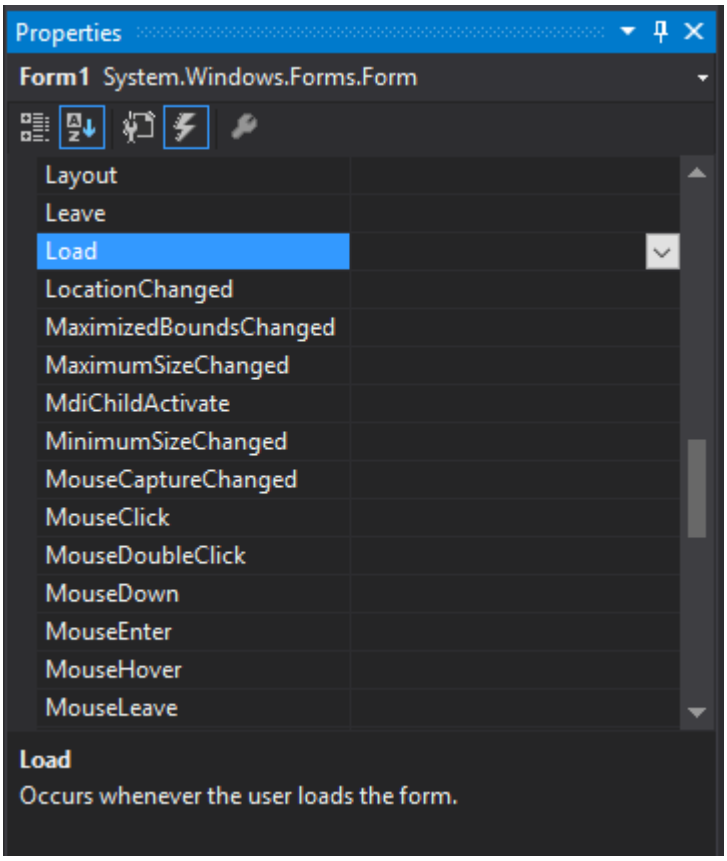
```
EventName += HandlerName;
```

Durch den Designer:

1. Klicken Sie im Eigenschaftenfenster des Steuerelements auf die Schaltfläche Ereignisse

(Blitz).

2. Doppelklicken Sie auf den Ereignisnamen:



3. Visual Studio generiert den Ereigniscode:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Aufrufen der Methode:

```
EventName(SenderObject, EventArguments);
```

Erklärung zum anonymen Ereignishandler

Ereigniserklärung:

```
public event EventHandler<EventArgsType> EventName;
```

Event-Handler-Deklaration mit Lambda-Operator => und Abonnieren des Events:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Event-Handler-Deklaration unter Verwendung der anonymen Methodenmethode für Delegaten :

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Deklaration und Subskription eines Ereignishandlers, der den Parameter des Ereignisses nicht verwendet und daher die obige Syntax verwenden kann, ohne Parameter angeben zu müssen:

```
EventName += delegate { /* Handler Logic */ }
```

Aufrufen der Veranstaltung:

```
EventName?.Invoke(SenderObject, EventArguments);
```

Nicht-Standard-Ereigniserklärung

Ereignisse können einen beliebigen `EventHandler` , nicht nur `EventHandler` und `EventHandler<T>` . Zum Beispiel:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

Dies wird ähnlich wie bei Standard- `EventHandler` Ereignissen verwendet:

```
//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, ...);
```

Es ist möglich, mehrere Ereignisse desselben Typs in einer einzigen Anweisung zu deklarieren, ähnlich wie bei Feldern und lokalen Variablen (dies ist jedoch oft eine schlechte Idee):

```
public event EventHandler Event1, Event2, Event3;
```

Hiermit werden drei separate Ereignisse (`Event1` , `Event2` und `Event3`) vom Typ `EventHandler` . *Anmerkung: Obwohl einige Compiler diese Syntax in Schnittstellen und Klassen akzeptieren, enthält die C # -Spezifikation (v5.0 §13.2.3) Grammatik für Schnittstellen, die dies nicht zulassen. Daher ist die Verwendung dieser Schnittstelle in Schnittstellen bei verschiedenen Compilern unzuverlässig.*

Erstellen von benutzerdefinierten EventArgs mit zusätzlichen Daten

Für benutzerdefinierte Ereignisse sind normalerweise benutzerdefinierte Ereignisargumente erforderlich, die Informationen zum Ereignis enthalten. Beispielsweise enthält `MouseEventArgs` das

von Mausereignissen wie `MouseDown` oder `MouseUp` Ereignissen verwendet wird, Informationen zum `Location` oder zu `Buttons` denen das Ereignis generiert wurde.

Wenn Sie neue Ereignisse erstellen, erstellen Sie ein benutzerdefiniertes Ereignis arg:

- Erstellen Sie eine von `EventArgs` Klasse, und definieren Sie Eigenschaften für die erforderlichen Daten.
- Als Konvention sollte der Name der Klasse mit `EventArgs` .

Beispiel

Im folgenden Beispiel erstellen wir ein `PriceChangingEventArgs` Ereignis für die `Price` Eigenschaft einer Klasse. Die Ereignisdatenklasse enthält einen `CurrentPrice` und einen `NewPrice` . Das Ereignis wird ausgelöst, wenn Sie der Eigenschaft `Price` einen neuen Wert zuweisen und den Verbraucher wissen, dass sich der Wert ändert, und ihn über den aktuellen Preis und den neuen Preis informieren.

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; }
}
```

Produkt

```
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);
    }
```

```
}
```

Sie können das Beispiel verbessern, indem Sie dem Verbraucher erlauben, den neuen Wert zu ändern. Der Wert wird dann für die Eigenschaft verwendet. Dazu genügt es, diese Änderungen in Klassen anzuwenden.

Ändern Sie die Definition von `NewPrice` , dass sie einstellbar ist:

```
public int NewPrice { get; set; }
```

Ändern Sie die Definition von `Price` , um `e.NewPrice` als Eigenschaftswert zu verwenden, nachdem Sie `OnPriceChanging` :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

Stornierbares Ereignis erstellen

Ein stornierbares Ereignis kann von einer Klasse `FormClosing` werden, wenn eine Aktion ausgeführt werden soll, die abgebrochen werden kann, beispielsweise das `FormClosing` Ereignis eines `Form` .

So erstellen Sie ein solches Ereignis:

- Erstellen Sie ein neues Ereignisargument, das von `CancelEventArgs` und fügen Sie zusätzliche Eigenschaften für Ereignisdaten hinzu.
- Erstellen Sie ein Ereignis mit `EventHandler<T>` und verwenden Sie die neue `EventHandler<T>` die Sie erstellt haben.

Beispiel

Im folgenden Beispiel erstellen wir ein `PriceChangingEventArgs` Ereignis für die `Price` Eigenschaft einer Klasse. Die Ereignisdatenklasse enthält einen `Value` der den Verbraucher über das Neue informiert. Das Ereignis wird ausgelöst, wenn Sie der Eigenschaft `Price` einen neuen Wert zuweisen. Der Verbraucher wird darüber informiert, dass sich der Wert ändert, und lässt das Ereignis abbrechen. Wenn der Verbraucher das Ereignis abbricht, wird der vorherige Wert für `Price` verwendet:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
```

```
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Produkt

```
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);
    }
}
```

Ereignis-Eigenschaften

Wenn eine Klasse eine große Anzahl von Ereignissen auslöst, sind die Speicherkosten für ein Feld pro Delegat möglicherweise nicht akzeptabel. .NET Framework stellt Ereigniseigenschaften für diese Fälle bereit. Auf diese Weise können Sie eine andere Datenstruktur wie `EventHandlerList` zum Speichern von Ereignisdelegaten verwenden:

```
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);
    }
}
```

Dieser Ansatz ist in GUI-Frameworks wie WinForms weit verbreitet, wo Steuerelemente Dutzende und sogar Hunderte von Ereignissen enthalten können.

Beachten Sie, dass `EventHandlerList` nicht threadsicher ist. Wenn Sie also erwarten, dass Ihre Klasse von mehreren Threads verwendet wird, müssen Sie `EventHandlerList` oder einen anderen Synchronisationsmechanismus hinzufügen (oder einen Speicher verwenden, der Thread-Sicherheit bietet).

Veranstaltungen online lesen: https://riptutorial.com/de/csharp/topic/64/veranstaltungen

Kapitel 148: Verbatim-Zeichenfolgen

Syntax

- @ "verbatim-Zeichenfolgen sind Zeichenfolgen, deren Inhalt nicht mit Escapezeichen versehen ist. In diesem Fall stellt \ n nicht das Zeilenvorschubzeichen, sondern zwei Einzelzeichen dar: \ und n.
- @ "Um Anführungszeichen zu" ", werden doppelte Anführungszeichen verwendet.

Bemerkungen

Verwenden Sie zum Verketten von String-Literalen das @ -Symbol am Anfang jeder Zeichenfolge.

```
var combinedString = @"\t means a tab" + @" and \n means a newline";
```

Examples

Mehrzeilige Saiten

```
var multiLine = @"This is a

multiline paragraph";
```

Ausgabe:

Das ist ein

mehrzeiliger Absatz

Live-Demo zu .NET-Geige

Mehrzeilige Zeichenfolgen, die doppelte Anführungszeichen enthalten, können ebenso wie in einer einzelnen Zeile mit Escapezeichen versehen werden, da sie wörtliche Zeichenfolgen sind.

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

Live-Demo zu .NET-Geige

Es ist zu beachten, dass die Leerzeichen / Tabulierungen am Anfang der Zeilen 2 und 3 hier tatsächlich im Wert der Variablen vorhanden sind; Überprüfen Sie diese Frage nach möglichen Lösungen.

Doppelte Anführungszeichen vermeiden

Doppelte Anführungszeichen innerhalb wörtlich Strings können durch Verwendung von 2 aufeinanderfolgende doppelte Anführungszeichen entgangen sein `""` repräsentieren ein doppeltes Anführungszeichen `"` im String.

```
var str = @"""I don't think so,"" he said.";
Console.WriteLine(str);
```

Ausgabe:

> "Ich glaube nicht", sagte er.

Live-Demo zu .NET-Geige

Interpolierte verbatim-Zeichenfolgen

Verbatim-Strings können mit den neuen String-Interpolationsfunktionen in C # 6 kombiniert werden.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

Ausgabe:

> Testen \ n 1 2 3
> Neue Zeile

Live-Demo zu .NET-Geige

Wie von einer wörtlichen Zeichenfolge erwartet, werden die Backslashes als Escape-Zeichen ignoriert. Wie von einem interpolierten String erwartet, wird jeder Ausdruck in geschweiften Klammern ausgewertet, bevor er an dieser Position in den String eingefügt wird.

Verbatim-Zeichenfolgen weisen den Compiler an, keine Zeichenumbrüche zu verwenden

In einer normalen Zeichenfolge ist das Backslash-Zeichen das Escape-Zeichen, das den Compiler anweist, die nächsten Zeichen zu suchen, um das tatsächliche Zeichen in der Zeichenfolge zu bestimmen. (Vollständige Liste der Zeichenfluchten)

In verbatim-Zeichenfolgen gibt es keine Zeichen-Escape-Zeichen (mit Ausnahme von `""` das in ein `"`). Um eine verbatim-Zeichenfolge zu verwenden, fügen Sie einfach ein `@` vor den Anfangsanführungszeichen ein.

Diese wörtliche Zeichenfolge

```
var filename = @"c:\temp\newfile.txt"
```

Ausgabe:

c: \ temp \ newfile.txt

Im Gegensatz zur Verwendung einer normalen (nicht wörtlichen) Zeichenfolge:

```
var filename = "c:\temp\newfile.txt"
```

das wird ausgeben:

```
c:    emp
ewfile.txt
```

mit Zeichenflucht. (Das `\t` wird durch ein Tabulatorzeichen ersetzt, und das `\n` wird durch eine neue Zeile ersetzt.)

Live-Demo zu .NET-Geige

Verbatim-Zeichenfolgen online lesen: https://riptutorial.com/de/csharp/topic/16/verbatim-zeichenfolgen

Kapitel 149: Vergleichbar

Examples

Versionen sortieren

Klasse:

```
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;
    }
}
```

Prüfung:

```
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");
```

Ausgabe:

NULL
1
1.0.1
1.1.5
2,0
3.0.10

Demo:

Live-Demo auf Ideone

Vergleichbar online lesen: https://riptutorial.com/de/csharp/topic/4222/vergleichbar

Kapitel 150: Vernetzung

Syntax

- TcpClient (String-Host, int-Port);

Bemerkungen

Sie können den `NetworkStream` von einem `TcpClient` mit `client.GetStream()` und an einen `StreamReader/StreamWriter` , um Zugriff auf die asynchronen Lese- und Schreibmethoden zu erhalten.

Examples

Grundlegender TCP-Kommunikationsclient

Dieses Codebeispiel erstellt einen TCP-Client, sendet "Hello World" über die Socket-Verbindung und schreibt die Serverantwort vor dem Schließen der Verbindung in die Konsole.

```
// 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();
```

Laden Sie eine Datei von einem Webserver herunter

Das Herunterladen einer Datei aus dem Internet ist eine sehr häufige Aufgabe, die von fast jeder Anwendung verlangt wird, die Sie wahrscheinlich erstellen werden.

Dazu können Sie die Klasse " System.Net.WebClient " verwenden.

Die einfachste Verwendung dieses Musters unter Verwendung des Musters "using" wird unten gezeigt:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

In diesem Beispiel wird "using" verwendet, um sicherzustellen, dass Ihr Webclient ordnungsgemäß bereinigt wird, und die benannte Ressource von der URL im ersten Parameter in die benannte Datei auf Ihrer lokalen Festplatte im zweiten übertragen wird Parameter.

Der erste Parameter ist vom Typ " System.Uri ", der zweite Parameter vom Typ " System.String ".

Sie können diese Funktion auch als asynchrones Formular verwenden, so dass sie abläuft und den Download im Hintergrund ausführt, während Ihre Anwendung mit etwas anderem beschäftigt ist. Die Verwendung des Aufrufs auf diese Weise ist in modernen Anwendungen von großer Bedeutung, da dies hilfreich ist um Ihre Benutzeroberfläche ansprechbar zu halten.

Wenn Sie die Async-Methoden verwenden, können Sie Ereignisbehandlungsroutinen anschließen, mit denen Sie den Fortschritt überwachen können, sodass Sie beispielsweise eine Fortschrittsleiste aktualisieren können, etwa wie folgt:

```
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");
```

Ein wichtiger Punkt, den Sie beachten sollten, wenn Sie die Async-Versionen verwenden, und das ist "Seien Sie sehr vorsichtig, wenn Sie sie in einer" using "-Syntax verwenden.

Der Grund dafür ist ziemlich einfach. Sobald Sie die Download-Datei-Methode aufgerufen haben, wird sie sofort zurückgegeben. Wenn Sie dies in einem using-Block haben, kehren Sie zurück, beenden den Block und geben das Klassenobjekt sofort frei und brechen so den laufenden Download ab.

Wenn Sie eine asynchrone Übertragung mit der Methode "using" durchführen, müssen Sie unbedingt innerhalb des umschließenden Blocks bleiben, bis die Übertragung abgeschlossen ist.

Asynchroner TCP-Client

Die Verwendung von `async/await` in C # -Anwendungen vereinfacht das Multi-Threading. So können Sie `async/await` in Verbindung mit einem TcpClient verwenden.

```
// 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();
```

Grundlegender UDP-Client

In diesem Codebeispiel wird ein UDP-Client erstellt, der dann "Hello World" über das Netzwerk an den vorgesehenen Empfänger sendet. Ein Listener muss nicht aktiv sein, da UDP Is ohne Verbindung ist und die Nachricht unabhängig davon gesendet wird. Sobald die Nachricht gesendet wurde, ist die Arbeit des Kunden abgeschlossen.

```
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());
}
```

Nachfolgend finden Sie ein Beispiel für einen UDP-Listener, der den obigen Client ergänzt. Es wird ständig auf einem bestimmten Port auf Datenverkehr warten und diese Daten einfach auf die Konsole schreiben. Dieses Beispiel enthält ein Kontrollkennzeichen ' `done` ', das nicht intern festgelegt ist. Es setzt etwas ein, um dies festzulegen, um den Listener zu beenden und zu

beenden.

```
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
    }
}
```

Vernetzung online lesen: https://riptutorial.com/de/csharp/topic/1352/vernetzung

Kapitel 151: Verwenden von SQLite in C

Examples

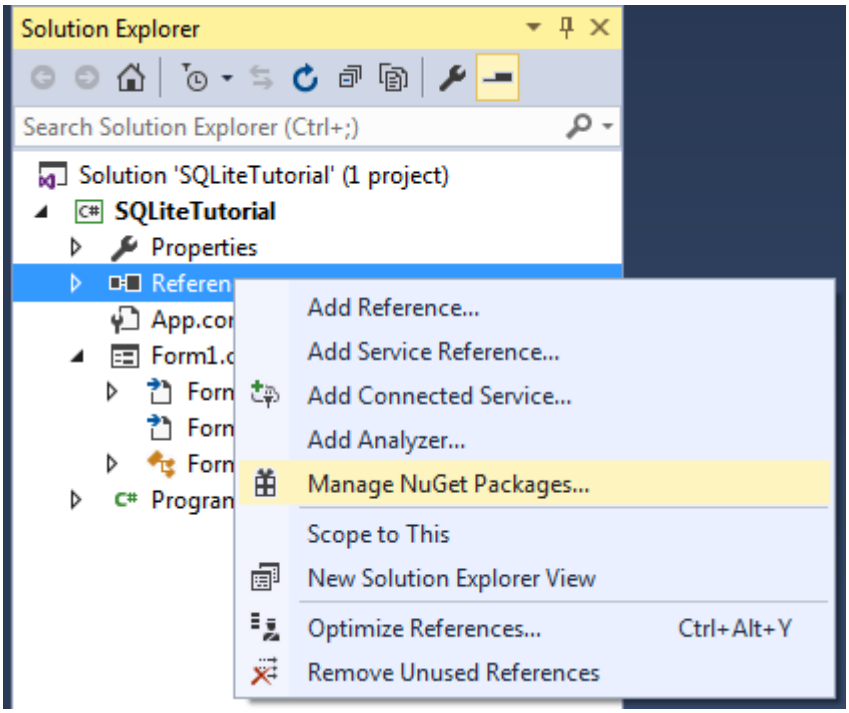
Einfaches CRUD mit SQLite in C # erstellen

Zunächst müssen wir unserer Anwendung die Unterstützung von SQLite hinzufügen. Dafür gibt es zwei Möglichkeiten

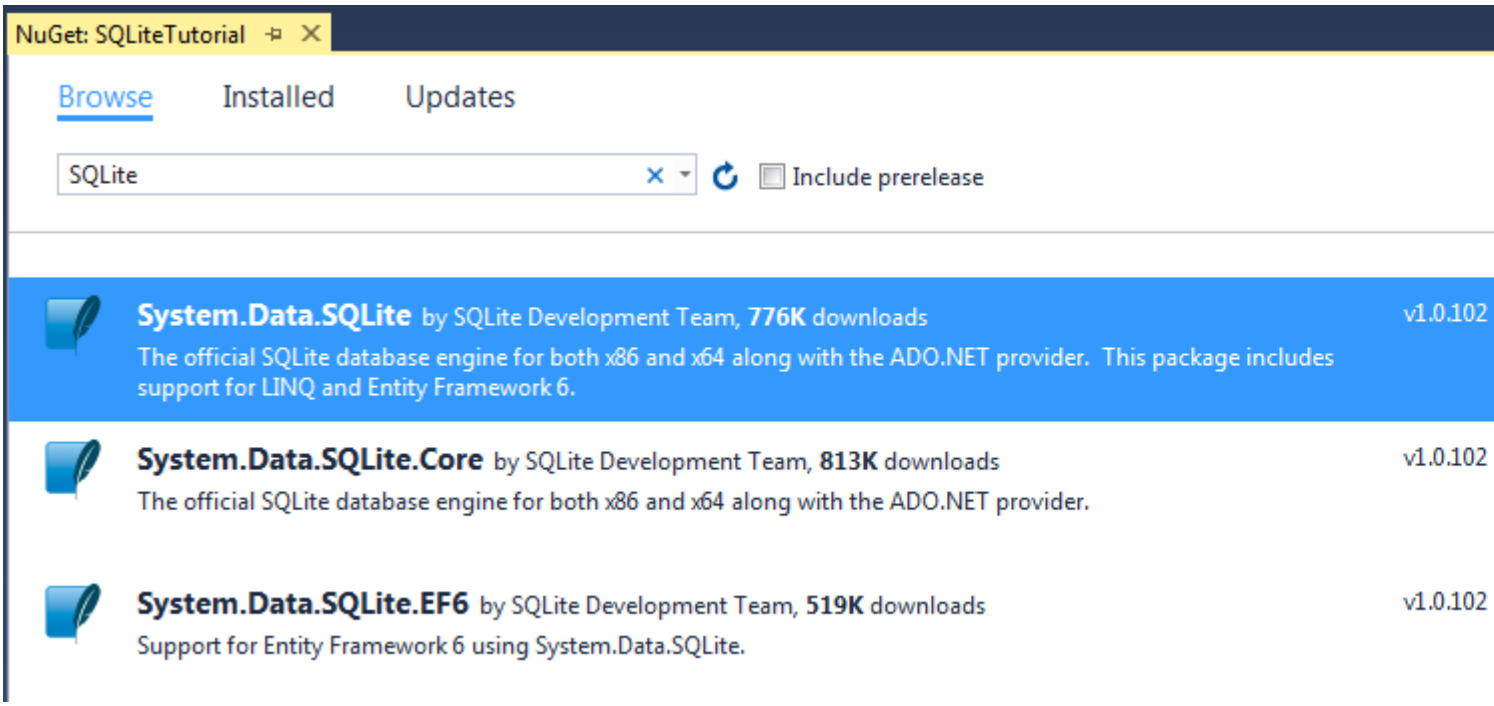
- Laden Sie die DLL für Ihr System von der SQLite-Downloadseite herunter und fügen Sie sie manuell zum Projekt hinzu
- SQLite-Abhängigkeit über NuGet hinzufügen

Wir machen es auf die zweite Weise

Öffnen Sie zuerst das NuGet-Menü



und suchen Sie nach **System.Data.SQLite** , wählen Sie es aus und **klicken** Sie auf **Installieren**



Die Installation kann auch über die Package Manager Console mit durchgeführt werden

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

Oder nur für Kernfunktionen

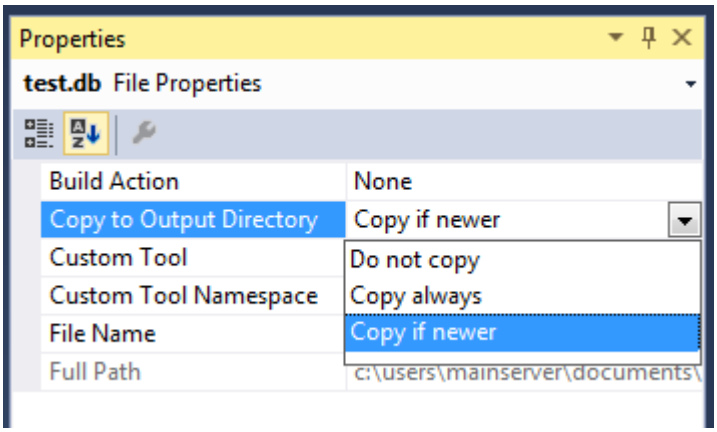
```
PM> Install-Package System.Data.SQLite.Core
```

Das wars für den Download, also können wir gleich mit dem Codieren beginnen.

Erstellen Sie zunächst eine einfache SQLite-Datenbank mit dieser Tabelle und fügen Sie sie dem Projekt als Datei hinzu

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

Vergessen Sie auch nicht, die **Copy to Output Directory-** Eigenschaft der Datei je nach Ihren Anforderungen auf **Copy zu setzen, wenn Sie** von **Copy immer neuer sind**



Erstellen Sie eine Klasse mit dem Namen User, die als Basiseinheit für unsere Datenbank dient

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

Wir schreiben zwei Methoden für die Abfrageausführung, zuerst eine für das Einfügen, Aktualisieren oder Entfernen aus der Datenbank

```
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;
    }
}
```

und der zweite zum Lesen aus der Datenbank

```
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;
        }
    }
}
```

Nun lasst uns in unsere **CRUD-** Methoden einsteigen

Benutzer hinzufügen

```
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);
}
```

Benutzer bearbeiten

```
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);
```

```
}
```

Benutzer löschen

```
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);
}
```

Benutzer nach `Id`

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

Abfrage ausführen

```
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
            }
        }
    }
}
```

Anmerkung : Durch das Festlegen von `FailIfMissing` auf true wird die Datei `data.db` falls diese fehlt. Die Datei ist jedoch leer. Daher müssen alle erforderlichen Tabellen neu erstellt werden.

Verwenden von SQLite in C # online lesen:
https://riptutorial.com/de/csharp/topic/4960/verwenden-von-sqlite-in-c-sharp

Kapitel 152: Verwendung der Richtlinie

Bemerkungen

Das Schlüsselwort `using` ist sowohl eine Direktive (dieses Thema) als auch eine Anweisung.

Für die `using` Anweisung (dh um den Gültigkeitsbereich eines `IDisposable` Objekts zu kapseln und sicherzustellen, dass das Objekt außerhalb dieses Gültigkeitsbereichs sauber entsorgt wird), `IDisposable` Using Statement .

Examples

Grundlegende Verwendung

```
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)");
    }
}
```

Verweisen Sie auf einen Namespace

```
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();
```

Verknüpfen Sie einen Alias mit einem Namespace

```
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();
```

Zugriff auf statische Member einer Klasse

6,0

Ermöglicht das Importieren eines bestimmten Typs und das Verwenden der statischen Member des Typs, ohne sie mit dem Typnamen zu qualifizieren. Dies zeigt ein Beispiel mit statischen Methoden:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

Und dies zeigt ein Beispiel mit statischen Eigenschaften und Methoden:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Verknüpfen Sie einen Alias, um Konflikte zu lösen

Wenn Sie mehrere Namespaces verwenden, die möglicherweise Klassen mit demselben Namen haben (z. B. `System.Random` und `UnityEngine.Random`), können Sie einen Alias verwenden, um anzugeben, dass `Random` aus dem einen oder dem anderen stammt, ohne den gesamten Namespace im Aufruf verwenden zu müssen .

Zum Beispiel:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

Dadurch wird der Compiler unsicher, unter welchem `Random` die neue Variable ausgewertet wird. Stattdessen können Sie Folgendes tun:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

Dies schließt nicht aus, dass Sie den anderen über seinen vollqualifizierten Namespace aufrufen, wie folgt:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` wird eine `System.Random` Variable und `unityRandom` eine `UnityEngine.Random` Variable.

Verwenden von Alias-Direktiven

Sie können use `using` , um einen Alias für einen Namespace oder einen Typ festzulegen. Weitere Details finden Sie hier .

Syntax:

```
using <identifier> = <namespace-or-type-name>;
```

Beispiel:

```
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>());
```

Verwendung der Richtlinie online lesen: https://riptutorial.com/de/csharp/topic/52/verwendung-der-richtlinie

Kapitel 153: Verwendung von C # -Strukturen zum Erstellen eines Union-Typs (ähnlich wie bei C-Unions)

Bemerkungen

Union-Typen werden in mehreren Sprachen verwendet, insbesondere in der C-Sprache, um verschiedene Typen zu enthalten, die sich im selben Speicherbereich "überlappen" können. Mit anderen Worten, sie können verschiedene Felder enthalten, die alle mit dem gleichen Speicheroffset beginnen, auch wenn sie unterschiedliche Längen und Typen haben. Dies hat den Vorteil, dass Sie sowohl Speicher sparen als auch eine automatische Konvertierung durchführen.

Bitte beachten Sie die Kommentare im Konstruktor des Struct. Die Reihenfolge, in der die Felder initialisiert werden, ist äußerst wichtig. Sie möchten zunächst alle anderen Felder initialisieren und dann den Wert, den Sie ändern möchten, als letzte Anweisung festlegen. Da sich die Felder überlappen, zählt der letzte Wert, der zählt.

Examples

C-Style-Vereinigungen in C

Union-Typen werden in mehreren Sprachen verwendet, beispielsweise in der C-Sprache, um verschiedene Typen zu enthalten, die sich "überlappen" können. Mit anderen Worten, sie können verschiedene Felder enthalten, die alle mit dem gleichen Speicheroffset beginnen, auch wenn sie unterschiedliche Längen und Typen haben. Dies hat den Vorteil, dass Sie sowohl Speicher sparen als auch eine automatische Konvertierung durchführen. Stellen Sie sich eine IP-Adresse als Beispiel vor. Intern wird eine IP-Adresse als Ganzzahl dargestellt, aber manchmal möchten wir auf die andere Byte-Komponente zugreifen, wie in Byte1.Byte2.Byte3.Byte4. Dies funktioniert für alle Werttypen, sei es Grundelemente wie Int32 oder Long oder für andere von Ihnen definierte Strukturen.

Wir können den gleichen Effekt in C # erzielen, indem Sie explizite Layoutstrukturen verwenden.

```
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}";
}
```

Nachdem wir Struct auf diese Weise definiert haben, können wir es so verwenden, wie wir eine Union in C verwenden würden. Erstellen Sie beispielsweise eine IP-Adresse als Random Integer und ändern Sie dann das erste Token in der Adresse in '100', indem Sie es ändern von 'ABCD' bis '100.BCD':

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

Ausgabe:

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

Demo anzeigen

Union-Typen in C # können auch Struct-Felder enthalten

Neben Primitiven können die Explicit Layout-Strukturen (Vereinigungen) in C # auch andere Structs enthalten. Solange ein Feld ein Werttyp und keine Referenz ist, kann es in einer Union enthalten sein:

```
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}/";
}
```

Wir können jetzt überprüfen, ob die gesamte Service Union in die Größe einer langen Länge (8 Byte) passt.

```
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}.");
```

Demo anzeigen

Verwendung von C # -Strukturen zum Erstellen eines Union-Typs (ähnlich wie bei C-Unions) online lesen: https://riptutorial.com/de/csharp/topic/5626/verwendung-von-c-sharp--strukturen-zum-erstellen-eines-union-typs--ahnlich-wie-bei-c-unions-

Kapitel 154: Werttyp vs. Referenztyp

Syntax

- Übergeben als Referenz: public void Double (ref int numberToDouble) {}

Bemerkungen

Einführung

Werttypen

Werttypen sind die einfachere der beiden. Werttypen werden häufig verwendet, um Daten selbst darzustellen. Eine Ganzzahl, ein Boolean oder ein Punkt im 3D-Raum sind Beispiele für gute Werttypen.

Werttypen (Strukturen) werden mit dem Schlüsselwort struct deklariert. Ein Beispiel zur Deklaration einer neuen Struktur finden Sie im Abschnitt zur Syntax.

Im Allgemeinen verfügen wir über 2 Schlüsselwörter, mit denen Werttypen deklariert werden:

- Structs
- Aufzählungen

Referenztypen

Referenztypen sind etwas komplexer. Referenztypen sind traditionelle Objekte im Sinne der objektorientierten Programmierung. Sie unterstützen also die Vererbung (und deren Vorteile) und unterstützen Finalisierer.

In C # haben wir im Allgemeinen folgende Referenztypen:

- Klassen
- Delegierte
- Schnittstellen

Neue Referenztypen (Klassen) werden mit dem Schlüsselwort class deklariert. Ein Beispiel finden Sie im Abschnitt zur Syntax, wie Sie einen neuen Referenztyp deklarieren.

Hauptunterschiede

Die wichtigsten Unterschiede zwischen Referenztypen und Werttypen sind unten zu sehen.

Werttypen sind auf dem Stack vorhanden, Referenztypen auf dem Heap

Dies ist der oft erwähnte Unterschied zwischen den beiden, aber eigentlich geht es darum, dass das Programm bei Verwendung eines Werttyps in C # wie einem int diese Variable verwendet, um sich direkt auf diesen Wert zu beziehen. Wenn Sie int mein = 0 sagen, bezieht sich die Variable mein direkt auf 0, was effizient ist. Referenztypen enthalten jedoch tatsächlich (wie der Name schon sagt) einen Verweis auf das zugrunde liegende Objekt. Dies ist vergleichbar mit Zeigern in anderen Sprachen wie C ++.

Sie werden die Auswirkungen möglicherweise nicht sofort bemerken, aber die Auswirkungen sind vorhanden, sind mächtig und subtil. Ein Beispiel finden Sie im Beispiel zum Ändern von Referenztypen an anderer Stelle.

Dieser Unterschied ist der Hauptgrund für die folgenden anderen Unterschiede und ist wissenswert.

Werttypen ändern sich nicht, wenn Sie sie in einer Methode ändern, Referenztypen

Wenn ein Werttyp als Parameter an eine Methode übergeben wird, ändert sich der Wert in keiner Weise durch die Methode. Der Wert wird jedoch nicht geändert. Wenn Sie jedoch einen Referenztyp an dieselbe Methode übergeben und ändern, ändert sich das zugrunde liegende Objekt Für andere Dinge, die dasselbe Objekt verwenden, wird das neu geänderte Objekt anstelle des ursprünglichen Werts angezeigt.

Weitere Informationen finden Sie im Beispiel für Werttypen und Referenztypen in Methoden.

Was ist, wenn ich sie ändern möchte?

Übergeben Sie sie einfach mit dem Schlüsselwort "ref" an Ihre Methode, und Sie übergeben dieses Objekt als Referenz. Das heißt, es ist das gleiche Objekt im Gedächtnis. Die von Ihnen vorgenommenen Änderungen werden daher respektiert. Ein Beispiel finden Sie im Beispiel zur Referenzübergabe.

Werttypen können nicht null sein, Referenztypen können dies tun

So wie es heißt, können Sie einem Referenztyp den Wert null zuweisen. Dies bedeutet, dass der Variablen, die Sie zugewiesen haben, kein Objekt zugewiesen werden kann. Bei Werttypen ist dies jedoch nicht möglich. Sie können jedoch Nullable verwenden, um zuzulassen, dass Ihr Werttyp auf Null gesetzt werden kann. Wenn dies eine Anforderung ist, sollten Sie jedoch bei starkem Nachdenken stark darüber nachdenken, ob eine Klasse hier möglicherweise nicht der beste Ansatz ist, wenn es Ihre eigene ist Art.

Examples

Werte an anderer Stelle ändern

```
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));
    }
}
```

Sie werden feststellen, dass obwohl die Liste der Drucklisten vor den Korrekturen der Schülernamen nach dem Tippfehler erstellt wurde, die Methode PrintPrintingList immer noch die korrigierten Namen ausgibt:

```
Scott Duke
Vincent Kong
Craig Brett
```

Dies liegt daran, dass beide Listen eine Liste von Verweisen auf dieselben Schüler enthalten. Das Ändern des zugrundeliegenden Studentenobjekts führt zu einer Verwendung durch eine der beiden Listen.

So würde die Studentenklasse aussehen.

```
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;
    }
}
```

Übergabe als Referenz

Wenn das Beispiel für Wertetypen im Vergleich zu Referenztypen ordnungsgemäß funktionieren

soll, verwenden Sie das Schlüsselwort ref in Ihrer Methodensignatur für den Parameter, den Sie als Referenz übergeben möchten, sowie beim Aufrufen der Methode.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Wenn Sie diese Änderungen vornehmen, wird die Anzahl wie erwartet aktualisiert. Dies bedeutet, dass die Konsolenausgabe für number 8 wäre.

Übergabe als Referenz mit dem Schlüsselwort ref.

Aus der Dokumentation :

> In C # können Argumente entweder per Wert oder als Referenz an Parameter übergeben werden. Durch das Übergeben als Referenz können Funktionsmitglieder, Methoden, Eigenschaften, Indexer, Operatoren und Konstruktoren den Wert der Parameter ändern und diese Änderung in der aufrufenden Umgebung beibehalten. Um einen Parameter als Referenz zu übergeben, verwenden Sie das Schlüsselwort `ref` oder `out` .

Der Unterschied zwischen `ref` und `out` ist , dass `out` bedeutet , dass der übergebene Parameter zugeordnet werden muss , bevor die Funktion ends.in Kontrastparameter mit geben `ref` geändert werden oder unverändert gelassen.

```
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);
    }
}
```

Ausgabe :

```
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
```

Zuordnung

```
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
```

Durch Zuweisen einer Variablen einer `List<int>` wird keine Kopie der `List<int>` . Stattdessen wird der Verweis in die `List<int>` kopiert. Wir nennen Typen, die sich als *Referenztypen* verhalten.

Unterschied bei den Methodenparametern ref und out

Es gibt zwei Möglichkeiten, einen Werttyp als Referenz zu übergeben: `ref` und `out` . Der Unterschied besteht darin, dass der Wert durch Übergeben mit `ref` initialisiert werden muss, nicht jedoch, wenn er nicht mitgegeben `out` . Durch `out` Verwendung von `out` sichergestellt, dass die Variable nach dem Methodenaufruf einen Wert hat:

```
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
}
```

Der Haken ist, dass der Parameter durch Verwendung von `out` initialisiert werden `must` , bevor die Methode verlassen wird. Daher ist die folgende Methode mit `ref` jedoch nicht ohne `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
```

Dies liegt daran, dass der `value` nicht zugewiesen wird, wenn die `condition` nicht gilt.

ref vs out-Parameter

Code

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the
method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Ausgabe

```
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
```

Werttyp vs. Referenztyp online lesen: https://riptutorial.com/de/csharp/topic/3014/werttyp-vs--

referenztyp

Kapitel 155: Windows Communication Foundation

Einführung

Windows Communication Foundation (WCF) ist ein Framework zum Erstellen von serviceorientierten Anwendungen. Mit WCF können Sie Daten als asynchrone Nachrichten von einem Serviceendpunkt an einen anderen senden. Ein Dienstendpunkt kann Teil eines kontinuierlich verfügbaren Dienstes sein, der von IIS gehostet wird, oder er kann ein Dienst sein, der in einer Anwendung gehostet wird. Die Nachrichten können so einfach wie ein einzelnes Zeichen oder Wort sein, das als XML gesendet wird, oder so komplex wie ein Strom binärer Daten.

Examples

Erste Schritte

Der Dienst beschreibt die Vorgänge, die er in einem Dienstvertrag ausführt, den er öffentlich als Metadaten verfügbar macht.

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

Die Dienstimplementierung berechnet das entsprechende Ergebnis und gibt es zurück, wie im folgenden Beispielcode gezeigt.

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

Der Dienst stellt einen Endpunkt für die Kommunikation mit dem Dienst bereit, der mithilfe einer Konfigurationsdatei (Web.config) definiert wird, wie in der folgenden Beispielkonfiguration gezeigt.

```
<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>
```

Das Framework macht standardmäßig keine Metadaten verfügbar. Daher aktiviert der Dienst das ServiceMetadataBehavior und macht einen MEX-Endpunkt (Metadata Exchange) unter http://localhost/servicemodelsamples/service.svc/mex verfügbar . Die folgende Konfiguration zeigt dies.

```
<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>
```

Der Client kommuniziert mit einem bestimmten Vertragstyp mithilfe einer Clientklasse, die vom ServiceModel Metadata Utility Tool (Svcutil.exe) generiert wird.

Führen Sie an der SDK-Eingabeaufforderung im Clientverzeichnis den folgenden Befehl aus, um den typisierten Proxy zu generieren:

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

Wie der Dienst verwendet der Client eine Konfigurationsdatei (App.config), um den Endpunkt anzugeben, mit dem er kommunizieren möchte. Die Clientendpunktkonfiguration besteht aus einer absoluten Adresse für den Serviceendpunkt, der Bindung und dem Vertrag, wie im folgenden Beispiel gezeigt.

```
<client>
     <endpoint
         address="http://localhost/servicemodelsamples/service.svc"
         binding="wsHttpBinding"
         contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

Die Clientimplementierung instantiiert den Client und verwendet die typisierte Schnittstelle, um mit der Kommunikation mit dem Dienst zu beginnen, wie im folgenden Beispielcode gezeigt.

```
// 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();
```

Windows Communication Foundation online lesen:
https://riptutorial.com/de/csharp/topic/10760/windows-communication-foundation

Kapitel 156: XDocument und der Namespace System.Xml.Linq

Examples

Generieren Sie ein XML-Dokument

Ziel ist es, das folgende XML-Dokument zu generieren:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

XML-Datei ändern

Um eine XML-Datei mit `XDocument` zu ändern, laden Sie die Datei in eine Variable vom Typ `XDocument` , ändern sie im Speicher, speichern sie und speichern die Originaldatei. Ein häufiger Fehler besteht darin, das XML im Arbeitsspeicher zu ändern und eine Änderung der Datei auf der

Festplatte zu erwarten.

Gegeben eine XML-Datei:

```
<?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>
```

Sie möchten die Farbe der Banana in braun ändern:

1. Wir müssen den Pfad zur Datei auf der Festplatte kennen.
2. Eine Überladung von `XDocument.Load` erhält einen URI (Dateipfad).
3. Da die XML-Datei einen Namespace verwendet, müssen Sie den Namespace AND-Elementnamen abfragen.
4. Eine Linq-Abfrage mit C # 6-Syntax, um die Möglichkeit von Nullwerten zu berücksichtigen. Jeder `.` Die in dieser Abfrage verwendete Methode kann eine Nullmenge zurückgeben, wenn die Bedingung keine Elemente findet. Vor C # 6 würden Sie dies in mehreren Schritten tun und dabei nach Null suchen. Das Ergebnis ist das `<Fruit>` -Element, das die Banane enthält. Eigentlich ein `IEnumerable<XElement>` , weshalb der nächste Schritt `FirstOfDefault()` .
5. Jetzt extrahieren wir das FruitColor-Element aus dem soeben gefundenen Fruit-Element. Wir gehen davon aus, dass es nur einen gibt, oder wir kümmern uns nur um den ersten.
6. Wenn es nicht null ist, setzen wir FruitColor auf "Brown".
7. Zum Schluss speichern wir das `XDocument` und überschreiben die Originaldatei auf der Festplatte.

```
// 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);
```

Die Datei sieht jetzt so aus:

```
<?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>
```

Generieren Sie ein XML-Dokument mit fließender Syntax

Tor:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

XDocument und der Namespace System.Xml.Linq online lesen:
https://riptutorial.com/de/csharp/topic/1866/xdocument-und-der-namespace-system-xml-linq

Kapitel 157: XmlDocument und der Namespace System.Xml

Examples

Grundlegende XML-Dokumentinteraktion

```
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);
}
```

Lesen aus einem XML-Dokument

Eine XML-Beispieldatei

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
```

```
    </Sample>
```

Lesen aus dieser XML-Datei:

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

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();

        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

XmlDocument vs XDocument (Beispiel und Vergleich)

Es gibt mehrere Möglichkeiten, mit einer XML-Datei zu interagieren.

1. XML-Dokument
2. XDocument
3. XmlReader / XmlWriter

Vor LINQ to XML wurden wir XMLDocument für Manipulationen in XML wie das Hinzufügen von Attributen, Elementen usw. verwendet. Jetzt verwendet LINQ to XML XDocument für dieselbe Art von Dingen. Syntaxes sind viel einfacher als XMLDocument und erfordern eine minimale Menge an Code.

Auch XDocument ist schneller als XmlDocument. XmlDoucument ist eine alte und schmutzige Lösung zum Abfragen eines XML-Dokuments.

Ich werde einige Beispiele der XmlDocument-Klasse und der XDocument- Klasse zeigen:

XML-Datei laden

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

Erstellen Sie XmlDocument

XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
 XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

Ändern Sie den InnerText des Knotens in XML

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

Speichern Sie die Datei nach der Bearbeitung

Stellen Sie sicher, dass Sie die XML-Datei nach jeder Änderung speichern.

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

Rückgabewerte aus XML

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

Rückgabewert von allen untergeordneten Elementen mit Attribut = etwas

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);
```

Einen Knoten anhängen

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);
```

XmlDocument und der Namespace System.Xml online lesen:
https://riptutorial.com/de/csharp/topic/1528/xmldocument-und-der-namespace-system-xml

Kapitel 158: XML-Dokumentationskommentare

Bemerkungen

Manchmal müssen Sie aus Ihren XML-Kommentaren **eine erweiterte Textdokumentation erstellen** . Leider ***gibt es dafür keinen Standard*** .

Es gibt jedoch einige separate Projekte, die Sie für diesen Fall verwenden können:

- Sandburg
- Doku
- NDoc
- DocFX

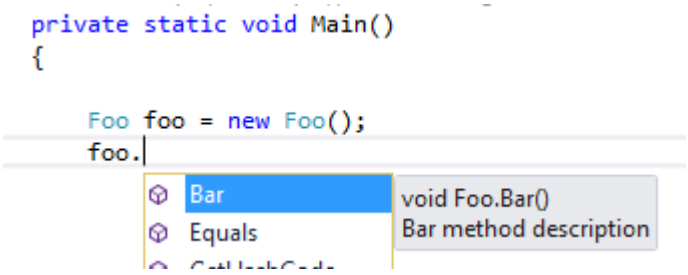
Examples

Einfache Methodenanmerkung

Dokumentationskommentare werden direkt über der von ihnen beschriebenen Methode oder Klasse platziert. Sie beginnen mit drei Schrägstrichen `///` und ermöglichen die Speicherung von Metainformationen über XML.

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

}
```

Informationen in den Tags können von Visual Studio und anderen Tools verwendet werden, um Dienste wie IntelliSense bereitzustellen:



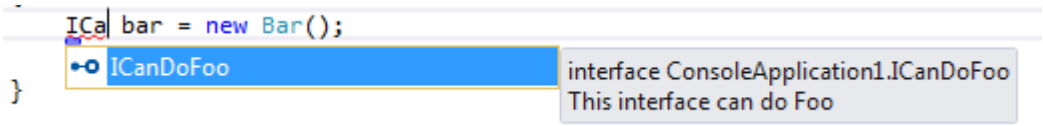
Siehe auch die Liste der üblichen Dokumentations-Tags von Microsoft .

Kommentare zur Schnittstellen- und Klassendokumentation

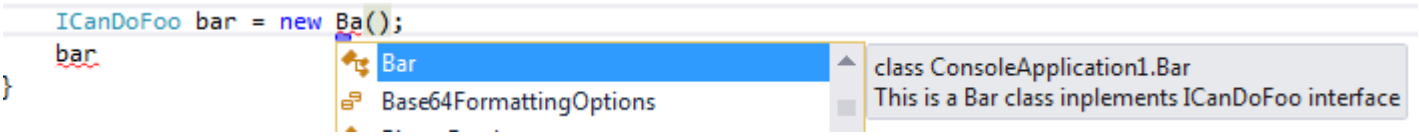
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

Ergebnis

Zusammenfassung der Benutzeroberfläche



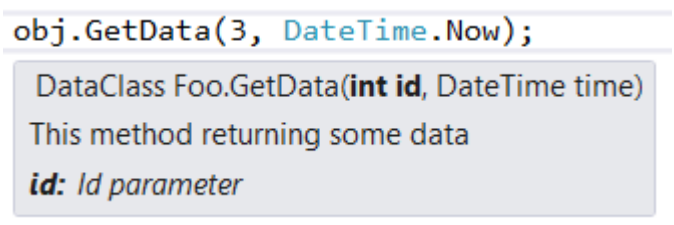
Klassenzusammenfassung



Methodendokumentationskommentar mit Parametern und gibt Elemente zurück

```
/// <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 zeigt Ihnen die Beschreibung für jeden Parameter an:

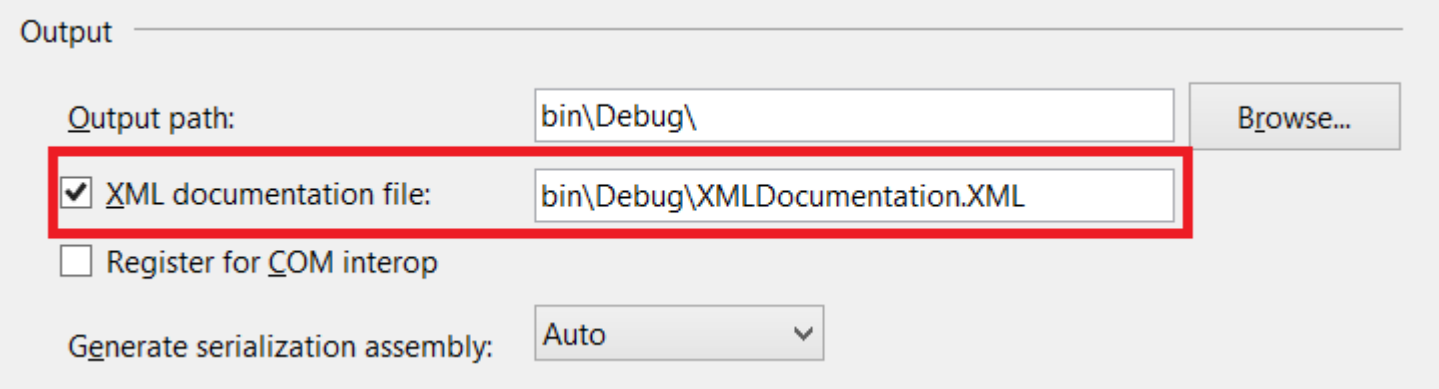


Tipp: Wenn Intellisense in Visual Studio nicht angezeigt wird, löschen Sie die erste Klammer oder das erste Komma und geben Sie es erneut ein.

Generieren von XML aus Dokumentationskommentaren

Verwenden Sie zum Generieren einer XML-Dokumentationsdatei aus Dokumentationskommentaren im Code die Option `/doc` mit dem C # -Compiler `csc.exe` .

Aktivieren Sie in Visual Studio 2013/2015 unter **Projekt** -> **Eigenschaften** -> **Erstellen** -> **Ausgabe das** Kontrollkästchen `XML documentation file` :



Beim Erstellen des Projekts wird vom Compiler eine XML-Datei mit einem Namen erstellt, der dem Projektnamen entspricht (z. B. `XMLDocumentation.dll` -> `XMLDocumentation.xml`).

Wenn Sie die Assembly in einem anderen Projekt verwenden, stellen Sie sicher, dass sich die XML-Datei in demselben Verzeichnis befindet wie die DLL, auf die verwiesen wird.

Dieses Beispiel:

```
/// <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();
    }
}
```

Erzeugt diese XML-Datei beim Build:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>
```

Verweis auf eine andere Klasse in der Dokumentation

Das `<see>` -Tag kann verwendet werden, um eine Verbindung zu einer anderen Klasse herzustellen. Es enthält den `cref` Member, der den Namen der Klasse enthalten soll, auf die verwiesen werden soll. Visual Studio stellt beim Schreiben dieses Tags Intelligense bereit, und solche Referenzen werden auch beim Umbenennen der referenzierten Klasse verarbeitet.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

In Visual Studio Intellisense-Popups werden diese Verweise auch farbig im Text angezeigt.

Verwenden Sie zum Verweisen auf eine generische Klasse Folgendes:

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
/// </summary>
public class SomeGenericClass<T>
{
}
```

XML-Dokumentationskommentare online lesen: https://riptutorial.com/de/csharp/topic/740/xml-dokumentationskommentare

Kapitel 159: Zeiger

Bemerkungen

Hinweise und `unsafe`

Zeiger erzeugen naturgemäß nicht überprüfbaren Code. Daher erfordert die Verwendung eines beliebigen Zeigertyps einen `unsafe` Kontext.

Der Typ `System.IntPtr` ist ein sicherer Wrapper um eine `void*` . Es ist eine bequemere Alternative zu `void*` wenn ansonsten ein unsicherer Kontext nicht erforderlich ist, um die vorliegende Aufgabe auszuführen.

Undefiniertes Verhalten

Wie in C und C ++ kann eine falsche Verwendung von Zeigern undefiniertes Verhalten verursachen, wobei mögliche Nebeneffekte Speicherbeschädigung und die Ausführung von nicht beabsichtigtem Code sein können. Aufgrund der nicht nachweisbaren Natur der meisten Zeigeroperationen liegt die korrekte Verwendung von Zeigern vollständig in der Verantwortung des Programmierers.

Typen, die Zeiger unterstützen

Im Gegensatz zu C und C ++ haben nicht alle C # -Typen entsprechende Zeigertypen. Ein Typ `T` kann einen entsprechenden Zeigertyp haben, wenn beide der folgenden Kriterien zutreffen:

- `T` ist ein Strukturtyp oder ein Zeigertyp.
- `T` enthält nur Member, die beide dieser Kriterien rekursiv erfüllen.

Examples

Zeiger für den Array-Zugriff

Dieses Beispiel zeigt, wie Zeiger für den C-ähnlichen Zugriff auf C # -Arrays verwendet werden können.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
```

```
            *(p + i) = i;
        }
    }
}
```

Das `unsafe` Schlüsselwort ist erforderlich, da der Zeigerzugriff keine Begrenzungsprüfungen auslöst, die normalerweise beim normalen Zugriff auf C # -Anordnungen ausgegeben werden.

Das `fixed` Schlüsselwort weist den C # -Compiler an, Anweisungen zu senden, um das Objekt auf eine ausnahmesichere Weise zu `fixed` . Es ist ein Pinning erforderlich, um sicherzustellen, dass der Garbage Collector das Array nicht in den Arbeitsspeicher versetzt, da dadurch alle Zeiger ungültig werden, die auf das Array zeigen.

Zeigerarithmetik

Addition und Subtraktion in Zeigern funktionieren anders als ganze Zahlen. Wenn ein Zeiger inkrementiert oder dekrementiert wird, wird die Adresse, auf die er zeigt, um die Größe des Referenztyps erhöht oder verringert.

Der Typ `int` (Alias für `System.Int32`) hat beispielsweise eine Größe von 4. Wenn ein `int` in Adresse 0 gespeichert werden kann, kann das nachfolgende `int` in Adresse 4 usw. gespeichert werden. In Code:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

In ähnlicher Weise hat der Typ `long` (Alias für `System.Int64`) eine Größe von 8. Wenn `long` in Adresse 0 gespeichert werden kann, kann der nachfolgende `long` in Adresse 8 usw. gespeichert werden. In Code:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

Der Typ `void` ist speziell und `void` Zeiger sind auch speziell und sie werden als Catch-All-Zeiger verwendet, wenn der Typ nicht bekannt ist oder keine Rolle spielt. `void` Zeiger können aufgrund ihrer größenunabhängigen Natur nicht inkrementiert oder dekrementiert werden:

```
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));
```

Das Sternchen ist Teil des Typs

In C und C ++ ist der Stern in der Deklaration einer Zeigervariable *Teil des* zu deklarierenden *Ausdrucks* . In C # ist der Stern in der Deklaration *Teil des Typs* .

In C, C ++ und C # deklariert der folgende Ausschnitt einen `int` Zeiger:

```
int* a;
```

In C und C ++ deklariert das folgende Snippet einen `int` Zeiger und eine `int` Variable. In C # werden zwei `int` Zeiger deklariert:

```
int* a, b;
```

In C und C ++ deklariert das folgende Snippet zwei `int` Zeiger. In c # ist es ungültig:

```
int *a, *b;
```

Leere*

C # erbt von C und C ++ die Verwendung von `void*` als typunabhängiger und größenagnostischer Zeiger.

```
void* ptr;
```

Jeder Zeigertyp kann mithilfe einer impliziten Konvertierung `void*` zugewiesen werden:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

Das Gegenteil erfordert eine explizite Konvertierung:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Mitgliederzugang mit ->

C # erbt von C und C ++ die Verwendung des Symbols `->` als Mittel zum Zugriff auf die Mitglieder einer Instanz über einen typisierten Zeiger.

Betrachten Sie die folgende Struktur:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

Dies ist ein Beispiel für die Verwendung von `->` um auf seine Mitglieder zuzugreifen:

```
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
```

Generische Zeiger

Die Kriterien, die ein Typ erfüllen muss, um Zeiger zu unterstützen (siehe *Anmerkungen*), können nicht in Form allgemeiner Einschränkungen ausgedrückt werden. Daher schlägt jeder Versuch fehl, einen Zeiger auf einen durch einen generischen Typparameter bereitgestellten Typ zu deklarieren.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Zeiger online lesen: https://riptutorial.com/de/csharp/topic/5524/zeiger

Kapitel 160: Zeiger und unsicherer Code

Examples

Einführung in unsicheren Code

C # erlaubt die Verwendung von Zeigervariablen in einer Funktion des Codeblocks, wenn dieser vom `unsafe` Modifikator markiert wird. Der unsichere Code oder der nicht verwaltete Code ist ein Codeblock, der eine Zeigervariable verwendet.

Ein Zeiger ist eine Variable, deren Wert die Adresse einer anderen Variablen ist, dh die direkte Adresse des Speicherplatzes. Ähnlich wie bei einer Variablen oder Konstante müssen Sie einen Zeiger deklarieren, bevor Sie ihn zum Speichern einer Variablenadresse verwenden können.

Die allgemeine Form einer Zeigerdeklaration lautet:

```
type *var-name;
```

Es folgen gültige Zeigerdeklarationen:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Das folgende Beispiel veranschaulicht die Verwendung von Zeigern in C # mit dem unsicheren Modifikator:

```
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();
      }
   }
}
```

Wenn der obige Code kompiliert und ausgeführt wurde, führt er zu folgendem Ergebnis:

```
Data is: 20
Address is: 99215364
```

Anstatt eine gesamte Methode als unsicher zu deklarieren, können Sie auch einen Teil des Codes als unsicher deklarieren:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

Datenwert mit einem Zeiger abrufen

Sie können die Daten abrufen, die an der durch die Zeigervariable referenzierten Stelle gespeichert wurden, mithilfe der ToString () - Methode. Das folgende Beispiel veranschaulicht dies:

```
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();
      }
   }
}
```

Wenn der obige Code kompiliert und ausgeführt wurde, führt er zu folgendem Ergebnis:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Zeiger als Parameter an Methoden übergeben

Sie können eine Zeigervariable als Parameter an eine Methode übergeben. Das folgende Beispiel veranschaulicht dies:

```
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();
      }
   }
}
```

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Zugriff auf Array-Elemente mit einem Zeiger

In C # sind ein Arrayname und ein Zeiger auf einen Datentyp, der mit den Arraydaten identisch ist, nicht derselbe Variablentyp. Zum Beispiel sind `int *p` und `int[] p` nicht vom gleichen Typ. Sie können die Zeigervariable `p` inkrementieren, da sie nicht im Speicher festgelegt ist, sondern eine Array-Adresse im Speicher und Sie können diese nicht erhöhen.

Wenn Sie also wie in C oder C ++ auf eine Array-Daten mit einer Zeigervariable zugreifen müssen, müssen Sie den Zeiger mit dem festen Schlüsselwort korrigieren.

Das folgende Beispiel veranschaulicht dies:

```
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();
        }
    }
}
```

Wenn der obige Code kompiliert und ausgeführt wurde, führt er zu folgendem Ergebnis:

```
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
```

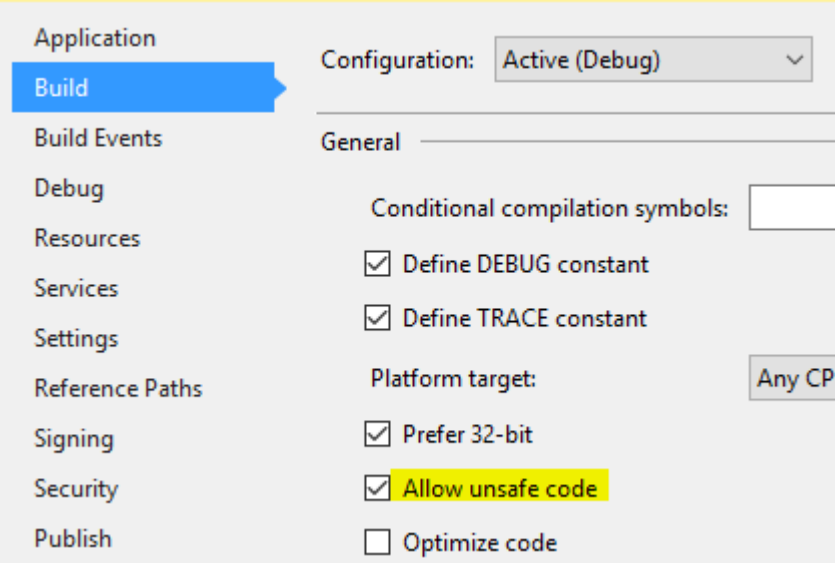
Unsicheren Code kompilieren

Um unsicheren Code zu kompilieren, müssen Sie den Befehlszeilenschalter `/unsafe` mit dem Befehlszeilencompiler angeben.

Um beispielsweise ein Programm mit dem Namen prog1.cs zu kompilieren, das unsicheren Code enthält, geben Sie den folgenden Befehl von der Befehlszeile aus:

```
csc /unsafe prog1.cs
```

Wenn Sie Visual Studio IDE verwenden, müssen Sie in den Projekteigenschaften die Verwendung von unsicherem Code aktivieren.



Um dies zu tun:

- Öffnen Sie die Projekteigenschaften, indem Sie im Projektmappen-Explorer auf den Eigenschaftsknoten doppelklicken.
- Klicken Sie auf die Registerkarte "Erstellen".
- Wählen Sie die Option "Unsicheren Code zulassen".

Zeiger und unsicherer Code online lesen: https://riptutorial.com/de/csharp/topic/5514/zeiger-und-unsicherer-code

Kapitel 161: ZIP-Dateien lesen und schreiben

Syntax

1. public static ZipArchive OpenRead (String ArchivDateiname)

Parameter

Parameter	Einzelheiten
Archivdateiname	Der Pfad zum zu öffnenden Archiv, angegeben als relativer oder absoluter Pfad. Ein relativer Pfad wird als relativ zum aktuellen Arbeitsverzeichnis interpretiert.

Examples

In eine ZIP-Datei schreiben

So schreiben Sie eine neue ZIP-Datei:

```
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-Dateien im Speicher schreiben

Das folgende Beispiel gibt die `byte[]` Daten einer komprimierten Datei mit den bereitgestellten Dateien zurück, ohne dass ein Zugriff auf das Dateisystem erforderlich ist.

```
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();
    }
}
```

Holen Sie sich Dateien aus einer Zip-Datei

In diesem Beispiel wird eine Liste von Dateien aus den bereitgestellten Binärdaten des ZIP-Archivs abgerufen:

```
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();
    }
}
```

Das folgende Beispiel zeigt, wie ein ZIP-Archiv geöffnet und alle TXT-Dateien in einen Ordner extrahiert werden

```
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));
                    }
                }
            }
        }
    }
}
```

ZIP-Dateien lesen und schreiben online lesen: https://riptutorial.com/de/csharp/topic/6709/zip-dateien-lesen-und-schreiben

Kapitel 162: Zufallszahlen in C # generieren

Syntax

- Zufällig()
- Zufällig (int Seed)
- int Next ()
- int Next (int maxValue)
- int Next (int minValue, int maxValue)

Parameter

Parameter	Einzelheiten
Samen	Ein Wert zum Generieren von Zufallszahlen. Wenn nicht festgelegt, wird der Standardwert von der aktuellen Systemzeit bestimmt.
minValue	Generierte Zahlen sind nicht kleiner als dieser Wert. Wenn nicht festgelegt, ist der Standardwert 0.
maxValue	Die generierten Zahlen sind kleiner als dieser Wert. Wenn nicht festgelegt, lautet der Standardwert `Int32.MaxValue` .
Rückgabewert	Gibt eine Zahl mit zufälligem Wert zurück.

Bemerkungen

Der vom System generierte Zufallsstartwert ist bei jedem Durchlauf nicht gleich.

Samen, die zur gleichen Zeit erzeugt werden, können gleich sein.

Examples

Erzeugen Sie ein zufälliges int

In diesem Beispiel werden Zufallswerte zwischen 0 und 2147483647 generiert.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Erzeugen Sie ein zufälliges Doppel

Erzeugen Sie eine Zufallszahl zwischen 0 und 1,0. (nicht einschließlich 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Generiere ein zufälliges int in einem bestimmten Bereich

Erzeugen Sie eine Zufallszahl zwischen `minValue` und `maxValue - 1` .

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Immer wieder die gleiche Folge von Zufallszahlen erzeugen

Beim Erstellen von `Random` mit demselben Startwert werden dieselben Zahlen generiert.

```
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();
}
```

Ausgabe:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

Erstellen Sie mehrere zufällige Klassen mit unterschiedlichen Startwerten gleichzeitig

Zwei zufällige Klassen, die gleichzeitig erstellt werden, haben den gleichen Startwert.

Das Verwenden von `System.Guid.NewGuid().GetHashCode()` kann sogar gleichzeitig einen anderen `System.Guid.NewGuid().GetHashCode()` erhalten.

```
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());
```

Eine andere Möglichkeit, verschiedene Startwerte zu erhalten, besteht darin, eine andere `Random` zum Abrufen der Startwerte zu verwenden.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

Dies macht es auch möglich , das Ergebnis aller zu steuern `Random` , indem nur der Startwert für die Einstellung `rndSeeds` . Alle anderen Fälle werden deterministisch von diesem einzelnen Startwert abgeleitet.

Erzeugen Sie ein zufälliges Zeichen

Generieren Sie einen zufälligen Buchstaben zwischen `a` und `z` indem Sie `Next()` für einen bestimmten Zahlenbereich verwenden und dann das resultierende `int` in ein `char` konvertieren

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

Generieren Sie eine Zahl, die einen Prozentsatz eines Maximalwerts darstellt

Eine häufige Notwendigkeit für Zufallszahlen ist es, eine Zahl zu generieren, die `X%` eines maximalen Werts ist. Dies kann durch Behandeln des Ergebnisses von `NextDouble()` in Prozent erfolgen:

```
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.
```

Zufallszahlen in C # generieren online lesen:
https://riptutorial.com/de/csharp/topic/1975/zufallszahlen-in-c-sharp-generieren

Kapitel 163: Zugriff auf Datenbanken

Examples

ADO.NET-Verbindungen

ADO.NET-Verbindungen sind eine der einfachsten Methoden, um von einer C # -Anwendung aus eine Verbindung zu einer Datenbank herzustellen. Sie sind auf die Verwendung eines Providers und einer Verbindungszeichenfolge angewiesen, die auf Ihre Datenbank verweist, um Abfragen auszuführen.

Allgemeine Datenanbieter-Klassen

Viele der folgenden Klassen werden häufig zum Abfragen von Datenbanken und den zugehörigen Namespaces verwendet:

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

Alle diese Optionen werden im Allgemeinen für den Zugriff auf Daten über C # verwendet. Sie werden häufig in gebäudedatenzentrierten Anwendungen angetroffen. Von vielen anderen Klassen, die nicht erwähnt werden und die dieselben Klassen `FooConnection` , `FooCommand` und `FooDataReader` `FooCommand` , kann erwartet werden, dass sie sich genauso verhalten.

Allgemeines Zugriffsmuster für ADO.NET-Verbindungen

Ein allgemeines Muster, das beim Zugriff auf Ihre Daten über eine ADO.NET-Verbindung verwendet werden kann, könnte wie folgt aussehen:

```
// 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
         }
    }
```

```
}
```

Oder wenn Sie nur ein einfaches Update durchführen und keinen Leser benötigen, würde dasselbe Grundkonzept gelten:

```
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();
     }
}
```

Sie können sogar für eine Reihe allgemeiner Schnittstellen programmieren und müssen sich nicht um die anbieterspezifischen Klassen kümmern. Die von ADO.NET bereitgestellten Kernschnittstellen sind:

- IDbConnection - zum Verwalten von Datenbankverbindungen
- IDbCommand - zum Ausführen von SQL-Befehlen
- IDbTransaction - zum Verwalten von Transaktionen
- IDataReader - zum Lesen von Daten, die von einem Befehl zurückgegeben werden
- IDataAdapter - zum Weiterleiten von Daten zu und von Datensätzen

```
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()) {
                ...
            }
        }
    }
}
```

Entity Framework-Verbindungen

Entity Framework macht Abstraktionsklassen `DbContext` , die zur Interaktion mit zugrunde liegenden Datenbanken in Form von Klassen wie `DbContext` . Diese Kontexte bestehen im Allgemeinen aus `DbSet<T>` -Eigenschaften, die die verfügbaren Sammlungen `DbSet<T>` , die abgefragt werden können:

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

Der `DbContext` selbst übernimmt das Herstellen der Verbindungen mit den Datenbanken und liest im Allgemeinen die entsprechenden Verbindungszeichenfolgendaten aus einer Konfiguration, um zu bestimmen, wie die Verbindungen hergestellt werden:

```
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; }
}
```

Ausführen von Entity Framework-Abfragen

Das Ausführen einer Entity Framework-Abfrage kann sehr einfach sein. Sie müssen lediglich eine Instanz des Kontexts erstellen und dann die verfügbaren Eigenschaften für den Abruf oder Zugriff auf Ihre Daten verwenden

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

Entity Framework bietet auch ein umfangreiches System zur Änderungsnachverfolgung, mit dem Sie die Aktualisierung von Einträgen in Ihrer Datenbank abwickeln können, indem Sie einfach die `SaveChanges()` -Methode aufrufen, um Änderungen in die Datenbank zu verschieben:

```
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();
      }
```

```
}
```

Verbindungszeichenfolgen

Eine Verbindungszeichenfolge ist eine Zeichenfolge, die Informationen zu einer bestimmten Datenquelle und zum Herstellen einer Verbindung zu ihr angibt, indem Anmeldeinformationen, Speicherorte und andere Informationen gespeichert werden.

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

Speichern der Verbindungszeichenfolge

In der Regel wird eine Verbindungszeichenfolge in einer Konfigurationsdatei (z. B. `app.config` oder `web.config` in ASP.NET-Anwendungen) gespeichert. Das folgende Beispiel zeigt, wie eine lokale Verbindung in einer dieser Dateien aussehen könnte:

```
<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>
```

Dadurch kann Ihre Anwendung programmgesteuert über `WidgetsContext` auf die Verbindungszeichenfolge `WidgetsContext` . Obwohl sowohl `Integrated Security=SSPI` als auch `Integrated Security=True` die gleiche Funktion erfüllen, `Integrated Security=SSPI` wird bevorzugt, da es sowohl mit dem SQLClient- als auch dem OleDB-Provider funktioniert, wobei `Integrated Security=true` eine Ausnahme auslöst, wenn er mit dem OleDb-Provider verwendet wird.

Verschiedene Anschlüsse für verschiedene Anbieter

Jeder Datenanbieter (SQL Server, MySQL, Azure usw.) verfügt über eine eigene Syntax für die Verbindungszeichenfolgen und stellt verschiedene verfügbare Eigenschaften zur Verfügung. ConnectionStrings.com ist eine unglaublich nützliche Ressource, wenn Sie sich nicht sicher sind, wie Ihre aussehen soll.

Zugriff auf Datenbanken online lesen: https://riptutorial.com/de/csharp/topic/4811/zugriff-auf-datenbanken

Kapitel 164: Zugriff auf den freigegebenen Netzwerkordner mit Benutzername und Kennwort

Einführung

Zugriff auf die Netzwerkfreigabedatei mit PInvoke.

Examples

Code für den Zugriff auf die freigegebene Netzwerkdatei

```
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
}
```

Zugriff auf den freigegebenen Netzwerkordner mit Benutzername und Kennwort online lesen: https://riptutorial.com/de/csharp/topic/9627/zugriff-auf-den-freigegebenen-netzwerkordner-mit-benutzername-und-kennwort

Kapitel 165: Zugriffsmodifizierer

Bemerkungen

Wenn der Zugriffsmodifizierer weggelassen wird,

- Klassen sind standardmäßig `internal`
- Methoden sind von doult `private`
- Getter und Setter erben den Modifizierer der Eigenschaft, standardmäßig ist dies `private`

Zugriffsmodifizierer für Setter oder Eigenschaften von Eigenschaften können den Zugriff nur einschränken, nicht erweitern: `public string someProperty {get; private set;}`

Examples

Öffentlichkeit

Das Schlüsselwort `public` stellt jedem Verbraucher eine Klasse (einschließlich verschachtelter Klassen), Eigenschaften, Methoden oder Felder zur Verfügung:

```
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;
    }
}
```

Privatgelände

Das `private` Schlüsselwort kennzeichnet Eigenschaften, Methoden, Felder und verschachtelte Klassen, die nur innerhalb der Klasse verwendet werden können:

```
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();
    }
}
```

intern

Das interne Schlüsselwort macht eine Klasse (einschließlich verschachtelter Klassen), eine Eigenschaft, eine Methode oder ein Feld für jeden Benutzer in derselben Assembly verfügbar:

```
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; }
    }
}
```

Dies kann unterbrochen werden, damit eine Test-Assembly durch Hinzufügen von Code zur AssemblyInfo.cs-Datei auf den Code zugreifen kann:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

geschützt

Das Feld für das `protected` Schlüsselwort kennzeichnet die Methodeneigenschaften und die

verschachtelten Klassen, die nur innerhalb derselben Klasse und derselben abgeleiteten Klassen verwendet werden:

```
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
    }
}
```

intern geschützt

Das `protected internal` Schlüsselwort kennzeichnet Feld, Methoden, Eigenschaften und verschachtelte Klassen zur Verwendung in derselben Assembly oder abgeleiteten Klassen in einer anderen Assembly:

Montage 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();
    }
}
```

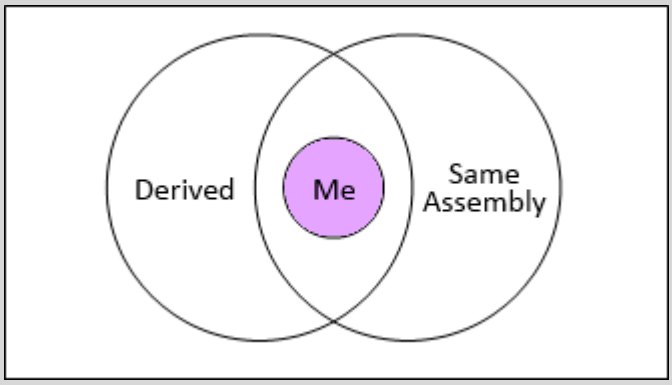
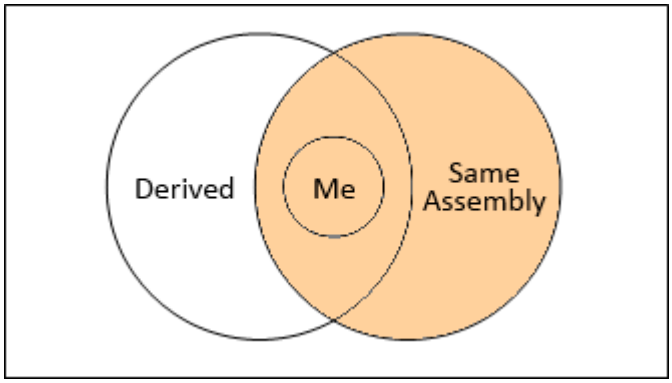
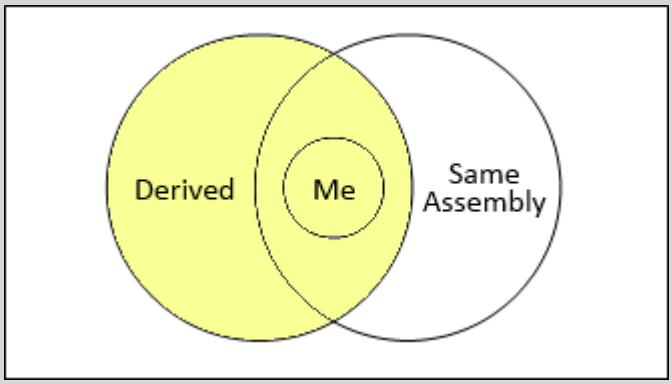
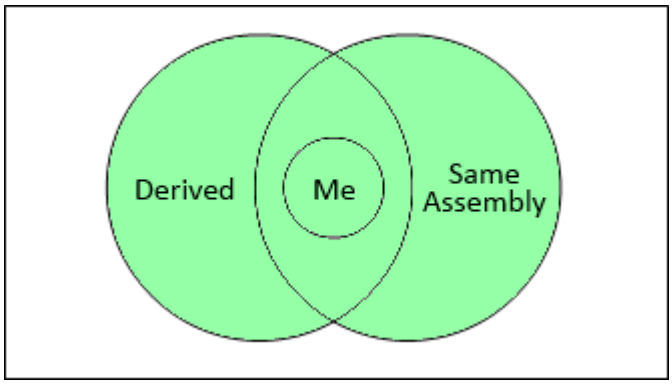
Montage 2

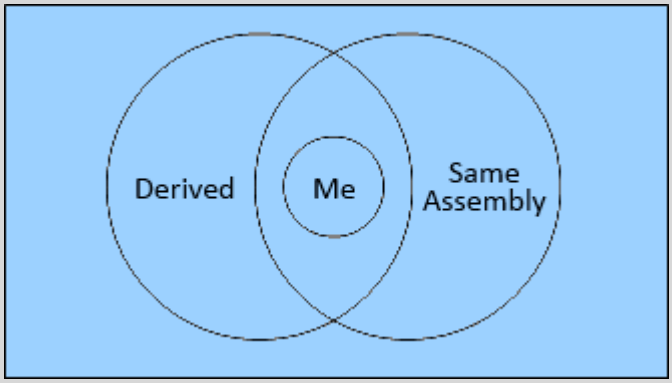
```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    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();
    }
```

```
}
```

Zugriffsmodifizierer-Diagramme

Hier sind alle Zugriffsmodifizierer in venn-Diagrammen, von einschränkend bis zugänglicher:

Zugriffsmodifizierer	Diagramm
Privatgelände	
intern	
geschützt	
intern geschützt	

Zugriffsmodifizierer	Diagramm
Öffentlichkeit	

Nachfolgend finden Sie weitere Informationen.

Zugriffsmodifizierer online lesen: https://riptutorial.com/de/csharp/topic/960/zugriffsmodifizierer

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit C # Language	4444, A. Raza, A_Arnold , aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Confiqure, crunchy, David G., David Pine, DavidG, DAXaholic , Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch , Frankenstine Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis , Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k , Kevin Montrose, Kritner , Krzyserious, leumas1960, M Monis

		Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput , Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki , Zze
2	.NET Compiler-Plattform (Roslyn)	4444, Lukáš Lánský
3	Abhängigkeitsspritze	Buh Buh, iaminvinicble, Kyle Trauberman, Wiktor Dębski
4	Aktionsfilter	Lokesh_Ram
5	Aliase von eingebauten Typen	Racil Hilan, Rahul Nikate , Stephen Leppik
6	Allgemeine Zeichenkettenoperationen	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

7	Anonyme Typen	Fernando Matsumoto, goric, Stephen Leppik
8	Anweisung verwenden	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
9	Arrays	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
10	ASP.NET-Identität	HappyCoding, Skullomania
11	AssemblyInfo.cs Beispiele	Adi Lester, Ameya Deshpande, AndreyAkinshin, Boggin, Dodzi Dzakuma, dove, Joel Martinez, pinkfloydx33, Ralf Bönning, Theodoros Chatzigiannakis, Wasabi Fan
12	Async / await, Backgroundworker, Task- und Thread-Beispiele	Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi
13	Async-Await	Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean, Craig Brett, Donald Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft, Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev, Stefano d'Antonio, Stephen Leppik,

		SynerCoder, trashr0x, Tseng, user2321864, Vincent
14	Asynchroner Sockel	Timon Post
15	Attribute	Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili
16	Ausdrucksbäume	Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer, Matas Vaitkevicius, matteeyah, meJustAndrew, Nathan Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Рахул Маквана
17	Ausnahmebehandlung	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
18	BackgroundWorker	Bovaz, Draken, ephtee, Jacobr365, Will
19	Bedingte Anweisungen	Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz
20	Behandlung von FormatException beim Konvertieren von Zeichenfolgen in andere Typen	Rakitić, un-lucky
21	Benannte Argumente	Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen Leppik
22	Benannte und optionale Argumente	RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov
23	BigInteger	4444, Ed Marty, James Hughes, Rob, The_Outsider
24	Binäre Serialisierung	David, Maxim, RamenChef, Stephen Leppik
25	Bindungsliste	Bovaz, Stephen Leppik, yumaikas
26	C # 3.0-Funktionen	0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob
27	C # 4.0-Funktionen	Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima,

		Sibeesh Venu, Squidward, Theodoros Chatzigiannakis
28	C # 5.0-Funktionen	Abob, alex.b, H. Pauwelyn
29	C # 6.0-Funktionen	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
30	C # 7.0-Funktionen	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
31	C # Authentifizierungshandler	Abbas Galiyakotwala
32	C # Skript	mehrandvd, Squidward, Stephen Leppik
33	Caching	Aliaksei Futryn, th1rdey3
34	Casting	Benjamin Hodgson, MSE , RamenChef, StriplingWarrior
35	CLSCompliantAttribute	mybirthname, Rob
36	Code-Verträge	MegaTron
37	Collection-Initialisierer	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
38	Datei- und Stream-E / A	BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgun DZ, intox, Mikko Viitala, Sender, Squidward, Tolga Evcimen, Wasabi Fan
39	Datenanmerkung	Maxime, Mikko Viitala, The_Outsider, Will Ray
40	Datenflusskonstruktionen für Task Parallel Library (TPL)	Droritos, Stephen Leppik
41	DateTime-Methoden	AbdulRahman Ansari, C4u, Christian Gollhardt, Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz , Oluwafemi, Pavel Yermalovich, Sondre, theinarasu, Thulani Chivandikwa
42	Delegierte	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
43	Diagnose	Jasmin Solanki, Luke

		Ryan, TylerH
44	Dynamischer Typ	Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob
45	Eigene MessageBox in Windows Form Application erstellen	Mansel Davies, Vaibhav_Welcomes_You
46	Eigenschaften	Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled
47	Eigenschaften initialisieren	Blorgbeard, hatchet, jaycer, Michael Sorens, Parth Patel, Stephen Leppik
48	Eine Übersicht über c # -Kollektionen	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
49	Einen variablen Thread sicher machen	Wyck
50	Einfädeln	Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepdelaw, Mamta D, Matthijs Wessels, Mellow , RamenChef, Zoba
51	Eingebaute Typen	Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee

52	Einschließlich Font-Ressourcen	Bales, Facebamm
53	Enum	Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked, Kyle Trauberman, Martin Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, Tanner Swett, TarkaDaal , Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai
54	Equals und GetHashCode	Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew, Rob, scher, Timitry, viggity
55	Erbe	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

56	Ergebnis Keyword	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
57	Erste Schritte: Json mit C #	Neo Vijay, Rob, VitorCioletti
58	Erweiterungsmethoden	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
59	FileSystemWatcher	Sondre
60	Flyweight Design Pattern implementieren	Jan Bońkowski
61	Func Delegierte	Theodoros Chatzigiannakis, Valentin
62	Funktion mit mehreren Rückgabewerten	Adam, Alexey Mitev, Durgpal Singh, Tolga Evcimen
63	Funktionale Programmierung	Andrei Epure, Boggin, Botond Balázs, richard
64	Generics	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
65	Generischer Lambda Query Builder	4444, PedroSouki
66	Geprüft und nicht geprüft	Botond Balázs, Rahul Nikate, Sam Johnson, ZenLulz
67	Gleichheitsoperator	Vadim Martynov
68	Google-Kontakte importieren	4444, Supraj v
69	Guid	Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void
70	Hash-Funktionen	Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz
71	HTTP-Anfragen ausführen	Gordon Bell, Jon Schneider, Mark Shevchenko
72	ICloneable	ja72, Rob
73	IDisposable-Schnittstelle	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

74	IEnumerable	4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider
75	ILGenerator	Aleks Andreev, thehennyy
76	Implementierung des Decorator Design Pattern	Jan Bońkowski
77	Indexer	A_Arnold, Ehsan Sajjad, jHilscher
78	INotifyPropertyChanged-Schnittstelle	mbrdev, Stephen Leppik, Vlad
79	Interoperabilität	Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward
80	IQueryable-Schnittstelle	lucavgobbi, Michiel van Oosterhout, RamenChef, Rob
81	Iteratoren	Botond Balázs, Lijo, Nate Barbettini, Tagc
82	Json.net verwenden	Aleks Andreev, Snipzwolf
83	Kodex-Verträge und Zusicherungen	Roy Dictus
84	Kommentare und Regionen	Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward
85	Konsolenanwendung mit einem Nur-Text-Editor und dem C # -Compiler erstellen (csc.exe)	delete me
86	Konstruktoren und Finalisierer	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
87	Kreationelle Designmuster	DWright, Jan Bońkowski, Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel
88	Kryptographie (System.Security.Cryptography)	glaubergft, MikeS159, Ogglas, Pete
89	Lambda-Ausdrücke	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
90	Laufzeit kompilieren	Artificial Stupidity, Stephen Leppik, Tommy
91	Linq zu Objekten	brijber, Christian Gollhardt, FortyTwo, Kevin Green, Raphael Pantaleão, Simon Halsey, Tanveer Badar
92	LINQ zu XML	Denis Elkhov, Stephen Leppik, Uali
93	LINQ-Abfragen	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
94	Literale	jaycer, NotEnoughData, Racil Hilan
95	Lock-Anweisung	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
96	Looping	Alisson, Andrei Rînea, B Hawkins, Benjamin Hodgson, Botond Balázs, connor, Dialecticus, DJCubed, Freelex, Jon Schneider, Oluwafemi, Racil Hilan, Squidward, Testing123, Tolga Evcimen
97	Methoden	Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky

98	Microsoft.Exchange.WebServices	Bassie
99	Müllsammler in .Net	Andrei Rînea, da_sann, Eamon Charles, J3soon, Luke Ryan, Squidward, Suren Srapyan
100	Name des Betreibers	Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan
101	Nicht zulässige Typen	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
102	Nullbedingte Operatoren	Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan
103	Nullkoaleszenzoperator	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
104	NullReferenceException	4444, Agramer, Ashutosh, krimog, Kyle Trauberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato
105	O (n) Algorithmus für die Kreisrotation eines Arrays	AFT
106	Objektinitialisierer	Andrei, Kroltan, LeopardSkinPillBoxHat, Marco, Nick DeVore, Stephen Leppik
107	Objektorientierte Programmierung in C #	Yashar Aliabasi
108	ObservableCollection	demonplus, GeralexGR, Jonathan Anctil, MuiBienCarlota
109	Operatoren	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
110	Parallele LINQ (PLINQ)	Adi Lester
111	Polymorphismus	Ade Stringer, ganchito55 , H. Pauwelyn, Karthik, Maximilian Ast, void
112	Präprozessor-Anweisungen	Andrei, Gilad Naaman, Matas Vaitkevicius, qJake, RamenChef, theB, volvis
113	Reaktive Erweiterungen (Rx)	stefankmitph
114	Reflexion	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
115	Regeln der Namensgebung	Ben Aaronson, Callum Watkins, PMF, ZenLulz
116	Regex-Analyse	C4u
117	Rekursion	Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes, Yves Schelpe
118	Schlüsselwörter	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
119	Schnittstellen	Avia, Botond Balázs, CyberFox, harriyott,

		hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511
120	Singleton-Implementierung	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
121	Stacktraces lesen und verstehen	S.L. Barth
122	Statische Klassen	MCronin, The_Outsider, Xiaoy312
123	Stoppuhren	Adam, demonplus, dotctor, Gavin Greenwalt , Jeppe Stig Nielsen, Sondre
124	String Escape-Sequenzen	Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush
125	String Interpolation	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
126	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
127	StringBuilder	ATechieThought, brijber, Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider
128	String-Manipulation	Blachshma, Jon Schneider, sferencik, The_Outsider
129	String-Verkettung	Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi, Rob, RubberDuck, Testing123, The_Outsider
130	Strom	Danny Bogers, jlawcordova, Jon Schneider, Nuri Tasdemir, Pushpendra
131	Structs	abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad

		Lachmi, ganchito55, Jon Schneider, NikolayKondratyev
132	Strukturelle Entwurfsmuster	Timon Post
133	Synchronisierungskontext in Async-Await	codeape, Mark Shevchenko, RamenChef
134	System.DirectoryServices.Protocols.LdapConnection	Andrew Stollak
135	System.Management.Automation	Mikko Viitala
136	T4-Codegenerierung	lloyd, Pavel Mayorov
137	Task Parallele Bibliothek	Benjamin Hodgson, Brandon, Collin Stevens, i3arnon, Mokhtar Ashour , Murtuza Vohra
138	Teilklasse und Methoden	Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider
139	Timer	Adam, Akshay Anand, Benjamin Kozuch, ephtee, RamenChef, Thennarasan
140	Tuples	Bovaz, Chawin, EFrank, H. Pauwelyn, Mark Benovsky, Muhammad Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011
141	Typumwandlung	Community, connor, Ehsan Sajjad, Lijo
142	Überlastauflösung	Dunno, Petr Hudeček, Stephen Leppik, TorbenJ
143	Überlauf	Akshay Anand, Nuri Tasdemir, tonirush

144	Unsicherer Code in .NET	Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz
145	Unveränderlichkeit	Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer
146	Veranstaltungen	Aaron Hudon, Adi Lester , Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng , NotEnoughData, OliPro007, paulius_l, PSGuy, Reza Aghaei, Roy Dictus, Squidward, Steven, vbnet3d
147	Verbatim-Zeichenfolgen	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
148	Vergleichbar	alex
149	Vernetzung	Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano
150	Verwenden von SQLite in C #	Carmine,

		NikolayKondratyev, th1rdey3, Tim Yusupov
151	Verwendung der Richtlinie	Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit, Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam
152	Verwendung von C # -Strukturen zum Erstellen eines Union-Typs (ähnlich wie bei C-Unions)	DLeh, Milton Hernandez, Squidward, usr
153	Werttyp vs. Referenztyp	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
154	Windows Communication Foundation	NtFreX
155	XDocument und der Namespace System.Xml.Linq	Crowcoder, Jon Schneider
156	XmlDocument und der Namespace System.Xml	Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr
157	XML-Dokumentationskommentare	Alexander Mandt, James , jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn
158	Zeiger	Jeppe Stig Nielsen, Theodoros Chatzigiannakis
159	Zeiger und unsicherer Code	Aaron Hudon, Botond Balázs, undefined

160	ZIP-Dateien lesen und schreiben	4444, DLeh, Naveen Gogineni, Nisarg Shah
161	Zufallszahlen in C # generieren	A. Can Aydemir, Adi Lester, Alexander Mandt , DLeh, J3soon, Rob
162	Zugriff auf Datenbanken	ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864
163	Zugriff auf den freigegebenen Netzwerkordner mit Benutzername und Kennwort	Mohsin khan
164	Zugriffsmodifizierer	Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz