**Kostenloses eBook**

# LERNEN .NET Framework

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#.net

# Inhaltsverzeichnis

# Über

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

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



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

# Kapitel 1: Erste Schritte mit .NET Framework

## Bemerkungen

Das .NET Framework besteht aus einer Reihe von Bibliotheken und einer Laufzeitumgebung, die ursprünglich von Microsoft entwickelt wurde. Alle .NET-Programme werden in einen Bytecode namens Microsoft Intermediate Language (MSIL) übersetzt. Die MSIL wird von der Common Language Runtime (CLR) ausgeführt.

Im Folgenden finden Sie einige Beispiele für "Hello World" in verschiedenen Sprachen, die .NET Framework unterstützen. "Hello World" ist ein Programm, das "Hello World" auf dem Anzeigegerät anzeigt. Es wird zur Veranschaulichung der grundlegenden Syntax zum Erstellen eines Arbeitsprogramms verwendet. Es kann auch als Sanitätsprüfung verwendet werden, um sicherzustellen, dass der Compiler, die Entwicklungsumgebung und die Laufzeitumgebung einer Sprache ordnungsgemäß funktionieren.

Liste der von .NET unterstützten Sprachen

## Versionen

## .NETZ

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 1,0 | 2002-02-13 |
| 1.1 | 2003-04-24 |
| 2,0 | 2005-11-07 |
| 3,0 | 2006-11-06 |
| 3,5 | 2007-11-19 |
| 3.5 SP1 | 2008-08-11 |
| 4,0 | 2010-04-12 |
| 4,5 | 2012-08-15 |
| 4.5.1 | 2013-10-17 |
| 4.5.2 | 2014-05-05 |
| 4.6 | 2015-07-20 |
| 4.6.1 | 2015-11-17 |

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 4.6.2 | 2016-08-02 |
| 4.7 | 2017-04-05 |

## Kompaktes Framework

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 1,0 | 2000-01-01 |
| 2,0 | 2005-10-01 |
| 3,5 | 2007-11-19 |
| 3.7 | 2009-01-01 |
| 3.9 | 2013-06-01 |

## Micro Framework

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 4.2 | 2011-10-04 |
| 4.3 | 2012-12-04 |
| 4.4 | 2015-10-20 |

## Examples

### Hallo Welt in C #

```
using System;

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

`Console.WriteLine` hat mehrere Überladungen. In diesem Fall ist die Zeichenfolge "Hello World" der Parameter, der die "Hello World" während der Ausführung an den Standardausgangsstrom

ausgibt. Andere Überladungen können den `.ToString` des Arguments `.ToString` bevor in den Stream geschrieben wird. Weitere Informationen finden Sie in der .NET Framework-Dokumentation .

Live-Demo in Aktion bei .NET Fiddle

Einführung in C #

### Hallo Welt in Visual Basic .NET

```
Imports System

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

Live-Demo in Aktion bei .NET Fiddle

Einführung in Visual Basic .NET

### Hallo Welt in F #

```
open System

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

Live-Demo in Aktion bei .NET Fiddle

Einführung in F #

### Hallo Welt in C ++ / CLI

```
using namespace System;

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

### Hallo Welt in PowerShell

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

Einführung in PowerShell

## Hallo Welt in Nemerle

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

## Hallo Welt in Oxygene

```
namespace HelloWorld;

interface

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

implementation

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

end.
```

## Hallo Welt in Boo

```
print "Hello World"
```

## Hallo Welt in Python (IronPython)

```
print "Hello World"
```

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

## Hallo Welt in IL

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

```
  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Erste Schritte mit .NET Framework online lesen: https://riptutorial.com/de/dot-net/topic/14/erste-schritte-mit--net-framework

# Kapitel 2: .NET Core

## Einführung

.NET Core ist eine allgemeine Entwicklungsplattform, die von Microsoft und der .NET-Community auf GitHub verwaltet wird. Es ist plattformübergreifend, unterstützt Windows, Mac OS und Linux und kann in Geräte-, Cloud- und Embedded / IoT-Szenarien verwendet werden.

Wenn Sie an .NET Core denken, sollten Sie Folgendes beachten (flexible Bereitstellung, plattformübergreifende Tools, Befehlszeilenprogramme, Open Source).

Eine weitere tolle Sache ist, dass Microsoft Open Source auch dann aktiv unterstützt, wenn es Open Source ist.

## Bemerkungen

.NET Core enthält ein einziges Anwendungsmodell - Konsolenanwendungen -, das für Tools, lokale Dienste und textbasierte Spiele nützlich ist. Auf Basis von .NET Core wurden zusätzliche Anwendungsmodelle erstellt, um die Funktionalität zu erweitern, z.

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

Außerdem implementiert .NET Core die .NET-Standardbibliothek und unterstützt daher .NET-Standardbibliotheken.

Die .NET-Standardbibliothek ist eine API-Spezifikation, die den konsistenten Satz von .NET-APIs beschreibt, den Entwickler in jeder .NET-Implementierung erwarten können. .NET-Implementierungen müssen diese Spezifikation implementieren, um als mit der .NET-Standardbibliothek kompatibel betrachtet zu werden und um Bibliotheken für die .NET-Standardbibliothek zu unterstützen.

## Examples

### Basic Console App

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

```
    }
}
```

.NET Core online lesen: https://riptutorial.com/de/dot-net/topic/9059/-net-core

# Kapitel 3: Abhängigkeitsspritze

## Bemerkungen

### Probleme, die durch Abhängigkeitseinspritzung gelöst werden

Wenn wir keine Abhängigkeitsinjektion verwendet haben, könnte die `Greeter` Klasse eher wie `Greeter` aussehen:

```
public class ControlFreakGreeter
{
    public void Greet()
    {
        var greetingProvider = new SqlGreetingProvider(
            ConfigurationManager.ConnectionStrings["myConnectionString"].ConnectionString);
        var greeting = greetingProvider.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

Es ist ein "Kontrollfreak", weil er das Erstellen der Klasse, die die Begrüßung bereitstellt, steuert, wo die SQL-Verbindungszeichenfolge herkommt und die Ausgabe steuert.

Mit der Abhängigkeitseinspritzung gibt die `Greeter` Klasse diese Verantwortlichkeiten zugunsten einer einzigen Verantwortung auf und schreibt eine Begrüßung.

Das Prinzip der Abhängigkeitsinversion legt nahe, dass Klassen von Abstraktionen (wie Schnittstellen) und nicht von anderen konkreten Klassen abhängen sollten. Direkte Abhängigkeiten (Kopplung) zwischen Klassen können die Wartung zunehmend erschweren. Abhängig von Abstraktionen kann diese Kopplung reduzieren.

Abhängigkeitsinjektion hilft uns, diese Abhängigkeitsinversion zu erreichen, weil sie dazu führt, dass Klassen geschrieben werden, die von Abstraktionen abhängen. Die `Greeter` Klasse "kennt" die Implementierungsdetails von `IGreetingProvider` und `IGreetingWriter` . Es weiß nur, dass die eingefügten Abhängigkeiten diese Schnittstellen implementieren. Das heißt, Änderungen an den konkreten Klassen, die `IGreetingProvider` und `IGreetingWriter` implementieren, `IGreetingWriter` sich nicht auf `Greeter` . Sie werden auch nicht durch ganz andere Implementierungen ersetzt. Nur Änderungen an den Schnittstellen werden vorgenommen. `Greeter` ist entkoppelt.

`ControlFreakGreeter` nicht richtig durchführen. Wir möchten eine kleine Code-Einheit testen, stattdessen würde unser Test das Herstellen einer Verbindung zu SQL und das Ausführen einer gespeicherten Prozedur umfassen. Dazu gehört auch das Testen der Konsolenausgabe. Da ControlFreakGreeter so viel tut, ist es nicht möglich, es isoliert von anderen Klassen zu testen.

`Greeter` sich leicht als Komponententest testen, da gemockte Implementierungen seiner Abhängigkeiten `Greeter` können, die einfacher auszuführen und zu überprüfen sind, als eine gespeicherte Prozedur aufzurufen oder die Ausgabe der Konsole zu lesen. Es ist keine Verbindungszeichenfolge in app.config erforderlich.

Die konkreten Implementierungen von `IGreetingProvider` und `IGreetingWriter` möglicherweise komplexer. Sie könnten wiederum ihre eigenen Abhängigkeiten haben, die in sie hineingelegt werden. (Zum Beispiel würden wir die SQL-Verbindungszeichenfolge in `SqlGreetingProvider` .) `SqlGreetingProvider` Komplexität wird jedoch von anderen Klassen "verborgen", die nur von den Schnittstellen abhängen. Das macht es einfacher, eine Klasse zu ändern, ohne einen "Ripple-Effekt", der die entsprechenden Änderungen an anderen Klassen erfordert.

# Examples

### Abhängigkeitsinjektion - einfaches Beispiel

Diese Klasse heißt `Greeter` . Ihre Aufgabe ist es, eine Begrüßung auszugeben. Es hat zwei *Abhängigkeiten* . Es braucht etwas, das ihm die Begrüßung zur Ausgabe gibt, und dann braucht es eine Möglichkeit, diese Begrüßung auszugeben. Diese Abhängigkeiten werden beide als Schnittstellen, `IGreetingProvider` und `IGreetingWriter` . In diesem Beispiel werden diese beiden Abhängigkeiten in `Greeter` "injiziert". (Weitere Erläuterungen folgen dem Beispiel.)

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

Die `Greeting` Klasse hängt sowohl von `IGreetingProvider` als auch von `IGreetingWriter` , ist jedoch nicht für das Erstellen von Instanzen verantwortlich. Stattdessen benötigt man sie in seinem Konstruktor. Was auch immer eine `Greeting` muss diese beiden Abhängigkeiten bieten. Wir können das "Injizieren" der Abhängigkeiten nennen.

Da der Klasse Abhängigkeiten in ihrem Konstruktor zur Verfügung gestellt werden, wird dies auch als "Konstruktorinjektion" bezeichnet.

Einige gängige Konventionen:

- Der Konstruktor speichert die Abhängigkeiten als `private` Felder. Sobald die Klasse instanziiert wird, sind diese Abhängigkeiten für alle anderen nicht statischen Methoden der Klasse verfügbar.
- Die `private` Felder sind `readonly` . Sobald sie im Konstruktor festgelegt sind, können sie nicht mehr geändert werden. Dies zeigt an, dass diese Felder nicht außerhalb des Konstruktors geändert werden dürfen (und nicht können). Dies stellt außerdem sicher, dass diese Abhängigkeiten für die Lebensdauer der Klasse verfügbar sind.
- Die Abhängigkeiten sind Schnittstellen. Dies ist nicht unbedingt erforderlich, wird jedoch häufig verwendet, da es einfacher ist, eine Implementierung der Abhängigkeit durch eine andere zu ersetzen. Es ermöglicht auch die Bereitstellung einer abgespeckten Version der Schnittstelle zum Testen von Einheiten.

## Wie Abhängigkeitseinspritzung den Komponententest einfacher macht

Dies baut auf dem vorherigen Beispiel der `Greeter` Klasse auf, die zwei Abhängigkeiten hat, `IGreetingProvider` und `IGreetingWriter` .

Die tatsächliche Implementierung von `IGreetingProvider` kann eine Zeichenfolge von einem API-Aufruf oder einer Datenbank abrufen. Bei der Implementierung von `IGreetingWriter` möglicherweise die Begrüßung in der Konsole angezeigt. Da `Greeter` jedoch Abhängigkeiten in seinen Konstruktor `Greeter` hat, ist es einfach, einen Komponententest zu schreiben, der verspottete Versionen dieser Schnittstellen enthält. In der Praxis könnten wir ein Framework wie Moq verwenden , aber in diesem Fall schreibe ich diese verspotteten Implementierungen.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
```

```
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

Das Verhalten von `IGreetingProvider` und `IGreetingWriter` ist für diesen Test nicht relevant. Wir möchten testen, dass `Greeter` eine Begrüßung erhält und diese schreibt. Der Entwurf von `Greeter` (mithilfe von Abhängigkeitseinspritzung) ermöglicht es uns, komplizierte Abhängigkeiten ohne komplizierte bewegliche Teile einzuspritzen. Alles, was wir testen, ist, dass `Greeter` mit diesen Abhängigkeiten so interagiert, wie wir es erwarten.

## Warum wir Abhängigkeitsinjektionscontainer (IoC-Container) verwenden

Abhängigkeitsinjektion bedeutet, Klassen so zu schreiben, dass sie ihre Abhängigkeiten nicht steuern - stattdessen werden ihnen ihre Abhängigkeiten zur Verfügung gestellt ("injiziert".)

Dies ist nicht das Gleiche wie das Verwenden eines Abhängigkeitsinjektions-Frameworks (oft als "DI-Container", "IoC-Container" oder nur "Container" bezeichnet) wie Castle Windsor, Autofac, SimpleInjector, Ninject, Unity oder andere.

Ein Container erleichtert die Abhängigkeitsinjektion. Nehmen Sie beispielsweise an, Sie schreiben eine Reihe von Klassen, die auf Abhängigkeiten angewiesen sind. Eine Klasse hängt von mehreren Schnittstellen ab, die Klassen, die diese Schnittstellen implementieren, hängen von anderen Schnittstellen ab und so weiter. Einige hängen von bestimmten Werten ab. Und nur zum Spaß implementieren einige dieser Klassen `IDisposable` und müssen entsorgt werden.

Jede einzelne Klasse ist gut geschrieben und leicht zu testen. Nun gibt es jedoch ein anderes Problem: Das Erstellen einer Instanz einer Klasse ist viel komplizierter geworden. Angenommen, wir erstellen eine Instanz einer `CustomerService` Klasse. Es hat Abhängigkeiten und seine Abhängigkeiten haben Abhängigkeiten. Das Konstruieren einer Instanz könnte ungefähr so aussehen:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint =
ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
```

```
}
```

Sie fragen sich vielleicht, warum nicht die gesamte Riesenkonstruktion in eine separate Funktion gestellt wird, die nur `CustomerService` zurückgibt. Ein Grund ist, dass eine Klasse nicht dafür verantwortlich ist, ob die Abhängigkeiten für jede Klasse in diese Klasse `IDisposable` sind, und nicht dafür verantwortlich ist, ob diese Abhängigkeiten `IDisposable` oder ob sie entsorgt werden sollen. Es benutzt sie nur. Wenn also eine `GetCustomerService()` Funktion vorhanden war, die einen vollständig erstellten `CustomerService` , enthält diese Klasse möglicherweise eine Reihe verfügbarer Ressourcen und keine Möglichkeit, auf sie zuzugreifen oder sie zu löschen.

Und abgesehen davon, `IDisposable` entsorgen, wer möchte schon immer eine Reihe verschachtelter Konstruktoren nennen? Das ist ein kurzes Beispiel. Es könnte viel schlimmer werden. Das bedeutet auch nicht, dass wir die Klassen falsch geschrieben haben. Der Unterricht kann individuell perfekt sein. Die Herausforderung besteht darin, sie zusammenzusetzen.

Ein Abhängigkeitsinjektionsbehälter vereinfacht das. Damit können wir festlegen, welche Klasse oder welcher Wert verwendet werden soll, um jede Abhängigkeit zu erfüllen. Dieses etwas zu vereinfachte Beispiel verwendet Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

Wir nennen dies "Abhängigkeiten registrieren" oder "Container konfigurieren". Übersetzt sagt dies unserem `WindsorContainer` :

- Wenn eine Klasse `ILogWriter` erfordert, erstellen Sie eine Instanz von `LogWriter` . `LogWriter` erfordert einen Dateipfad. Verwenden Sie diesen Wert aus `AppSettings` .
- Wenn eine Klasse `IAuthorizationRepository` erfordert, erstellen Sie eine Instanz von `SqlAuthorizationRepository` . Es erfordert eine Verbindungszeichenfolge. Verwenden Sie diesen Wert aus dem Abschnitt `ConnectionStrings` .
- Wenn eine Klasse `ICustomerDataProvider` erfordert, erstellen Sie einen `CustomerApiClient` und geben Sie die erforderliche Zeichenfolge aus `AppSettings` .

Wenn wir eine Abhängigkeit vom Container anfordern, nennen wir das "Auflösen" einer Abhängigkeit. Es ist eine schlechte Praxis, dies direkt mit dem Container zu tun, aber das ist eine andere Geschichte. Zu Demonstrationszwecken könnten wir jetzt Folgendes tun:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

Der Container weiß, dass `CustomerService` von `IAuthorizationRepository` und `ICustomerDataProvider` . Es weiß, welche Klassen erstellt werden müssen, um diese Anforderungen zu erfüllen. Diese Klassen haben wiederum mehr Abhängigkeiten, und der Container kann diese erfüllen. Es erstellt jede Klasse, die es benötigt, bis es eine Instanz von `CustomerService` .

Wenn es zu einem Punkt kommt, an dem eine Klasse eine Abhängigkeit erfordert, die wir nicht registriert haben, wie `IDoesSomethingElse` , wird beim Versuch, `CustomerService` zu lösen, eine eindeutige Ausnahme `IDoesSomethingElse` , die uns darüber `IDoesSomethingElse` , dass wir nichts registriert haben, um diese Anforderung zu erfüllen.

Jedes DI-Framework verhält sich ein wenig anders, aber in der Regel können Sie steuern, wie bestimmte Klassen instanziiert werden. `LogWriter` Sie beispielsweise, dass eine Instanz von `LogWriter` und jeder Klasse `ILogWriter` , die von `ILogWriter` abhängt, oder möchten Sie, dass jedes Mal eine neue erstellt wird? Die meisten Container haben eine Möglichkeit, dies anzugeben.

Was ist mit Klassen, die `IDisposable` implementieren? Deshalb nennen wir `container.Release(customerService);` Am Ende. Die meisten Behälter (einschließlich Windsor) wird durch alle Abhängigkeiten Schritt zurück erstellt und `Dispose` diejenigen , die Beseitigung benötigen. Wenn `CustomerService` `IDisposable` , wird auch dies entsorgt.

Das Registrieren von Abhängigkeiten, wie oben gezeigt, kann wie mehr zu schreibender Code aussehen. Aber wenn wir viele Klassen mit vielen Abhängigkeiten haben, lohnt es sich wirklich. Und wenn wir dieselben Klassen *ohne* Abhängigkeitseinspritzung schreiben müssten, wäre es schwierig, dieselbe Anwendung mit vielen Klassen zu warten und zu testen.

Dies verkratzt die Oberfläche, *warum* wir Abhängigkeitsinjektionsbehälter verwenden. *Wie* wir unsere Anwendung für die Verwendung einer Anwendung konfigurieren (und sie korrekt verwenden), ist nicht nur ein Thema, sondern eine Reihe von Themen, da die Anweisungen und Beispiele von Container zu Container unterschiedlich sind.

Abhängigkeitsspritze online lesen: https://riptutorial.com/de/dot-net/topic/5085/abhangigkeitsspritze

# Kapitel 4: ADO.NET

## Einführung

ADO (ActiveX Data Objects) .Net ist ein Tool von Microsoft, das über seine Komponenten Zugriff auf Datenquellen wie SQL Server, Oracle und XML bietet. .NET-Front-End-Anwendungen können Daten abrufen, erstellen und bearbeiten, sobald sie über ADO.Net mit entsprechenden Datenrechten mit einer Datenquelle verbunden sind.

ADO.Net bietet eine verbindungslose Architektur. Dies ist ein sicherer Ansatz für die Interaktion mit einer Datenbank, da die Verbindung nicht während der gesamten Sitzung aufrechterhalten werden muss.

## Bemerkungen

**Ein Hinweis zur Parametrisierung von SQL mit `Parameters.AddWithValue` :** `AddWithValue` ist niemals ein guter Ausgangspunkt. Diese Methode beruht darauf, dass der Datentyp aus den übergebenen Daten abgeleitet wird. Dadurch kann es vorkommen, dass die Konvertierung die Verwendung eines Index durch die Konvertierung verhindert. Beachten Sie, dass einige SQL Server-Datentypen, beispielsweise `char` / `varchar` (ohne vorangestelltes "n") oder `date` , keinen entsprechenden .NET-Datentyp haben. In diesen Fällen sollte stattdessen `Add` mit dem richtigen Datentyp verwendet werden .

## Examples

### SQL-Anweisungen als Befehl ausführen

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

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

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

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

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

```
    // those have been added).

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

***Hinweis 1: Die*** für MSFT SQL Server spezifische Variation finden Sie unter SqlDbType-Aufzählung .

***Hinweis 2:*** Die MySQL-spezifische Variante finden Sie unter MySqlDbType-Enumeration .

## Best Practices - SQL-Anweisungen ausführen

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

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

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

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

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

```
    }
}

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

---

## Best Practice für die Arbeit mit ADO.NET

- Als Faustregel gilt, die Verbindung für minimale Zeit zu öffnen. Schließen Sie die Verbindung explizit, sobald Ihre Prozedur ausgeführt wurde. Dadurch wird das Verbindungsobjekt wieder in den Verbindungspool zurückgegeben. Max. Größe des Standardverbindungspools = 100. Durch das Verbindungspooling wird die Leistung der physischen Verbindung zu SQL Server verbessert. Verbindungspooling in SQL Server
- Binden Sie alle Datenbankverbindungen in einen using-Block ein, sodass sie auch im Falle einer Ausnahme immer geschlossen und entsorgt werden. Weitere Informationen zur Verwendung von Anweisungen finden Sie unter using-Anweisung (C # -Referenz)
- Rufen Sie die Verbindungszeichenfolgen nach Namen aus der Datei app.config oder web.config ab (abhängig vom Anwendungstyp).
    - Dies erfordert einen Assemblyverweis auf `System.configuration`
    - Weitere Informationen zum Strukturieren Ihrer Konfigurationsdatei finden Sie unter Verbindungszeichenfolgen und Konfigurationsdateien
- Verwenden Sie immer Parameter für eingehende Werte bis
    - Vermeiden Sie SQL-Injektionsangriffe
    - Vermeiden Sie Fehler, wenn fehlerhafter Text verwendet wird, z. B. ein einfaches Anführungszeichen. Dies ist das SQL-Äquivalent für das Escape-Zeichen oder das Starten einer Zeichenfolge (varchar / nvarchar).
    - Durch die Wiederverwendung von Abfrageplänen durch den Datenbankanbieter (nicht von allen Datenbankanbietern unterstützt) wird die Effizienz erhöht
- Beim Arbeiten mit Parametern
    - Typ und Größe der SQL-Parameter sind eine häufige Ursache für Fehler beim Einfügen / Aktualisieren / Auswählen
    - Geben Sie Ihren SQL-Parametern sinnvolle Namen, genau wie Sie Variablen in Ihrem Code verwenden
    - Geben Sie den Datenbankdatentyp der von Ihnen verwendeten Spalte an. Dadurch wird sichergestellt, dass keine falschen Parametertypen verwendet werden, die zu unerwarteten Ergebnissen führen können
    - Bestätigen Sie Ihre eingehenden Parameter, bevor Sie sie an den Befehl übergeben (wie das Sprichwort sagt: " Müll rein, Müll raus "). Überprüfen Sie eingehende Werte so früh wie möglich im Stapel
    - Verwenden Sie beim Zuweisen Ihrer Parameterwerte die richtigen Typen. Beispiel: Weisen Sie nicht den Zeichenfolgenwert einer DateTime zu, sondern weisen Sie dem Wert des Parameters eine tatsächliche DateTime-Instanz zu

    - Geben Sie die Größe der Zeichenfolgentypparameter an. Dies liegt daran, dass SQL Server Ausführungspläne wiederverwenden kann, wenn die Parameter in Typ *und* Größe übereinstimmen. Verwenden Sie -1 für MAX
    - Verwenden Sie nicht die Methode AddWithValue . Der Hauptgrund ist, dass es sehr leicht ist, bei Bedarf den Parametertyp oder die Genauigkeit / Skalierung anzugeben. Weitere Informationen finden Sie unter Können AddWithValue bereits eingestellt werden?
- Bei Verwendung von Datenbankverbindungen
    - Öffnen Sie die Verbindung so spät wie möglich und schließen Sie sie so bald wie möglich. Dies ist eine allgemeine Richtlinie, wenn Sie mit externen Ressourcen arbeiten
    - Teilen Sie niemals Datenbankverbindungsinstanzen gemeinsam (Beispiel: Ein Singleton-Host hat eine gemeinsam genutzte Instanz vom Typ `SqlConnection` ). Lassen Sie Ihren Code bei Bedarf immer eine neue Datenbankverbindungsinstanz erstellen, und dann den aufrufenden Code entsorgen und "wegwerfen", wenn er fertig ist. Der Grund dafür ist
        - Die meisten Datenbankanbieter verfügen über eine Art Verbindungspooling, sodass das Erstellen neuer verwalteter Verbindungen kostengünstig ist
        - Es beseitigt zukünftige Fehler, wenn der Code mit mehreren Threads arbeitet

## Verwenden gemeinsamer Schnittstellen, um herstellerspezifische Klassen zu isolieren

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

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

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

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

ADO.NET online lesen: https://riptutorial.com/de/dot-net/topic/3589/ado-net

# Kapitel 5: Akronym Glossar

## Examples

### .Net-bezogene Akronyme

Bitte beachten Sie, dass einige Begriffe wie JIT und GC generisch genug sind, um auf viele Programmiersprachenumgebungen und Laufzeiten angewendet zu werden.

CLR: Common Language Runtime
IL: Zwischensprache
EE: Execution Engine
JIT: Just-in-Time-Compiler
GC: Müllsammler
OOM: Nicht genügend Speicherplatz
STA: Wohnung mit einem Thread
MTA: Apartment mit mehreren Threads

Akronym Glossar online lesen: https://riptutorial.com/de/dot-net/topic/10939/akronym-glossar

# Kapitel 6: Ausdrucksbäume

## Bemerkungen

Ausdrucksbäume sind Datenstrukturen, die zur Darstellung von Codeausdrücken in .NET Framework verwendet werden. Sie können per Code generiert und programmgesteuert durchlaufen werden, um den Code in eine andere Sprache zu übersetzen oder auszuführen. Der bekannteste Generator von Expression Trees ist der C # -Compiler selbst. Der C # -Compiler kann Ausdrucksbäume generieren, wenn einer Variablen des Typs Ausdruck <Func <... >> ein Lambda-Ausdruck zugewiesen wird. Normalerweise geschieht dies im Zusammenhang mit LINQ. Der beliebteste Verbraucher ist der LINQ-Anbieter von Entity Framework. Es verbraucht die Ausdrucksbäume, die an Entity Framework übergeben werden, und generiert entsprechenden SQL-Code, der dann für die Datenbank ausgeführt wird.

## Examples

### Einfache Ausdrucksbaumstruktur, die vom C # -Compiler generiert wird

Betrachten Sie den folgenden C # -Code

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

Da der C # -Compiler erkennt, dass der Lambda-Ausdruck einem Ausdruckstyp und nicht einem Delegatentyp zugeordnet ist, generiert er eine Ausdrucksstruktur, die diesem Code in etwa entspricht

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

Die Wurzel des Baums ist der Lambda-Ausdruck, der einen Hauptteil und eine Liste von Parametern enthält. Das Lambda hat 1 Parameter namens "a". Der Hauptteil ist ein einzelner Ausdruck des CLR-Typs BinaryExpression und NodeType of Add. Dieser Ausdruck steht für die Addition. Es gibt zwei Unterausdrücke, die als Left und Right bezeichnet werden. Left ist die ParameterExpression für den Parameter "a" und Right ist eine ConstantExpression mit dem Wert 1.

Die einfachste Verwendung dieses Ausdrucks ist das Ausdrucken:

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

Welche gibt den entsprechenden C # -Code aus.

Der Ausdrucksbaum kann in einen C # -Delegierten kompiliert und von der CLR ausgeführt werden

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

Normalerweise werden Ausdrücke in andere Sprachen wie SQL übersetzt, können aber auch zum Aufrufen von privaten, geschützten und internen Mitgliedern von öffentlichen oder nicht öffentlichen Typen als Alternative zu Reflection verwendet werden.

## Erstellen eines Prädikats von Formularfeld == Wert

Um einen Ausdruck wie `_ => _.Field == "VALUE"` zur Laufzeit `_ => _.Field == "VALUE"` .

`_ => _.Field` ein Prädikat `_ => _.Field` und einen Zeichenfolgenwert `"VALUE"` einen Ausdruck, der prüft, ob das Prädikat wahr ist.

Der Ausdruck ist geeignet für:

- `IQueryable<T>` , `IEnumerable<T>` , um das Prädikat zu testen.
- Entity Framework oder `Linq` to `SQL` , um eine `Where` Klausel zu erstellen, die das Prädikat testet.

Diese Methode erstellt einen entsprechenden `Equal` Ausdruck, der prüft, ob `Field` gleich `"VALUE"` oder nicht.

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

Das Prädikat kann verwendet werden, indem das Prädikat in eine `Where` Erweiterungsmethode eingeschlossen wird.

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

## Ausdruck zum Abrufen eines statischen Feldes

Beispiel-Typ wie folgt eingeben:

```
public TestClass
```

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

Wir können den Wert von StaticPublicField abrufen:

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

Es kann dann zB in einen Delegierten zum Abrufen des Feldwerts kompiliert werden.

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

// Das Ergebnis von fieldValue ist StaticPublicFieldValue

## InvocationExpression-Klasse

Die InvocationExpression-Klasse ermöglicht das Aufrufen anderer Lambda-Ausdrücke, die Teile des gleichen Ausdrucksbaums sind.

Sie erstellen sie mit der statischen `Expression.Invoke` Methode.

Problem Wir möchten auf die Artikel zugreifen, die in ihrer Beschreibung "Auto" haben. Wir müssen den Wert auf null überprüfen, bevor Sie nach einer Zeichenfolge suchen. Wir möchten jedoch nicht, dass er übermäßig aufgerufen wird, da die Berechnung teuer sein kann.

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

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

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

        Console.WriteLine(elementIsInterestingExpression.ToString());

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

```
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

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

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

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

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

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

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

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

Ausgabe

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

Das erste, was zu beachten ist, ist, wie die eigentliche Eigenschaft, in eine Invoke gehüllt, zugreift:

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

, und dies ist der einzige Teil, der `e.Description` berührt, und anstelle dessen wird der `extracted` Parameter vom Typ `string` an den nächsten übergeben:

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

Eine weitere wichtige Sache, die Sie hier beachten sollten, ist `AndAlso` . Es berechnet nur den linken Teil, wenn der erste Teil 'false' zurückgibt. Es ist ein häufiger Fehler, den bitweisen Operator 'And' anstelle von diesem zu verwenden, der immer beide Teile berechnet und in diesem Beispiel mit einer NullReferenceException fehlschlägt.

Ausdrucksbäume online lesen: https://riptutorial.com/de/dot-net/topic/2657/ausdrucksbaume

# Kapitel 7: Ausnahmen

## Bemerkungen

Verbunden:

- MSDN: Ausnahmen und Ausnahmebehandlung (C # -Programmierhandbuch)
- MSDN: Behandeln und Auslösen von Ausnahmen
- MSDN: CA1031: Allgemeine Ausnahmetypen nicht abfangen
- MSDN: Try-Catch (C # -Referenz)

## Examples

### Eine Ausnahme abfangen

Code kann und sollte in Ausnahmefällen Ausnahmen auslösen. Beispiele hierfür sind:

- Versuch, über das Ende eines Streams zu lesen
- Sie verfügen nicht über die erforderlichen Berechtigungen, um auf eine Datei zuzugreifen
- Versuch, eine ungültige Operation auszuführen, beispielsweise durch Nullen
- Ein Timeout beim Herunterladen einer Datei aus dem Internet

Der Anrufer kann mit diesen Ausnahmen umgehen, indem er sie "fängt", und sollte dies nur tun, wenn:

- Sie kann den außergewöhnlichen Umstand tatsächlich lösen oder sich angemessen erholen, oder;
- Es kann zusätzlichen Kontext für die Ausnahme bereitstellen, der nützlich wäre, wenn die Ausnahme erneut ausgelöst werden muss (erneut geworfene Ausnahmen werden von Ausnahmebehandlern weiter oben im Aufrufstapel abgefangen.)

Es ist zu beachten, dass die Entscheidung, eine Ausnahme *nicht* abzufangen, vollkommen gültig ist, wenn beabsichtigt ist, auf einer höheren Ebene behandelt zu werden.

Das Abfangen einer Ausnahme erfolgt, indem der potenziell auslösende Code wie folgt in einen `try { ... }` Block eingeschlossen wird und die Ausnahmen erfasst werden, die in einem `catch (ExceptionType) { ... }` -Block behandelt werden können:

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

Stream fileStream;

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

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

### Mit einem endgültigen Block

Der `finally { ... }` -Block eines `try-finally` oder `try-catch-finally` wird immer ausgeführt, unabhängig davon, ob eine Ausnahmebedingung aufgetreten ist oder nicht (außer wenn eine `StackOverflowException` ausgelöst wurde oder ein Aufruf an `Environment.FailFast()` ).

Es kann verwendet werden, um Ressourcen, die im `try { ... }` -Block gesichert wurden, sicher freizugeben oder zu bereinigen.

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

Stream fileStream = null;

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

### Gefangene Ausnahmen fangen und erneut werfen

Wenn Sie eine Ausnahme abfangen und etwas tun möchten, die Ausführung des aktuellen Codeblocks jedoch aufgrund der Ausnahme nicht fortsetzen können, möchten Sie möglicherweise die Ausnahme für den nächsten Ausnahmebehandler im Aufrufstapel erneut auslösen. Es gibt gute und schlechte Wege, dies zu tun.

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

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
```

```
        // we do not need a reference to the exception
        Console.WriteLine("Dividing by zero would destroy the universe.");

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

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

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

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

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

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

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

Sie können nach Ausnahme-Typ und sogar nach Ausnahme-Eigenschaften filtern (neu in C # 6.0, etwas länger in VB.NET verfügbar (Zitat erforderlich)):

Dokumentation / C # / neue Funktionen

## Ausnahmefilter

Seit C # 6.0 können Ausnahmen mit dem Operator `when` gefiltert werden.

Dies ist vergleichbar mit der Verwendung eines einfachen `if` , der den Stack jedoch nicht abwickelt, wenn die Bedingung innerhalb des `when` nicht erfüllt ist.

**Beispiel**

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

Die gleichen Informationen finden Sie in den C # 6.0-Features hier: Ausnahmefilter

## Eine Ausnahme innerhalb eines catch-Blocks erneut auslösen

Innerhalb eines `catch` Blocks kann das `throw` Schlüsselwort alleine verwendet werden, ohne einen Ausnahmewert anzugeben, um die gerade abgefangene Ausnahme erneut *auszulösen* . Beim erneuten Auslösen einer Ausnahme kann die ursprüngliche Ausnahme die Ausnahmebehandlungskette fortsetzen, wobei der Aufrufstack oder die zugehörigen Daten erhalten bleiben:

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

Ein häufiges Anti-Pattern ist das `throw ex` , was die Ansicht des Stack-Trace des nächsten Exception-Handlers einschränkt:

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

Im Allgemeinen ist die Verwendung von `throw ex` nicht wünschenswert, da zukünftige Ausnahmebehandler, die die Stack-Ablaufverfolgung untersuchen, nur Aufrufe bis zu `throw ex` . Wenn Sie die `ex` Variable weglassen und nur das `throw` Schlüsselwort verwenden, sprudelt die ursprüngliche Ausnahme.

## Eine Ausnahme von einer anderen Methode auslösen und dabei die Informationen beibehalten

Gelegentlich möchten Sie eine Ausnahme abfangen und aus einem anderen Thread oder einer anderen Methode werfen, während der ursprüngliche Ausnahmestapel erhalten bleibt. Dies kann mit `ExceptionDispatchInfo` :

```
using System.Runtime.ExceptionServices;

void Main()
{
```

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

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

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

Ausnahmen online lesen: https://riptutorial.com/de/dot-net/topic/33/ausnahmen

# Kapitel 8: Benutzerdefinierte Typen

## Bemerkungen

Normalerweise wird eine `struct` nur verwendet, wenn die Leistung sehr wichtig ist. Da Werttypen auf dem Stapel leben, kann auf sie viel schneller zugegriffen werden als auf Klassen. Der Stapel hat jedoch viel weniger Speicherplatz als der Heap, daher sollten die `struct` klein gehalten werden (Microsoft empfiehlt, dass die `struct` nicht mehr als 16 Byte beansprucht).

Eine `class` ist der meistgenutzte (von diesen drei) Typ in C # und sollte im Allgemeinen zuerst verwendet werden.

Eine `enum` wird immer dann verwendet, wenn Sie eine klar definierte, eindeutige Liste von Elementen haben, die nur einmal (zur Kompilierzeit) definiert werden müssen. Enumerationen sind für Programmierer hilfreich als einfache Referenz für einen bestimmten Wert: Anstatt eine Liste von `constant` Variablen für den Vergleich zu definieren, können Sie eine Enumeration verwenden und die Intellisense-Unterstützung in Anspruch nehmen, um sicherzustellen, dass Sie nicht versehentlich einen falschen Wert verwenden.

## Examples

### Strukturdefinition

## Strukturen, die von System.ValueType erben, sind Werttypen und leben auf dem Stack. Wenn Werttypen als Parameter übergeben werden, werden sie als Wert übergeben.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

**Übergeben durch Wert** bedeutet, dass der Wert des Parameters für die Methode *kopiert* wird und Änderungen, die an dem Parameter in der Methode vorgenommen werden, nicht außerhalb der Methode angezeigt werden. Betrachten Sie zum Beispiel den folgenden Code, der eine Methode namens `AddNumbers` , die die Variablen `a` und `b` übergibt, die vom Typ `int` sind, also vom Typ Value.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
```

```
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
    z = x + y; // now z becomes 16
}
```

Auch wenn wir 5 bis hinzugefügt `x` innerhalb der Methode, der Wert von `a` bleibt unverändert, da es sich um ein Werttyp ist, und das bedeutet , `x` eine *Kopie* war `a` ,s - Wert, aber nicht wirklich `a` .

Denken Sie daran, dass Werttypen auf dem Stapel leben und als Wert übergeben werden.

### Klassendefinition

## Klassen, die von System.Object erben, sind Referenztypen und leben auf dem Heap. Wenn Referenztypen als Parameter übergeben werden, werden sie als Referenz übergeben.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

**Übergeben als Referenz** bedeutet, dass eine *Referenz* auf den Parameter an die Methode übergeben wird und alle Änderungen an dem Parameter bei der Rückgabe außerhalb der Methode angezeigt werden, da die Referenz *auf dasselbe Objekt im Speicher* verweist. Verwenden wir dasselbe Beispiel wie zuvor, aber wir "wickeln" die `int` s zuerst in einer Klasse.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

Dieses Mal, wenn wir geändert `sample.a` bis `10` , der Wert von `instanceOfMyClass.a` ändert sich *auch,* weil sie *als Referenz übergeben* wurde. Übergeben durch Referenz bedeutet, dass eine *Referenz* (manchmal auch als *Zeiger bezeichnet* ) auf das Objekt anstelle einer Kopie des Objekts selbst an die Methode übergeben wurde.

Denken Sie daran, dass Verweistypen auf dem Heap leben und als Verweise übergeben werden.

### Aufzählungsdefinition

## Eine Enumeration ist eine besondere Klasse. Das Schlüsselwort `enum` teilt dem Compiler mit, dass diese Klasse von der abstrakten System.Enum-Klasse erbt. Aufzählungen werden für verschiedene Listen von Elementen verwendet.

```
public enum MyEnum
{
    Monday = 1,
    Tuesday,
    Wednesday,
    //...
}
```

Sie können sich eine Aufzählung als eine bequeme Möglichkeit vorstellen, Konstanten einem zugrunde liegenden Wert zuzuordnen. Die oben definierte Aufzählung deklariert Werte für jeden Wochentag und beginnt mit `1` . `Tuesday` würde dann automatisch auf `2` abgebildet, `Wednesday` auf `3` usw.

Standardmäßig verwenden Aufzählungen `int` als zugrunde liegenden Typ und beginnen bei 0, Sie können jedoch einen der folgenden *ganzzahligen Typen verwenden* : `byte, sbyte, short, ushort, int, uint, long, or ulong` und können explizite Werte für any angeben Artikel. Wenn einige Elemente explizit angegeben sind, andere jedoch nicht, wird jedes Element nach dem zuletzt definierten Element um 1 erhöht.

Wir würden dieses Beispiel verwenden , indem Sie einen anderen Wert zu einem *MyEnum* wie so *Gießen:*

```
MyEnum instance = (MyEnum)3; // the variable named 'instance' gets a
                             //value of MyEnum.Wednesday, which maps to 3.

int x = 2;
instance = (MyEnum)x; // now 'instance' has a value of MyEnum.Tuesday
```

Eine andere nützliche, wenn auch komplexere Art der Aufzählung wird `Flags` . Durch das *Dekorieren* einer Aufzählung mit dem `Flags` Attribut können Sie einer Variablen mehrere Werte gleichzeitig zuweisen. Beachten Sie, dass Sie dabei explizit Werte in der Basis-2-Darstellung definieren *müssen* .

```
[Flags]
public enum MyEnum
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    Saturday = 32,
    Sunday = 64
}
```

Jetzt können Sie mehr als einen Wert gleichzeitig vergleichen, entweder mit *bitweisen Vergleichen* oder, wenn Sie .NET 4.0 oder höher verwenden, die integrierte `Enum.HasFlag` Methode.

```
MyEnum instance = MyEnum.Monday | MyEnum.Thursday; // instance now has a value of
                                                   // *both* Monday and Thursday,
                                                   // represented by (in binary) 0100.

if (instance.HasFlag(MyEnum.Wednesday))
{
    // it doesn't, so this block is skipped
}
else if (instance.HasFlag(MyEnum.Thursday))
{
    // it does, so this block is executed
}
```

Da die Enum-Klasse von `System.ValueType` , wird sie als `System.ValueType` behandelt und als Wert übergeben, nicht als Referenz. Das Basisobjekt wird auf dem Heap erstellt. Wenn Sie jedoch einen Aufzählungswert an einen Funktionsaufruf übergeben, wird eine Kopie des Werts mit dem zugrunde liegenden Wertetyp der Aufzählung (normalerweise System.Int32) in den Stapel geschrieben. Der Compiler verfolgt die Zuordnung zwischen diesem Wert und dem Basisobjekt, das auf dem Stapel erstellt wurde. Weitere Informationen finden Sie unter ValueType-Klasse (System) (MSDN) .

Benutzerdefinierte Typen online lesen: https://riptutorial.com/de/dot-net/topic/57/benutzerdefinierte-typen

# Kapitel 9: CLR

## Examples

### Eine Einführung in die Common Language Runtime

Die **Common Language Runtime (CLR)** ist eine virtuelle Maschinenumgebung und Teil von .NET Framework. Es beinhaltet:

- Eine portable Bytecode-Sprache, die als **Common Intermediate Language** (abgekürzt CIL oder IL) bezeichnet wird.
- Ein Just-In-Time-Compiler, der Maschinencode generiert
- Ein Trace-Garbage-Collector, der eine automatische Speicherverwaltung ermöglicht
- Unterstützung für leichte Unterprozesse, die als AppDomains bezeichnet werden
- Sicherheitsmechanismen durch die Konzepte nachprüfbaren Codes und Vertrauensniveaus

Code, der in der CLR ausgeführt wird, wird als *verwalteter Code bezeichnet* , um ihn von *Code* zu unterscheiden, der außerhalb der CLR (normalerweise nativer Code) ausgeführt wird, der als nicht *verwalteter Code bezeichnet wird* . Es gibt verschiedene Mechanismen, die die Interoperabilität zwischen verwaltetem und nicht verwaltetem Code erleichtern.

CLR online lesen: https://riptutorial.com/de/dot-net/topic/3942/clr

# Kapitel 10: Code-Verträge

## Bemerkungen

Codeverträge ermöglichen die Analyse oder Analyse von Vor- und Nachbedingungen von Methoden sowie von unveränderlichen Bedingungen für Objekte. Diese Bedingungen können verwendet werden, um sicherzustellen, dass Anrufer und Rückgabewert mit gültigen Status für die Anwendungsverarbeitung übereinstimmen. Andere Verwendungen für Codeverträge umfassen die Dokumentationsgenerierung.

## Examples

### Voraussetzungen

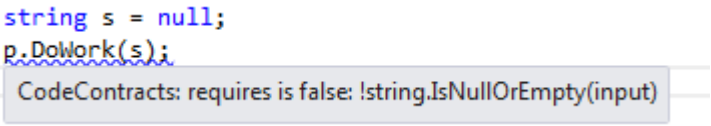
Vorbedingungen ermöglicht es den Methoden, minimal erforderliche Werte für Eingabeparameter bereitzustellen

**Beispiel...**

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

    //do work
}
```

**Statisches Analyseergebnis ...**




### Nachbedingungen

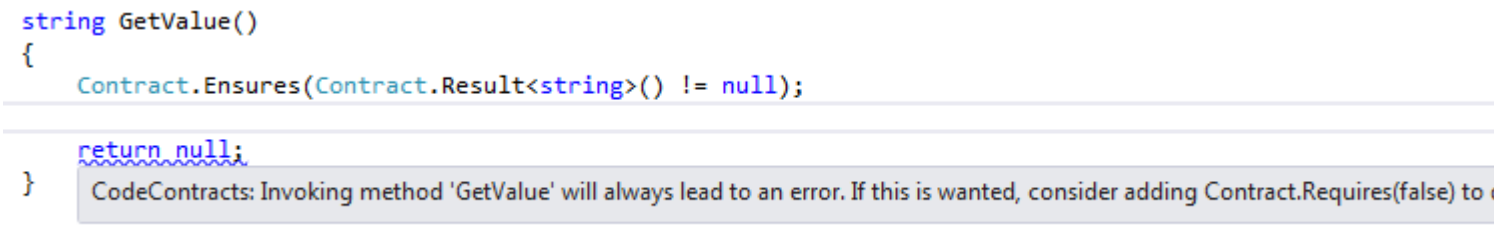
Nachbedingungen stellen sicher, dass die zurückgegebenen Ergebnisse einer Methode mit der bereitgestellten Definition übereinstimmen. Dadurch erhält der Anrufer eine Definition des erwarteten Ergebnisses. Nachbedingungen können vereinfachte Implementierungen zulassen, da der statische Analysator einige mögliche Ergebnisse liefern kann.

**Beispiel...**

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

    return null;
}
```

**Statische Analyse Ergebnis ...**




## Verträge für Schnittstellen

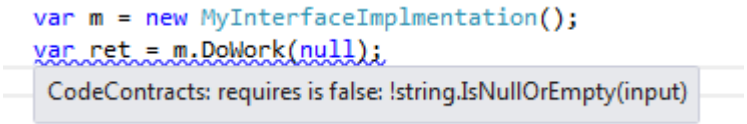
Mit Codeverträgen ist es möglich, einen Vertrag auf eine Schnittstelle anzuwenden. Dazu wird eine abstrakte Klasse deklariert, die die Schnittstellen implementiert. Die Schnittstelle sollte mit dem `ContractClassAttribute` und die Vertragsdefinition (die abstrakte Klasse) mit dem `ContractClassForAttribute`

**C # Beispiel ...**

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

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

**Statisches Analyseergebnis ...**

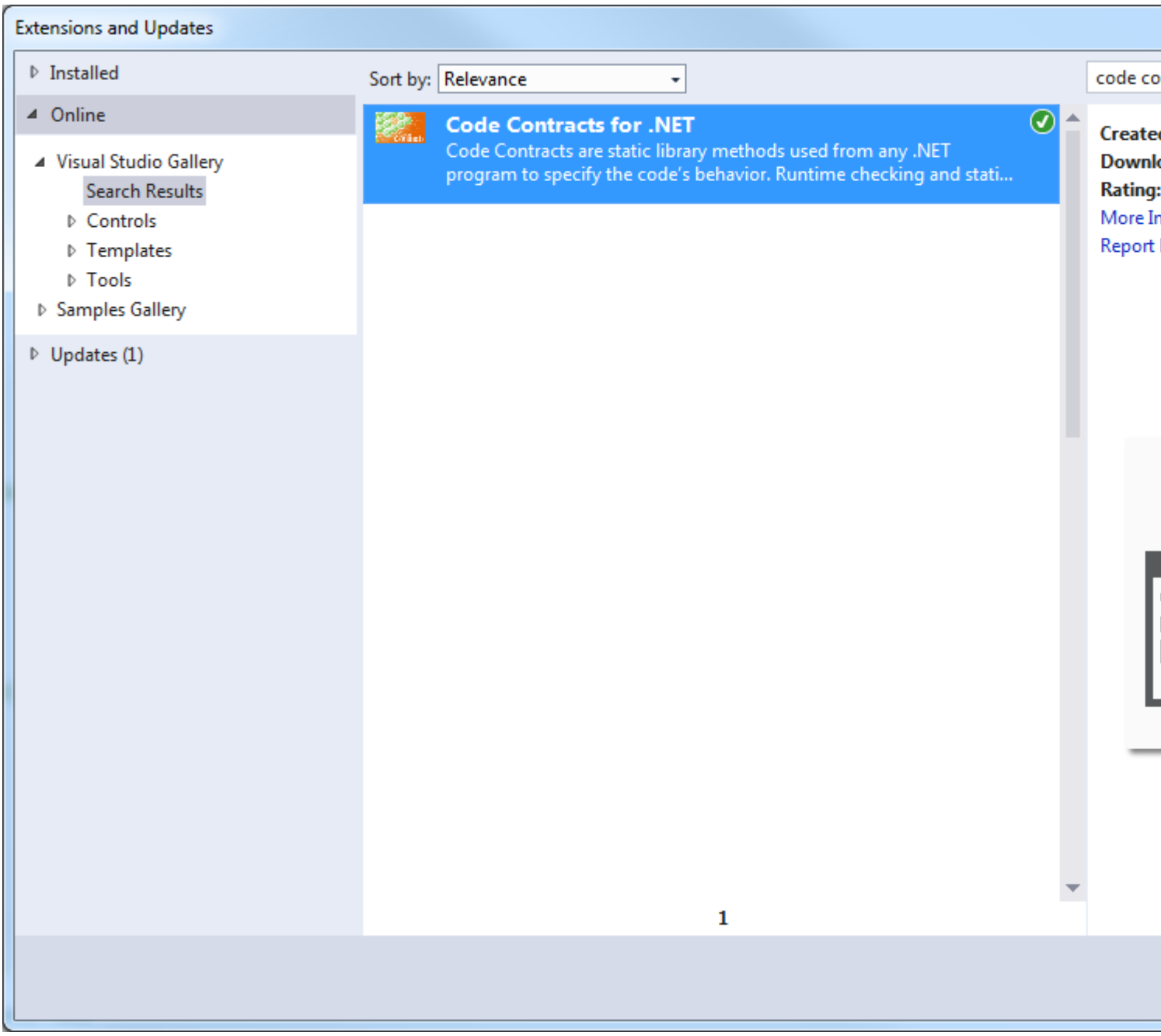



## Codeverträge installieren und aktivieren

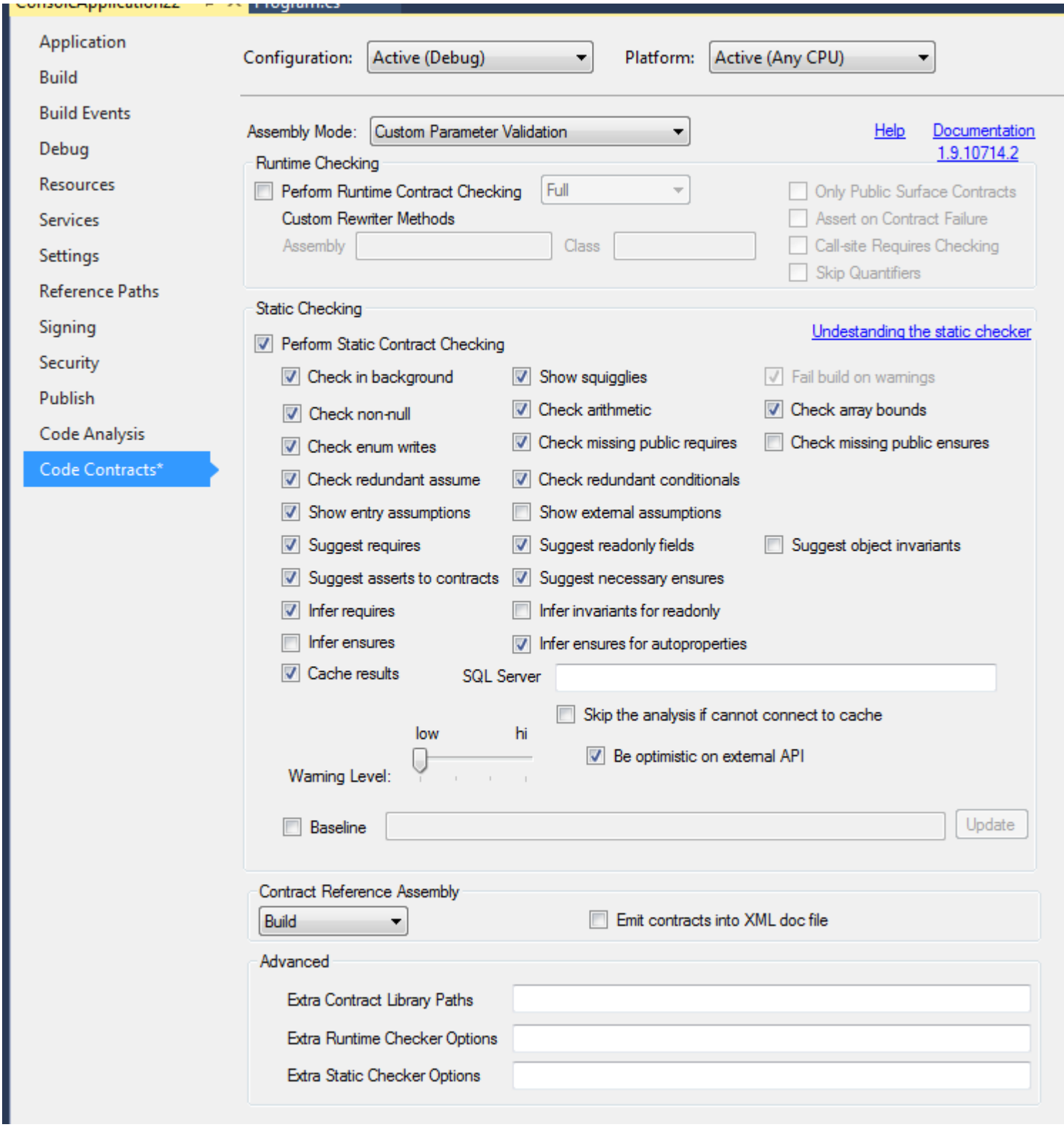
Während `System.Diagnostics.Contracts` im .NET Framework enthalten ist. Um Code Contracts

verwenden zu können, müssen Sie die Visual Studio-Erweiterungen installieren.

Suchen Sie unter `Extensions and Updates` nach `Code Contracts` und installieren Sie dann die `Code Contracts Tools`




Nach der Installation der Tools müssen Sie `Code Contracts` in Ihrer Project-Lösung aktivieren. Zumindest möchten Sie wahrscheinlich die `Static Checking` aktivieren (Überprüfung nach dem Build). Wenn Sie eine Bibliothek implementieren, die von anderen Lösungen verwendet wird, möchten Sie möglicherweise auch die `Runtime Checking` aktivieren.




Code-Verträge online lesen: https://riptutorial.com/de/dot-net/topic/1937/code-vertrage

# Kapitel 11: Datei Eingabe / Ausgabe

## Parameter

| Parameter | Einzelheiten |
|---|---|
| String-Pfad | Pfad der zu überprüfenden Datei. (relativ oder voll qualifiziert) |

## Bemerkungen

Gibt true zurück, wenn die Datei vorhanden ist, andernfalls false.

## Examples

### VB WriteAllText

```
Imports System.IO

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

### VB StreamWriter

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

### C # StreamWriter

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

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

### C # WriteAllText ()

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

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

## C # File.Exists ()

```
using System;
using System.IO;

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

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

Kann auch in einem ternären Operator verwendet werden.

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

Datei Eingabe / Ausgabe online lesen: https://riptutorial.com/de/dot-net/topic/1376/datei-eingabe---ausgabe

# Kapitel 12: DateTime-Analyse

## Examples

### ParseExact

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

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

24.11.2015 12:00:00 Uhr

Beachten Sie, dass die Übergabe von `CultureInfo.CurrentCulture` als dritter Parameter mit der Übergabe von `null` identisch ist. Oder Sie können eine bestimmte Kultur übergeben.

**Zeichenketten formatieren**

*Die Eingabezeichenfolge kann ein beliebiges Format haben, das mit der Formatzeichenfolge übereinstimmt*

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

24.11.2015 12:00:00 Uhr

*Alle Zeichen, die keine Formatbezeichner sind, werden als Literale behandelt*

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

24.11.2015 12:00:00 Uhr

*Der Fall ist für Formatbezeichner wichtig*

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

24.11.2015 11:01:30 Uhr

Beachten Sie, dass die Monats- und Minutenwerte in die falschen Ziele geparst wurden.

*Einzelzeichen-Zeichenfolgen müssen eines der Standardformate sein*

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

### Ausnahmen

*ArgumentNullException*

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

*FormatException*

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

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

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

### Umgang mit mehreren möglichen Formaten

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

### Umgang mit kulturellen Unterschieden

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

Tag: 11; Monat: 10

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

Tag: 10; Monat: 11

## TryParse

Diese Methode akzeptiert eine Zeichenfolge als Eingabe, versucht sie in eine `DateTime` zu parsen und gibt ein boolesches Ergebnis zurück, das den Erfolg oder Misserfolg angibt. Wenn der Aufruf erfolgreich ist, wird die als `out` Parameter übergebene Variable mit dem geparsten Ergebnis aufgefüllt.

Wenn die Analyse fehlschlägt, wird die als `out` Parameter übergebene Variable auf den Standardwert `DateTime.MinValue` .

### TryParse (String, out DateTime)

```
DateTime parsedValue;

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

Diese Methode versucht, die Eingabezeichenfolge basierend auf den regionalen Systemeinstellungen und bekannten Formaten wie ISO 8601 und anderen gängigen Formaten zu analysieren.

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

Da diese Methode keine Kulturinformationen akzeptiert, verwendet sie das Systemgebietsschema. Dies kann zu unerwarteten Ergebnissen führen.

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

Falsch

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

Falsch

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

Wahr

Wenn Sie sich in den USA aufhalten, werden Sie vielleicht überrascht sein, dass das geparste Ergebnis der 10. November und nicht der 11. Oktober ist.

### TryParse (String, IFormatProvider, DateTimeStyles, outTime)

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

Im Gegensatz zu seiner Geschwistermethode können bei dieser Überladung eine bestimmte

Kultur und ein bestimmter Stil angegeben werden. Das Übergeben von `null` für den Parameter `IFormatProvider` verwendet die `IFormatProvider` .

*Ausnahmen*

Beachten Sie, dass diese Methode unter bestimmten Bedingungen eine Ausnahme auslösen kann. Diese beziehen sich auf die für diese Überladung eingeführten Parameter: `IFormatProvider` und `DateTimeStyles` .

- `NotSupportedException` : `IFormatProvider` gibt eine neutrale Kultur an
- `ArgumentException` : `DateTimeStyles` ist keine gültige Option oder enthält inkompatible Flags wie `AssumeLocal` und `AssumeUniversal` .

## TryParseExact

Diese Methode verhält sich wie eine Kombination aus `TryParse` und `ParseExact` : Sie ermöglicht die Angabe von benutzerdefinierten Formaten und gibt ein boolesches Ergebnis zurück, das auf Erfolg oder Misserfolg hinweist, und gibt keine Ausnahme aus, wenn die Analyse fehlschlägt.

**TryParseExact (Zeichenfolge, Zeichenfolge, IFormatProvider, DateTimeStyles, outTime)**

Diese Überladung versucht, die Eingabezeichenfolge anhand eines bestimmten Formats zu analysieren. Die Eingabezeichenfolge muss diesem Format entsprechen, damit sie analysiert werden kann.

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

**TryParseExact (string, string [], IFormatProvider, DateTimeStyles, outTime)**

Diese Überladung versucht, die Eingabezeichenfolge mit einem Array von Formaten zu analysieren. Die Eingabezeichenfolge muss mindestens einem Format entsprechen, damit analysiert werden kann.

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

DateTime-Analyse online lesen: https://riptutorial.com/de/dot-net/topic/58/datetime-analyse

# Kapitel 13: die Einstellungen

## Examples

### AppSettings aus ConfigurationSettings in .NET 1.x

### Veraltete Nutzung

Die ConfigurationSettings- Klasse war der ursprüngliche Weg, um Einstellungen für eine Assembly in .NET 1.0 und 1.1 abzurufen. Sie wurde durch die ConfigurationManager- Klasse und die WebConfigurationManager- Klasse ersetzt.

Wenn Sie im Abschnitt `appSettings` der Konfigurationsdatei zwei Schlüssel mit demselben Namen haben, wird der letzte verwendet.

**app.config**

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

**Program.cs**

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

### AppSettings aus ConfigurationManager in .NET 2.0 und höher lesen

Die ConfigurationManager- Klasse unterstützt die `AppSettings` Eigenschaft, mit der Sie das Lesen der Einstellungen aus dem `appSettings` Abschnitt einer Konfigurationsdatei auf dieselbe Weise wie bei .NET 1.x fortsetzen können.

**app.config**

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

**Program.cs**

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

## Einführung in die Unterstützung stark typisierter Anwendungen und Benutzereinstellungen von Visual Studio

Visual Studio hilft bei der Verwaltung von Benutzer- und Anwendungseinstellungen. Die Verwendung dieses Ansatzes hat diese Vorteile gegenüber der Verwendung des `appSettings` Abschnitts der Konfigurationsdatei.

1. Einstellungen können stark getippt werden. Für einen Einstellungswert kann jeder Typ verwendet werden, der serialisiert werden kann.

2. Anwendungseinstellungen können leicht von den Benutzereinstellungen getrennt werden. Anwendungseinstellungen werden in einer einzigen Konfigurationsdatei gespeichert: `web.config` für Websites und Webanwendungen und app.config, umbenannt in *Assembly* .exe.config, wobei *Assembly* der Name der ausführbaren Datei ist. Benutzereinstellungen (die nicht von `user.config` verwendet werden) werden in einer Datei `user.config` im

Anwendungsdatenordner des Benutzers gespeichert (je nach Betriebssystemversion).

3. Anwendungseinstellungen aus Klassenbibliotheken können ohne Risiko von Namenskollisionen in einer einzigen Konfigurationsdatei zusammengefasst werden, da jede Klassenbibliothek über einen eigenen Abschnitt für benutzerdefinierte Einstellungen verfügen kann.

In den meisten Projekttypen verfügt der Projekteigenschaften-Designer über eine Registerkarte Einstellungen , auf der sich benutzerdefinierte Anwendungs- und Benutzereinstellungen erstellen lassen. Anfänglich ist die Registerkarte "Einstellungen" leer, mit einem einzigen Link zum Erstellen einer Standardeinstellungsdatei. Wenn Sie auf den Link klicken, werden folgende Änderungen vorgenommen:

1. Wenn für das Projekt keine Konfigurationsdatei ( `app.config` oder `web.config` ) vorhanden ist, wird eine erstellt.

2. Die Registerkarte Einstellungen wird durch ein Raster-Steuerelement ersetzt, mit dem Sie einzelne Einstellungen erstellen, bearbeiten und löschen können.

3. Im Projektmappen-Explorer wird unter dem Sonderordner Eigenschaften ein Element `Settings.settings` hinzugefügt. Durch Öffnen dieses Elements wird die Registerkarte Einstellungen geöffnet.

4. Eine neue Datei mit einer neuen Teilklasse wird unter dem Ordner `Properties` im Projektordner hinzugefügt. Diese neue Datei heißt `Settings.Designer.__` (.cs, .vb usw.) und die Klasse heißt `Settings` . Die Klasse wird durch einen Code generiert. Sie sollte also nicht bearbeitet werden. Die Klasse ist jedoch eine Teilklasse. Sie können die Klasse erweitern, indem Sie zusätzliche Elemente in eine separate Datei einfügen. Darüber hinaus wird die Klasse mithilfe des Singleton-Musters implementiert, wodurch die Singleton-Instanz mit der Eigenschaft `Default` .

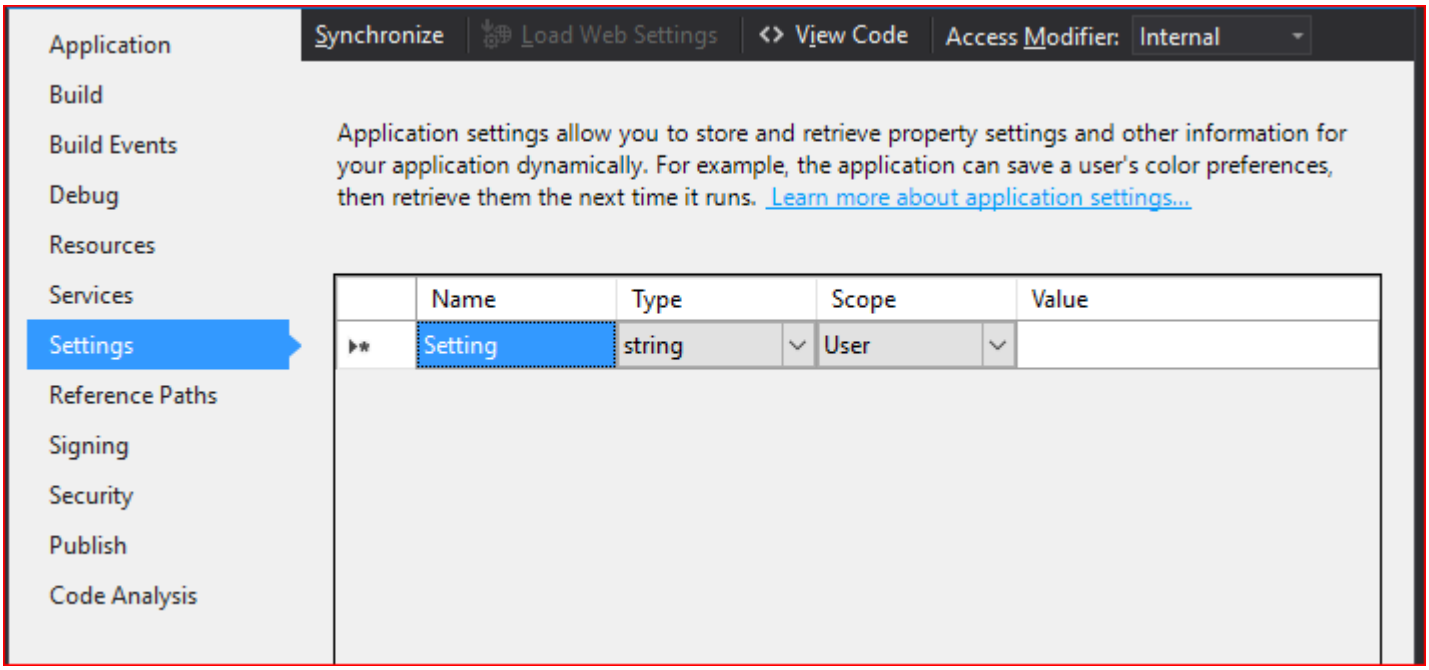
Wenn Sie jeden neuen Eintrag zur Registerkarte "Einstellungen" hinzufügen, führt Visual Studio die folgenden zwei Aktionen aus:

1. Speichert die Einstellung in der Konfigurationsdatei in einem benutzerdefinierten Konfigurationsabschnitt, der von der Settings-Klasse verwaltet werden soll.

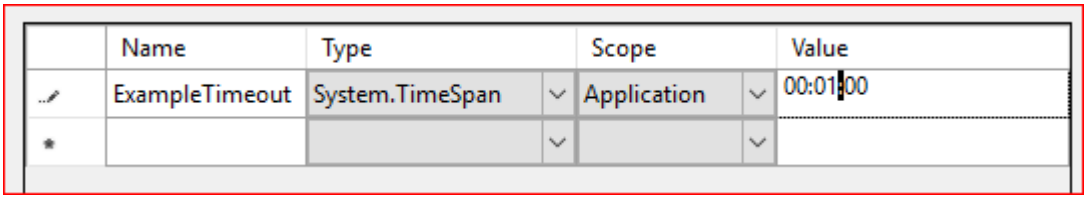
2. Erstellt ein neues Mitglied in der Settings-Klasse, um die Einstellung in dem bestimmten Typ zu lesen, zu schreiben und darzustellen, der auf der Registerkarte Settings ausgewählt ist.

## Lesen stark typisierter Einstellungen aus dem benutzerdefinierten Abschnitt der Konfigurationsdatei

Beginnen Sie mit einem neuen Einstellungsbereich und einer benutzerdefinierten Konfiguration:




Fügen Sie mit der Zeit System.Timespan eine Anwendungseinstellung mit dem Namen ExampleTimeout hinzu, und legen Sie den Wert auf 1 Minute fest:




Speichern Sie die Projekteigenschaften, wodurch die Einträge auf der Registerkarte Einstellungen gespeichert werden, die benutzerdefinierte Einstellungsklasse neu generiert und die Projektkonfigurationsdatei aktualisiert wird.

Verwenden Sie die Einstellung aus Code (C #):

**Program.cs**

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

## Unter der Decke

Schauen Sie in der Projektkonfigurationsdatei nach, wie der Eintrag für die Anwendungseinstellungen erstellt wurde:

**app.config** (Visual Studio aktualisiert dies automatisch)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Beachten Sie, dass der Abschnitt `appSettings` nicht verwendet wird. Der Abschnitt `applicationSettings` enthält einen benutzerdefinierten, für einen Namespace qualifizierten Abschnitt, der für jeden Eintrag ein `setting` enthält. Der Typ des Werts wird nicht in der Konfigurationsdatei gespeichert. Es ist nur der `Settings` Klasse bekannt.

In der `Settings` Klasse `Settings` , wie diese `ConfigurationManager` Klasse zum Lesen dieses benutzerdefinierten Abschnitts verwendet wird.

**Settings.designer.cs** (für C # -Projekte)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
        }
    }
...
```

Beachten Sie, dass ein `DefaultSettingValueAttribute` erstellt wurde, um den auf der Registerkarte Einstellungen des Projekteigenschaften-Designers eingegebenen Wert zu speichern. Wenn der Eintrag in der Konfigurationsdatei fehlt, wird stattdessen dieser Standardwert verwendet.

die Einstellungen online lesen: https://riptutorial.com/de/dot-net/topic/54/die-einstellungen

# Kapitel 14: Einfädeln

## Examples

### Zugriff auf Formularsteuerelemente von anderen Threads

Wenn Sie ein Attribut eines Steuerelements (z. B. ein Textfeld oder eine Beschriftung) aus einem anderen Thread als dem GUI-Thread, mit dem das Steuerelement erstellt wurde, ändern möchten, müssen Sie es aufrufen. Andernfalls erhalten Sie möglicherweise eine Fehlermeldung, die Folgendes angibt:

> "Cross-Thread-Vorgang nicht gültig: Auf Steuerelement 'Steuerelementname' wurde von einem anderen Thread als dem Thread zugegriffen, in dem er erstellt wurde."

Die Verwendung dieses Beispielcodes in einem system.windows.forms-Formular führt zu einer Ausnahme mit dieser Nachricht:

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

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

Wenn Sie den Text eines Textfelds innerhalb eines Threads ändern möchten, der nicht Eigentümer ist, verwenden Sie Control.Invoke oder Control.BeginInvoke. Sie können auch Control.InvokeRequired verwenden, um zu prüfen, ob das Aufrufen des Steuerelements erforderlich ist.

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

Wenn Sie dies häufig tun müssen, können Sie eine Erweiterung für aufrufbare Objekte schreiben, um den für diese Prüfung erforderlichen Code zu reduzieren:

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

```
        else
            action();
    }
}
```

Das Aktualisieren des Textfelds von einem beliebigen Thread wird ein bisschen einfacher:

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

Beachten Sie, dass Control.BeginInvoke, wie in diesem Beispiel verwendet, asynchron ist. Dies bedeutet, dass Code, der nach einem Aufruf von Control.BeginInvoke kommt, sofort ausgeführt werden kann, nachdem der übergebene Delegat bereits ausgeführt wurde.

Wenn Sie sicherstellen möchten, dass textBox1 aktualisiert wird, bevor Sie fortfahren, verwenden Sie stattdessen Control.Invoke. Dadurch wird der aufrufende Thread so lange blockiert, bis Ihr Delegat ausgeführt wurde. Beachten Sie, dass dieser Ansatz Ihren Code erheblich verlangsamen kann, wenn Sie viele Aufrufe aufrufen und beachten, dass Ihre Anwendung dadurch blockiert wird, wenn Ihr GUI-Thread darauf wartet, dass der aufrufende Thread eine angehaltene Ressource beendet oder freigibt.

Einfädeln online lesen: https://riptutorial.com/de/dot-net/topic/3098/einfadeln

# Kapitel 15: Fortschritt verwenden und IProgress

## Examples

### Einfache Fortschrittsberichterstattung

`IProgress<T>` kann verwendet werden, um den Fortschritt eines Verfahrens an ein anderes Verfahren zu melden. Dieses Beispiel zeigt, wie Sie eine grundlegende Methode erstellen, die den Fortschritt anzeigt.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Ausgabe:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Wenn Sie diesen Code ausführen, werden möglicherweise Zahlen außerhalb der Reihenfolge ausgegeben. Dies liegt daran, dass die `IProgress<T>.Report()` -Methode asynchron ausgeführt wird und daher nicht für Situationen geeignet ist, in denen der Fortschritt in Reihenfolge gemeldet werden muss.

### Verwendung von IProgress

Beachten Sie, dass für die Klasse `System.Progress<T>` nicht die Methode `Report()` verfügbar ist.

Diese Methode wurde explizit von der `IProgress<T>` -Schnittstelle implementiert und muss daher für einen `Progress<T>` aufgerufen werden, wenn sie in einen `IProgress<T>` .

```
var p1 = new Progress<int>();
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works

var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

Fortschritt verwenden und IProgress online lesen: https://riptutorial.com/de/dot-net/topic/5628/fortschritt-verwenden--t--und-iprogress--t-

# Kapitel 16: Für jeden

## Bemerkungen

### Verwenden Sie es überhaupt?

Sie könnten argumentieren, dass die Absicht des .NET-Frameworks ist, dass Abfragen keine Nebenwirkungen haben und die `ForEach` Methode definitionsgemäß einen Nebeneffekt verursacht. Wenn Sie stattdessen ein einfaches `foreach` verwenden, ist der Code möglicherweise wartungsfreundlicher und leichter zu testen.

## Examples

### Aufrufen einer Methode für ein Objekt in einer Liste

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

### Erweiterungsmethode für IEnumerable

`ForEach()` ist in der `List<T>` -Klasse definiert, jedoch nicht in `IQueryable<T>` oder `IEnumerable<T>` . In diesen Fällen haben Sie zwei Möglichkeiten:

#### ToList zuerst

Die Aufzählung (oder Abfrage) wird ausgewertet, die Ergebnisse in eine neue Liste kopiert oder die Datenbank aufgerufen. Die Methode wird dann für jedes Element aufgerufen.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

Diese Methode hat einen offensichtlichen Overhead für die Speicherauslastung, da eine Zwischenliste erstellt wird.

#### Erweiterungsmethode

Schreibe eine Erweiterungsmethode:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Benutzen:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Achtung: Die LINQ-Methoden des Frameworks wurden mit der Absicht entwickelt, *rein zu sein* , was bedeutet, dass sie keine Nebenwirkungen erzeugen. Die `ForEach` Methode hat nur den Zweck, Nebenwirkungen zu erzeugen, und weicht in diesem Aspekt von den anderen Methoden ab. Sie können auch nur eine einfache `foreach` Schleife verwenden.

Für jeden online lesen: https://riptutorial.com/de/dot-net/topic/2225/fur-jeden

# Kapitel 17: Globalisierung in ASP.NET MVC mithilfe von Smart Internationalisierung für ASP.NET

## Bemerkungen

Intelligente Internationalisierung für ASP.NET-Seite

Der Vorteil dieses Ansatzes besteht darin, dass Sie Controller und andere Klassen nicht mit Code überladen müssen, um Werte aus RESX-Dateien nachzuschlagen. Sie umgeben den Text einfach in [[[[dreifache Klammern]]]] (Das Trennzeichen ist konfigurierbar.) Ein `HttpModule` sucht in Ihrer .po-Datei nach einer Übersetzung, um den Text mit Trennzeichen zu ersetzen. Wenn eine Übersetzung gefunden wird, ersetzt das `HttpModule` die Übersetzung. Wenn keine Übersetzung gefunden wird, werden die dreifachen Klammern entfernt und die Seite mit dem ursprünglichen nicht übersetzten Text dargestellt.

.po-Dateien sind ein Standardformat für die Bereitstellung von Übersetzungen für Anwendungen. Daher stehen eine Reihe von Anwendungen zur Bearbeitung zur Verfügung. Es ist einfach, eine .po-Datei an einen nichttechnischen Benutzer zu senden, damit er Übersetzungen hinzufügen kann.

## Examples

### Grundkonfiguration und Setup

1. Fügen Sie das I18N-Nuget-Paket Ihrem MVC-Projekt hinzu.
2. Fügen Sie in web.config das `i18n.LocalizingModule` zum Abschnitt `<httpModules>` oder `<modules>` .

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Fügen Sie im Stammverzeichnis Ihrer Site einen Ordner mit dem Namen "locale" hinzu. Erstellen Sie für jede Kultur, die Sie unterstützen möchten, einen Unterordner. Zum Beispiel `/locale/fr/` .
4. Erstellen Sie in jedem kulturspezifischen Ordner eine Textdatei mit dem Namen `messages.po` .

5. Geben Sie zu Testzwecken die folgenden Textzeilen in Ihre Datei `messages.po` :

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. Fügen Sie Ihrem Projekt einen Controller hinzu, der den zu übersetzenden Text zurückgibt.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Führen Sie Ihre MVC-Anwendung aus, und navigieren Sie zu der Route, die Ihrer Controller-Aktion entspricht, z. B. http: // localhost: [Ihre Portnummer] / default .
   Beachten Sie, dass die URL entsprechend Ihrer Standardkultur geändert wird, z. B. http: // localhost: [IhrePortnummer] / de / default .
8. Ersetzen Sie `/en/` in der URL durch `/fr/` (oder die von Ihnen ausgewählte Kultur.) Die Seite sollte jetzt die übersetzte Version Ihres Textes anzeigen.
9. Ändern Sie die Spracheinstellung Ihres Browsers, um Ihre alternative Kultur zu bevorzugen, und navigieren Sie erneut zu `/default` . Beachten Sie, dass die URL entsprechend Ihrer alternativen Kultur geändert wird und der übersetzte Text angezeigt wird.
10. Fügen Sie in web.config Handler hinzu, damit Benutzer nicht zu Ihrem `locale` navigieren können.

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

Globalisierung in ASP.NET MVC mithilfe von Smart Internationalisierung für ASP.NET online lesen: https://riptutorial.com/de/dot-net/topic/5086/globalisierung-in-asp-net-mvc-mithilfe-von-

smart-internationalisierung-fur-asp-net

# Kapitel 18: HTTP-Clients

## Bemerkungen

Die aktuell relevanten HTTP / 1.1-RFCs sind:

- 7230: Nachrichtensyntax und Routing
- 7231: Semantik und Inhalt
- 7232: Bedingte Anfragen
- 7233: Bereichsanfragen
- 7234: Zwischenspeicherung
- 7235: Authentifizierung
- 7239: Weitergeleitete HTTP-Erweiterung
- 7240: Header für HTTP vorziehen

Es gibt auch die folgenden Informations-RFCs:

- 7236: Registrierungen des Authentifizierungsschemas
- 7237: Methodenregistrierungen

Und der experimentelle RFC:

- 7238: Der Hypertext Transfer Protocol-Statuscode 308 (Permanente Umleitung)

Verwandte Protokolle:

- 4918: HTTP-Erweiterungen für Web Distributed Authoring und Versioning (WebDAV)
- 4791: Kalendererweiterungen für WebDAV (CalDAV)

## Examples

### Lesen der GET-Antwort als Zeichenfolge mit System.Net.HttpWebRequest

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

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

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

### GET-Antwort wird als Zeichenfolge mit System.Net.WebClient gelesen

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

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

## GET-Antwort wird als Zeichenfolge mit System.Net.HttpClient gelesen

`HttpClient` ist über NuGet verfügbar : Microsoft HTTP-Clientbibliotheken .

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

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

## Senden einer POST-Anforderung mit einer String-Nutzlast mithilfe von System.Net.HttpWebRequest

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

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

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

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

## Senden einer POST-Anforderung mit einer String-Nutzlast mithilfe von System.Net.WebClient

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

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

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

## Senden einer POST-Anforderung mit einer String-Nutzlast mithilfe von System.Net.HttpClient

`HttpClient` ist über NuGet verfügbar : Microsoft HTTP-Clientbibliotheken .

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

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

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

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

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

## Grundlegender HTTP-Downloader mit System.Net.Http.HttpClient

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

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

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

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

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

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

HTTP-Clients online lesen: https://riptutorial.com/de/dot-net/topic/32/http-clients

# Kapitel 19: HTTP-Server

## Examples

### Grundlegender schreibgeschützter HTTP-Dateiserver (HttpListener)

**Anmerkungen:**

Dieses Beispiel muss im Verwaltungsmodus ausgeführt werden.

Es wird nur ein gleichzeitiger Client unterstützt.

Der Einfachheit halber wird angenommen, dass Dateinamen ausschließlich aus ASCII - Dateien bestehen (für den *Dateinamen*- Teil des *Content-Disposition*- Headers), und Dateizugriffsfehler werden nicht behandelt.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);

        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
```

```
            ServerLoop();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }

    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
```

```
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
    }
}
```

## Grundlegender schreibgeschützter HTTP-Dateiserver (ASP.NET Core)

1 - Erstellen Sie einen leeren Ordner, der die in den nächsten Schritten erstellten Dateien enthält.

2 - Erstellen Sie eine Datei mit dem Namen `project.json` mit dem folgenden Inhalt (passen Sie die Portnummer und `rootDirectory` entsprechend an):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Erstellen Sie eine Datei mit dem Namen `Startup.cs` mit dem folgenden Code:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
```

```
            context.Context.Response.Headers.Add(
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };

        app.UseStaticFiles(options);
    }
}
```

4 - Öffnen Sie eine Eingabeaufforderung, navigieren Sie zu dem Ordner und führen Sie Folgendes aus:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

**Hinweis:** Diese Befehle müssen nur einmal ausgeführt werden. Verwenden Sie `dnvm list` , um die tatsächliche Nummer der zuletzt installierten Version der Core-CLR zu überprüfen.

5 - Starten Sie den Server mit: `dnx web` . Dateien können jetzt unter `http://localhost:60000/path/to/file.ext` .

Der Einfachheit halber wird angenommen, dass Dateinamen ausschließlich aus ASCII-Dateien bestehen (für den Dateinamen-Teil des Content-Disposition-Headers), und Dateizugriffsfehler werden nicht behandelt.

HTTP-Server online lesen: https://riptutorial.com/de/dot-net/topic/53/http-server

# Kapitel 20: JIT-Compiler

## Einführung

Die JIT-Kompilierung oder Just-in-Time-Kompilierung ist ein alternativer Ansatz zur Interpretation von Code oder zur Vorausberechnung. Die JIT-Kompilierung wird im .NET-Framework verwendet. Der CLR-Code (C #, F #, Visual Basic usw.) wird zuerst in etwas übersetzt, das als Interpretierte Sprache (IL) bezeichnet wird. Dies ist ein Code auf niedrigerer Ebene, der näher an Maschinencode liegt, jedoch nicht plattformspezifisch ist. Dieser Code wird zur Laufzeit vielmehr in Maschinencode für das betreffende System zusammengefasst.

## Bemerkungen

Warum JIT-Kompilierung verwenden?

- Bessere Kompatibilität: Jede CLR-Sprache benötigt nur einen Compiler für IL, und diese IL kann auf jeder Plattform ausgeführt werden, auf der sie in Maschinencode konvertiert werden kann.
- Geschwindigkeit: Bei der JIT-Kompilierung wird versucht, die Geschwindigkeit, mit der der kompilierte Code vor der Ausführung ausgeführt wird, und die Flexibilität der Interpretation zu kombinieren.

Wikipedia-Seite für weitere Informationen zur JIT-Kompilierung im Allgemeinen: https://en.wikipedia.org/wiki/Just-in-time_compilation

## Examples

### IL Zusammenstellungsbeispiel

Simple Hello World-Anwendung:

```
using System;

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

Äquivalenter IL-Code (der von JIT erstellt wird)

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

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



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

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

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


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

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

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

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

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

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

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

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

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

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


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

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

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

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

Generiert mit MS ILDASM-Tool (IL-Disassembler)

JIT-Compiler online lesen: https://riptutorial.com/de/dot-net/topic/9222/jit-compiler

# Kapitel 21: JSON in .NET mit Newtonsoft.Json

## Einführung

Das NuGet-Paket `Newtonsoft.Json` wurde zum Defacto-Standard für die Verwendung und Bearbeitung von JSON-formatiertem Text und Objekten in .NET. Es ist ein robustes Werkzeug, das schnell und einfach zu bedienen ist.

## Examples

### Objekt in JSON serialisieren

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

Dies führt zu diesem JSON: `{"Name":"Joe Smith","Age":21}`

### Deserialisieren Sie ein Objekt aus JSON-Text

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

Dies ergibt ein `Person` Objekt mit dem Namen "Joe Smith" und dem Alter von 21.

JSON in .NET mit Newtonsoft.Json online lesen: https://riptutorial.com/de/dot-net/topic/8746/json-in--net-mit-newtonsoft-json

# Kapitel 22: JSON-Serialisierung

## Bemerkungen

### JavaScriptSerializer vs Json.NET

Die `JavaScriptSerializer` Klasse wurde in .NET 3.5 eingeführt und wird intern von der asynchronen Kommunikationsschicht von .NET für AJAX-fähige Anwendungen verwendet. Es kann verwendet werden, um mit JSON in verwaltetem Code zu arbeiten.

Trotz der Existenz der `JavaScriptSerializer` Klasse empfiehlt Microsoft die Verwendung der Open Source- Bibliothek Json.NET für die Serialisierung und Deserialisierung. Json.NET bietet eine bessere Leistung und eine freundlichere Schnittstelle für die Zuordnung von JSON zu benutzerdefinierten Klassen (ein benutzerdefiniertes `JavaScriptConverter` Objekt wäre erforderlich, um dies mit `JavaScriptSerializer` ).

## Examples

### Deserialisierung mit System.Web.Script.Serialization.JavaScriptSerializer

Die `JavaScriptSerializer.Deserialize<T>(input)` -Methode `JavaScriptSerializer.Deserialize<T>(input)` versucht, eine Zeichenfolge gültiger JSON in ein Objekt des angegebenen Typs `<T>` zu deserialisieren, wobei die Standardzuordnungen verwendet werden, die von `JavaScriptSerializer` standardmäßig unterstützt werden.

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Hinweis: Das `JavaScriptSerializer` Objekt wurde in .NET Version 3.5 eingeführt

### Deserialisierung mit Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...
```

```
string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

Weitere Informationen finden Sie auf der offiziellen Website von Json.NET .

Hinweis: Json.NET unterstützt .NET Version 2 und höher.

## Serialisierung mit Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address"
};
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Beachten Sie, wie Eigenschaften (und Klassen) mit Attributen versehen werden können, um ihr Erscheinungsbild in der resultierenden Json-Zeichenfolge zu ändern oder sie überhaupt aus der Json-Zeichenfolge zu entfernen (JsonIgnore).

Weitere Informationen zu Json.NET-Serialisierungsattributen finden Sie hier .

In C # werden öffentliche Bezeichner nach Konventionen in *PascalCase* geschrieben. In JSON ist es üblich, *camelCase* für alle Namen zu verwenden. Sie können einen Vertragsauflöser verwenden, um zwischen den beiden zu konvertieren.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

    Console.WriteLine(json); // {"name":"Andrius","age":99}
```

```
}
```

## Serialisierung-Deserialisierung mit Newtonsoft.Json

Im Gegensatz zu den anderen Helfern verwendet dieser Assistent statische Klassenhelfer zum Serialisieren und Deserialisieren. Daher ist es etwas einfacher als die anderen.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

## Dynamische Bindung

Mit Json.NET von Newtonsoft können Sie Json dynamisch binden (mithilfe von ExpandoObject / Dynamic-Objekten), ohne dass Sie den Typ explizit erstellen müssen.

### Serialisierung

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

### De-Serialisierung

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Beachten Sie, dass die Schlüssel im rawJson-Objekt im dynamischen Objekt in Member-Variablen umgewandelt wurden.

Dies ist in Fällen nützlich, in denen eine Anwendung unterschiedliche JSON-Formate akzeptieren / produzieren kann. Es wird jedoch empfohlen, eine zusätzliche Validierungsstufe für den Json-String oder für das als Ergebnis der Serialisierung / Deserialisierung generierte dynamische Objekt zu verwenden.

## Serialisierung mit Json.NET mit JsonSerializerSettings

Dieser Serialisierer verfügt über einige nette Funktionen, die der standardmäßige .net json-Serialisierer nicht bietet. Wie bei der Nullwertverarbeitung müssen Sie lediglich die `JsonSerializerSettings` erstellen:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore});
```

```
   return result;
}
```

Ein weiteres ernstes Problem des Serializers in .net ist die selbstreferenzierende Schleife. Im Falle eines Studenten, der an einem Kurs angemeldet ist, verfügt seine Instanz über eine Kurseigenschaft und ein Kurs enthält eine Sammlung von Studenten, dh eine `List<Student>` die eine Referenzschleife erstellt. Sie können dies mit `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Sie können verschiedene Serialisierungsoptionen wie folgt setzen:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore, ReferenceLoopHandling =
ReferenceLoopHandling.Ignore});
   return result;
}
```

JSON-Serialisierung online lesen: https://riptutorial.com/de/dot-net/topic/183/json-serialisierung

# Kapitel 23: Laden Sie Datei- und POST-Daten auf den Webserver hoch

## Examples

### Datei mit WebRequest hochladen

Um eine Datei und Formulardaten in einer einzelnen Anforderung zu senden, sollte der Inhalt einen Multipart / Form-Datentyp haben .

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Verwendungszweck:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
    });
```

Laden Sie Datei- und POST-Daten auf den Webserver hoch online lesen:
https://riptutorial.com/de/dot-net/topic/10845/laden-sie-datei--und-post-daten-auf-den-webserver-hoch

# Kapitel 24: LINQ

## Einführung

LINQ (Language Integrated Query) ist ein Ausdruck, der Daten aus einer Datenquelle abruft. LINQ vereinfacht diese Situation, indem es ein konsistentes Modell für die Arbeit mit Daten über verschiedene Arten von Datenquellen und -formaten anbietet. In einer LINQ-Abfrage arbeiten Sie immer mit Objekten. Sie verwenden dieselben grundlegenden Codierungsmuster zum Abfragen und Umwandeln von Daten in XML-Dokumenten, SQL-Datenbanken, ADO.NET-Datensätzen, .NET-Sammlungen und anderen Formaten, für die ein Anbieter verfügbar ist. LINQ kann in C # und VB verwendet werden.

## Syntax

- public static TSource Aggregate <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, TSource, TSource> func)
- public static TAccumulate Aggregate <TSource, TAccumulate> (diese IEnumerable <TSource> -Quelle, TAccumulate-Samen, Func <TAccumulate, TSource, TAccumulate> -Funk)
- public static TResult Aggregate <TSource, TAccumulate, TResult> (diese IEnumerable <TSource> -Quelle, TAccumulate-Samen, Func <TAccumulate, TSource, TAccumulate> func, Func <TAccumulate, TResult> resultSelector)
- public static Boolean All <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static Boolean Any <TSource> (diese IEnumerable-Quelle <TSource>)
- public static Boolean Any <TSource> (diese IEnumerable-Quelle <TSource>, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> AsEnumerable <TSource> (diese IEnumerable-Quelle <TSource>)
- public static decimal Average (diese IEnumerable-Quelle <Decimal>)
- public static Double Average (diese IEnumerable-Quelle <Double>)
- public static Double Average (diese IEnumerable-Quelle <Int32>)
- public static Double Average (diese IEnumerable-Quelle <Int64>)
- public static Nullable <Decimal> Average (diese IEnumerable <Nullable <Decimal >> -Quelle)
- public static Nullable <Double> Average (diese IEnumerable <Nullable <Double >> -Quelle)
- public static Nullable <Double> Average (diese IEnumerable <Nullable <Int32 >> -Quelle)
- public static Nullable <Double> Average (diese IEnumerable <Nullable <Int64 >> -Quelle)
- public static Nullable <Single> Average (diese IEnumerable <Nullable <Single >> -Quelle)
- public static Single Average (diese IEnumerable-Quelle <Single>)
- public static Decimal Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Decimal> -Selektor)
- public static Double Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Double> -Selektor)

- public static Double Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32> -Selektor)
- public static Double Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int64> -Selektor)
- public static Nullable <Decimal> Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Decimal >> -Selektor)
- public static Nullable <Double> Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Double >> -Selektor)
- public static Nullable <Double> Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int32 >> -Selektor)
- public static Nullable <Double> Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int64 >> -Selektor)
- public static Nullable <Single> Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Single >> -selektor)
- public static Single Average <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Single> -Selektor)
- public static IEnumerable <TResult> Cast <TResult> (diese IEnumerable-Quelle)
- public static IEnumerable <TSource> Concat <TSource> (zuerst IEnumerable <TSource>, zuerst IEnumerable <TSource>)
- public static Boolean Enthält <TSource> (diese IEnumerable <TSource> -Quelle, TSource-Wert)
- public static Boolean Enthält <TSource> (diese IEnumerable <TSource> -Quelle, TSource-Wert, IEqualityComparer <TSource> -Vergleicher)
- public static Int32 Count <TSource> (diese IEnumerable-Quelle <TSource>)
- public static Int32 Count <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (diese IEnumerable-Quelle <TSource>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (diese IEnumerable <TSource> -Quelle, TSource defaultValue)
- public static IEnumerable <TSource> Distinct <TSource> (diese IEnumerable-Quelle <TSource>)
- public static IEnumerable <TSource> Distinct <TSource> (diese IEnumerable <TSource> -Quelle, IEqualityComparer <TSource> -Vergleicher)
- public static TSource ElementAt <TSource> (diese IEnumerable <TSource> -Quelle, Int32-Index)
- public static TSource ElementAtOrDefault <TSource> (diese IEnumerable-Quelle <TSource>, Int32-Index)
- public static IEnumerable <TResult> Leer <TResult> ()
- public static IEnumerable <TSource> Außer <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource> Sekunde)
- public static IEnumerable <TSource> Außer <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource>, IEqualityComparer <TSource>)
- public static TSource First <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource First <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)

- öffentliche statische TSource FirstOrDefault <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource FirstOrDefault <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> -Vergleicher)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (diese IEnumerable <TSource> -Quelle, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> -Vergleicher)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector, IEqualityComparer <TKey> Vergleicher)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult.) > resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult.) > resultSelector, IEqualityComparer <TKey> (Vergleicher)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (dieses IEnumerable <TOuter> äußere, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (dieses IEnumerable <TOuter> äußere, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector, IEqualityComparer <TKey> (Vergleicher)
- public static IEnumerable <TSource> Intersect <TSource> (zuerst IEnumerable <TSource>, zuerst IEnumerable <TSource>)
- public static IEnumerable <TSource> Intersect <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource>, IEqualityComparer <TSource>)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (dieses IEnumerable <TOuter> äußere, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, TInner, TResult> resultSelector)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (dieses

IEnumerable <TOuter> äußere, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, TInner, TResult> resultSelector, IEqualityComparer <TKey> (Vergleicher)

- public static TSource Last <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource Last <TSource> (diese IEnumerable-Quelle <TSource>, Func <TSource, Boolean> -Prädikat)
- public static TSource LastOrDefault <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource LastOrDefault <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static Int64 LongCount <TSource> (diese IEnumerable <TSource> -Quelle)
- public static Int64 LongCount <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static decimal Max (diese IEnumerable-Quelle <Decimal>)
- public static Double Max (diese IEnumerable-Quelle <Double>)
- public static Int32 Max (diese IEnumerable-Quelle <Int32>)
- public static Int64 Max (diese IEnumerable-Quelle <Int64>)
- public static Nullable <Decimal> Max (diese IEnumerable <Nullable <Decimal >> -Quelle)
- public static Nullable <Double> Max (diese IEnumerable <Nullable <Double >> -Quelle)
- public static Nullable <Int32> Max (diese IEnumerable <Nullable <Int32 >> -Quelle)
- public static Nullable <Int64> Max (diese IEnumerable <Nullable <Int64 >> -Quelle)
- public static Nullable <Single> Max (diese IEnumerable <Nullable <Single >> -Quelle)
- public static Single Max (diese IEnumerable-Quelle <Single>)
- public static TSource Max <TSource> (diese IEnumerable-Quelle <TSource>)
- public static Decimal Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Decimal> -Selektor)
- public static Double Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Double> -Selektor)
- public static Int32 Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32> -Selektor)
- public static Int64 Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int64> -Selektor)
- public static Nullable <Decimal> Max. <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Decimal >> -Selektor)
- public static Nullable <Double> Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Double >> -Selektor)
- public static Nullable <Int32> Max. <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int32 >> -Selektor)
- public static Nullable <Int64> Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int64 >> -Selektor)
- public static Nullable <Single> Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Single >> -selektor)
- public static Single Max <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Single> -Selektor)
- public static TResult Max <TSource, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, TResult> -Selektor)
- public static decimal Min (diese IEnumerable-Quelle <Decimal>)

- public static Double Min (diese IEnumerable-Quelle <Double>)
- public static Int32 Min (diese IEnumerable-Quelle <Int32>)
- public static Int64 Min (diese IEnumerable-Quelle <Int64>)
- public static Nullable <Decimal> Min (diese IEnumerable <Nullable <Decimal >> -Quelle)
- public static Nullable <Double> Min (diese IEnumerable <Nullable <Double >> -Quelle)
- public static Nullable <Int32> Min (diese IEnumerable <Nullable <Int32 >> -Quelle)
- public static Nullable <Int64> Min (diese IEnumerable <Nullable <Int64 >> -Quelle)
- public static Nullable <Single> Min (diese IEnumerable <Nullable <Single >> -Quelle)
- public static Single Min (diese IEnumerable-Quelle <Single>)
- öffentliche statische TSource Min <TSource> (diese IEnumerable-Quelle <TSource>)
- public static Decimal Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Decimal> -Selektor)
- public static Double Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Double> -Selektor)
- public static Int32 Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32> -Selektor)
- public static Int64 Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int64> -Selektor)
- public static Nullable <Decimal> Min. <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Decimal >> -Selektor)
- public static Nullable <Double> Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Double >> -Selektor)
- public static Nullable <Int32> Min. <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int32 >> -Selektor)
- public static Nullable <Int64> Min. <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int64 >> -Selektor)
- public static Nullable <Single> Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Single >> -selektor)
- public static Single Min <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Single> -Selektor)
- public static TResult Min <TSource, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, TResult> -Selektor)
- public static IEnumerable <TResult> OfType <TResult> (diese IEnumerable-Quelle)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> -Vergleicher)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> -Vergleicher)
- public static IEnumerable <Int32> Bereich (Int32-Start, Int32-Zählung)
- public static IEnumerable <TResult> Repeat <TResult> (TResult-Element, Int32-Zählung)
- public static IEnumerable <TSource> Reverse <TSource> (diese IEnumerable-Quelle

<TSource>)
- public static IEnumerable <TResult> Select <TSource, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, TResult> -Selektor)
- public static IEnumerable <TResult> Wählen Sie <TSource, TResult> aus (diese IEnumerable-Quelle <TSource>, Func <TSource, Int32, TResult> -Selektor).
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, IEnumerable <TResult >> -Selektor)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32, IEnumerable <TResult >> -Selektor)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static Boolean SequenceEqual <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource> Sekunde)
- public static Boolean SequenceEqual <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource>, IEqualityComparer <TSource>)
- public static TSource Single <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource Single <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static TSource SingleOrDefault <TSource> (diese IEnumerable-Quelle <TSource>)
- public static TSource SingleOrDefault <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> Überspringen <TSource> (diese IEnumerable <TSource> -Quelle, Int32-Zähler)
- public static IEnumerable <TSource> SkipWhile <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> SkipWhile <TSource> (diese IEnumerable-Quelle <TSource>, Func <TSource, Int32, Boolean>)
- öffentliche statische Dezimalsumme (diese IEnumerable-Quelle <Decimal>)
- öffentliche statische Doppelsumme (diese IEnumerable-Quelle <Double>)
- public static Int32 Sum (diese IEnumerable-Quelle <Int32>)
- public static Int64 Sum (diese IEnumerable-Quelle <Int64>)
- public static Nullable <Decimal> Summe (diese IEnumerable <Nullable <Decimal >> -Quelle)
- public static Nullable <Double> Summe (diese IEnumerable <Nullable <Double >> -Quelle)
- public static Nullable <Int32> Summe (diese IEnumerable <Nullable <Int32 >> -Quelle)
- public static Nullable <Int64> Summe (diese IEnumerable <Nullable <Int64 >> -Quelle)
- public static Nullable <Single> Summe (diese IEnumerable <Nullable <Single >> -Quelle)
- public static Single Sum (diese IEnumerable-Quelle <Single>)
- public static Decimal Sum <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Decimal> -Selektor)
- public static Double Sum <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Double> -Selektor)
- public static Int32 Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func

<TSource, Int32> -Selektor)
- public static Int64 Sum <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int64> -Selektor)
- public static Nullable <Decimal> Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Decimal >> -Selektor)
- public static Nullable <Double> Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Double >> -Selektor)
- public static Nullable <Int32> Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int32 >> -Selektor)
- public static Nullable <Int64> Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Int64 >> -Selektor)
- public static Nullable <Single> Summe <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Nullable <Single >> -Selektor)
- public static Single Sum <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Single> -Selektor)
- public static IEnumerable <TSource> Nehmen Sie <TSource> (diese IEnumerable <TSource> -Quelle, Int32-Zählung)
- public static IEnumerable <TSource> TakeWhile <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> TakeWhile <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Int32, Boolean> -Prädikat)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (diese IOrderedEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (diese IOrderedEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> -Vergleicher)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (diese IOrderedEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (diese IOrderedEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> -Vergleicher)
- public static TSource [] ToArray <TSource> (diese IEnumerable-Quelle <TSource>)
- public static Dictionary <TKey, TSource> ToDictionary <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static Dictionary <TKey, TSource> ToDictionary <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> -Vergleicher)
- public static Dictionary <TKey, TElement> ToDictionary <TSource, TKey, TElement> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static Dictionary <TKey, TElement> ToDictionary <TSource, TKey, TElement> (diese IEnumerable <TSource> -Quelle, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> -Vergleicher)
- öffentliche statische Liste <TSource> ToList <TSource> (diese IEnumerable-Quelle <TSource>)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (diese IEnumerable-

Quelle <TSource>, Func <TSource, TKey> keySelector)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> -Vergleicher)
- public static ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (diese IEnumerable-Quelle <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> -Vergleicher)
- public static IEnumerable <TSource> Union <TSource> (zuerst IEnumerable <TSource>, zuerst IEnumerable <TSource>)
- public static IEnumerable <TSource> Union <TSource> (zuerst IEnumerable <TSource>, IEnumerable <TSource>, IEqualityComparer <TSource>)
- public static IEnumerable <TSource> Where <TSource> (diese IEnumerable <TSource> -Quelle, Func <TSource, Boolean> -Prädikat)
- public static IEnumerable <TSource> Where <TSource> (diese IEnumerable-Quelle <TSource>, Func <TSource, Int32, Boolean>)
- public static IEnumerable <TResult> Zip <TFirst, TSecond, TResult> (zuerst IEnumerable <TFirst>, IEnumerable <TSecond> Zweitens, Func <TFirst, TSecond, TResult> resultSelector)

## Bemerkungen

- Siehe auch LINQ .

Die in LINQ integrierten Methoden sind Erweiterungsmethoden für die `IEnumerable<T>` -Schnittstelle, die in der `System.Linq.Enumerable` Klasse in der `System.Core` Assembly leben. Sie sind in .NET Framework 3.5 und höher verfügbar.

LINQ ermöglicht die einfache Änderung, Transformation und Kombination verschiedener `IEnumerable` mithilfe einer `IEnumerable` oder funktionalen Syntax.

Die Standard-LINQ-Methoden können zwar auf jedem `IEnumerable<T>` , einschließlich der einfachen Arrays und `List<T>` , verwendet werden, sie können jedoch auch auf Datenbankobjekte angewendet werden, bei denen der Satz von LINQ-Ausdrücken in vielen Fällen in SQL umgewandelt werden kann, wenn das Datenobjekt unterstützt es. Siehe LINQ to SQL .

Für die Methoden, die Objekte vergleichen (z. B. `Contains` und `Except` ), wird `IEquatable<T>.Equals` verwendet, wenn der Typ T der Auflistung diese Schnittstelle implementiert. Andernfalls wird die Standard - `Equals` und `GetHashCode` des Typs (möglicherweise von den Standard überschrieben `Object` verwendet werden. Es gibt auch Überladungen für diese Methoden, mit denen ein benutzerdefinierter `IEqualityComparer<T>` .

Bei den `...OrDefault` Methoden wird `default(T)` verwendet, um Standardwerte zu generieren.

Offizielle Referenz: Enumerable-Klasse

# Faule Bewertung

Praktisch jede Abfrage, die ein `IEnumerable<T>` wird nicht sofort ausgewertet. Stattdessen wird die Logik verzögert, bis die Abfrage wiederholt wird. Eine Konsequenz ist, dass jedes Mal, wenn jemand einen `IEnumerable<T> .Where()` , der aus einer dieser Abfragen erstellt wurde, z. B. `.Where()` , die vollständige Abfragelogik wiederholt wird. Wenn das Prädikat lange dauert, kann dies zu Leistungsproblemen führen.

Eine einfache Lösung (wenn Sie die ungefähre Größe der resultierenden Sequenz kennen oder steuern können) ist das vollständige `.ToArray()` der Ergebnisse mithilfe von `.ToArray()` oder `.ToList()` . `.ToDictionary()` oder `.ToLookup()` kann dieselbe Rolle erfüllen. Man kann natürlich auch über die gesamte Sequenz iterieren und die Elemente gemäß anderer benutzerdefinierter Logik puffern.

## `ToArray()` oder `ToList()` ?

Sowohl `.ToArray()` als auch `.ToList()` durchlaufen alle Elemente einer `IEnumerable<T>` -Sequenz und speichern die Ergebnisse in einer im Speicher gespeicherten Auflistung. Verwenden Sie die folgenden Richtlinien, um zu bestimmen, welche Auswahl Sie treffen sollen:

- Einige APIs erfordern möglicherweise ein `T[]` oder eine `List<T>` .
- `.ToList()` läuft normalerweise schneller und erzeugt weniger Abfall als `.ToArray()` , da letztere alle Elemente einmal in eine neue Sammlung fester Größe kopieren muss, in fast allen Fällen.
- Elemente können der von `.ToList() List<T>` hinzugefügt oder daraus entfernt werden, wohingegen das von `.ToList() T[]` während seiner gesamten Lebensdauer eine feste Größe `.ToArray()` . Mit anderen Worten, `List<T>` ist veränderlich und `T[]` ist unveränderlich.
- Die `T[]` zurückgegeben von `.ToArray()` benötigt weniger Speicher als die `List<T>` von zurück `.ToList()` , so dass , wenn das Ergebnis für eine lange Zeit gelagert wird, bevorzugen `.ToArray()` . Durch das Aufrufen der `List<T>.TrimExcess()` der Speicherunterschied streng akademisch, wobei jedoch der relative Geschwindigkeitsvorteil von `.ToList()` .

# Examples

### Auswählen (karte)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

Diese Art von Funktion wird in funktionalen Programmiersprachen normalerweise als `map` .

## Wo (Filter)

Diese Methode gibt ein IEnumerable mit allen Elementen zurück, die den Lambda-Ausdruck erfüllen

**Beispiel**

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

**Ausgabe:**

Foo, Fizz

Demo anzeigen

## Sortieren nach

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

## OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p =>
p.Id).ToArray()));
```

```
//1,3,4,2
```

## Enthält

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

## Außer

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

## Sich schneiden

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

## Concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Beachten Sie, dass Duplikate im Ergebnis bleiben. Wenn dies nicht erwünscht ist, verwenden Sie stattdessen `Union` .

## Erstes (Finden)

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1
```

```
var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

Der folgende `InvalidOperationException` mit der Meldung "Sequenz enthält kein übereinstimmendes Element" aus:

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

## Single

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

Der folgende `InvalidOperationException` da die Sequenz mehrere Elemente enthält:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

## Zuletzt

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

Die folgende `InvalidOperationException` :

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

## LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0
```

```
var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
Console.WriteLine(lastMissingWord); // null
```

## SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

Die folgende `InvalidOperationException` :

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

## FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

## Irgendein

Gibt " `true` wenn die Auflistung Elemente enthält, die die Bedingung im Lambda-Ausdruck erfüllen:

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

## Alles

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Beachten Sie, dass die `All` Methode funktioniert, indem geprüft wird, ob das erste Element gemäß dem Prädikat als `false` bewertet wird. Daher gibt die Methode für *jedes* Prädikat `true` wenn die Menge leer ist:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

## SelectMany (flache Karte)

`Enumerable.Select` gibt für jedes Eingabeelement ein Ausgabeelement zurück. Während `Enumerable.SelectMany` für jedes Eingabeelement eine variable Anzahl von Ausgabeelementen erzeugt. Dies bedeutet, dass die Ausgabefolge möglicherweise mehr oder weniger Elemente enthält als in der Eingabefolge.

An `Enumerable.Select` `Lambda expressions` müssen ein einzelnes Element zurückgeben. An `Enumerable.SelectMany` Lambda-Ausdrücke müssen eine `Enumerable.SelectMany` Sequenz erzeugen. Diese untergeordnete Sequenz kann eine unterschiedliche Anzahl von Elementen für jedes Element in der Eingabesequenz enthalten.

**Beispiel**

```
class Invoice
```

```
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

**Ausgabe:**

1,2,3,4,5,6

Demo anzeigen

`Enumerable.SelectMany` kann auch mit einer auf Syntax basierenden Abfrage erreicht werden, wobei zwei aufeinanderfolgende `from` Klauseln verwendet werden:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

## Summe

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
```

```
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

## Überspringen

Überspringen wird die ersten N Elemente auflisten, ohne sie zurückzugeben. Sobald die Artikelnummer N + 1 erreicht ist, gibt Skip alle aufgelisteten Elemente zurück:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

## Nehmen

Diese Methode entnimmt die ersten `n` Elemente einer Aufzählung.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

## SequenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

## Umkehren

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

## OfType

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

## Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

## Mindest

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

## Durchschnittlich

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

Diese Methode berechnet den Durchschnitt der Anzahl der Zahlen.

```
var cities = new[] {
    new {Population = 1000},
```

```
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

Diese Methode berechnet den Durchschnitt der Aufzählungszeichen mithilfe der delegierten Funktion.

## Postleitzahl

.NET 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

## Eindeutig

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

## Gruppiere nach

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Gruppieren Sie die Rechnungen nach Land und erstellen Sie ein neues Objekt mit der Anzahl der Datensätze, der Gesamtzahl der Zahlungen und dem Durchschnittsgehalt

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Wenn wir nur die Summen wollen, keine Gruppe

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Wenn wir mehrere Zählungen benötigen

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

## ToDictionary

Gibt ein neues Wörterbuch aus dem Quell- `IEnumerable` mit der bereitgestellten keySelector-Funktion zurück, um Schlüssel zu ermitteln. Löst eine `ArgumentException` wenn keySelector nicht injektiv ist (gibt einen eindeutigen Wert für jedes Mitglied der Quellensammlung zurück.) Es gibt Überladungen, mit denen der zu speichernde Wert sowie der Schlüssel angegeben werden können.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Wenn Sie nur eine Schlüsselauswahlfunktion `Dictionary<TKey,TVal>` wird ein `Dictionary<TKey,TVal>` mit `TKey` der Rückgabetyp des Schlüsselwählers, `TVal` der ursprüngliche Objekttyp und das ursprüngliche Objekt als gespeicherter Wert erstellt.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

Wenn Sie auch eine Wertauswahlfunktion `Dictionary<TKey,TVal>` wird ein `Dictionary<TKey,TVal>` wobei `TKey` weiterhin den Rückgabetyp der Schlüsselauswahl, aber `TVal` jetzt den Rückgabetyp der `TVal` und den zurückgegebenen Wert als gespeicherten Wert darstellt.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

Wie bereits erwähnt, müssen die vom Schlüsselwähler zurückgegebenen Schlüssel eindeutig sein. Das folgende wird eine Ausnahme auslösen.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

Wenn für die Quellensammlung kein eindeutiger Schlüssel angegeben werden kann, sollten Sie stattdessen ToLookup verwenden. Auf der Oberfläche verhält sich ToLookup ähnlich wie ToDictionary, in der resultierenden Suche wird jeder Schlüssel mit einer Auflistung von Werten mit übereinstimmenden Schlüsseln gepaart.

## Union

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Beachten Sie, dass Duplikate aus dem Ergebnis entfernt werden. Wenn dies nicht erwünscht ist, verwenden `Concat` stattdessen `Concat` .

## ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

## Auflisten

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

## Anzahl

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

## ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

## ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

## SkipWährend

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);
```

```
Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

### TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

### DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

### Aggregat (falten)

In jedem Schritt ein neues Objekt erzeugen:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

In allen Schritten dasselbe Objekt verwenden:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Verwenden eines Ergebniswählers:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

Wenn ein Seed weggelassen wird, wird das erste Element zum Seed:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

## Nachschlagen

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

## Beitreten

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

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
```

```
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

## GroupJoin

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

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
```

```
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}

//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

## Besetzung

`Cast` unterscheidet sich von den anderen Methoden von `Enumerable` darin, dass es eine Erweiterungsmethode für `IEnumerable` , nicht für `IEnumerable<T>` . Somit können Instanzen des ersteren in Instanzen des späteren umgewandelt werden.

Dies wird nicht kompiliert, da `ArrayList IEnumerable<T>` nicht implementiert:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

Das funktioniert wie erwartet:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` führt **keine** Konvertierungscasts durch. Die folgenden Kompilierungen `InvalidCastException` zur Laufzeit aus:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

Die richtige Methode zum Umwandeln einer Besetzung in eine Sammlung ist wie folgt:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

## Leeren

So erstellen Sie ein leeres IEnumerable von int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

Dieses leere IEnumerable wird für jeden Typ T zwischengespeichert, so dass:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

## ThenBy

`ThenBy` kann nur nach einer `OrderBy` Klausel verwendet werden, die die Bestellung anhand mehrerer Kriterien ermöglicht

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

## Angebot

Die beiden Parameter für `Range` sind die *erste* Anzahl und die *Anzahl* der zu erzeugenden Elemente (nicht die letzte Anzahl).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

## Linke äußere Verbindung

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };

    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

## Wiederholen

`Enumerable.Repeat` generiert eine Folge eines wiederholten Werts. In diesem Beispiel wird 4-mal "Hallo" generiert.

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
```

```
    Console.WriteLine(item);
}

/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

LINQ online lesen: https://riptutorial.com/de/dot-net/topic/34/linq

# Kapitel 25: Managed Extensibility Framework

## Bemerkungen

Ein großer Vorteil von MEF gegenüber anderen Technologien, die das Control Inversion-of-Control-Verfahren unterstützen, ist die Unterstützung von Abhängigkeiten, die zur Entwurfszeit nicht bekannt sind.

Alle Beispiele erfordern einen Verweis auf die System.ComponentModel.Composition-Assembly.

Alle (Basis-) Beispiele verwenden diese auch als Beispiel für Geschäftsobjekte:

```
using System.Collections.ObjectModel;

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

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

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

## Examples

### Typ exportieren (Basic)

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

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

```
                new User(0, "admin"),
                new User(1, "Dennis"),
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

Dies könnte praktisch überall definiert werden. Wichtig ist nur, dass die Anwendung weiß, wo sie suchen soll (über die erstellten ComposablePartCatalogs).

## Importieren (Basic)

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

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

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

Dies ist ein Typ, der von einem `IUserProvider` , der an einer `IUserProvider` Stelle definiert werden kann. Wie im vorherigen Beispiel ist es nur wichtig, dass die Anwendung weiß, wo nach dem entsprechenden Export gesucht werden soll (über die erstellten ComposablePartCatalogs).

## Anschließen (Basic)

Siehe die anderen (grundlegenden) Beispiele oben.

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

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

```
                UserWriter writer = new UserWriter();

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

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

Solange etwas im Assembly-Suchpfad der Anwendung `[Export(typeof(IUserProvider))]` , `UserWriter` der entsprechende Import von `UserWriter` erfüllt und die Benutzer werden gedruckt.

Andere Arten von Katalogen (z. B. `DirectoryCatalog` ) können anstelle von (oder zusätzlich zu) `ApplicationCatalog` , um an anderen Stellen nach Exporten zu suchen, die die Importe erfüllen.

Managed Extensibility Framework online lesen: https://riptutorial.com/de/dot-net/topic/62/managed-extensibility-framework

# Kapitel 26: Mit SHA1 in C # arbeiten

## Einführung

In diesem Projekt sehen Sie, wie Sie mit der kryptographischen Hash-Funktion von SHA1 arbeiten. Holen Sie sich zum Beispiel Hash von String und wie man SHA1-Hash knackt. Quelle auf Git Hub: https://github.com/mahdiabasi/SHA1Tool

## Examples

### #Erzeugen Sie die SHA1-Prüfsumme einer Dateifunktion

Zuerst fügen Sie Ihrem Projekt System.Security.Cryptography und System.IO hinzu

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Mit SHA1 in C # arbeiten online lesen: https://riptutorial.com/de/dot-net/topic/9457/mit-sha1-in-c-sharp-arbeiten

# Kapitel 27: Mit SHA1 in C # arbeiten

## Einführung

In diesem Projekt sehen Sie, wie Sie mit der kryptographischen Hash-Funktion von SHA1 arbeiten. Holen Sie sich zum Beispiel Hash von String und wie man SHA1-Hash knackt.

Quellcode auf github: https://github.com/mahdiabasi/SHA1Tool

## Examples

### #Erzeugen Sie die SHA1-Prüfsumme einer Datei

Zuerst fügen Sie Ihrem Projekt System.Security.Cryptography-Namespace hinzu

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

### #Generieren Sie den Hash eines Textes

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

Mit SHA1 in C # arbeiten online lesen: https://riptutorial.com/de/dot-net/topic/9458/mit-sha1-in-c-sharp-arbeiten

# Kapitel 28: Müllsammlung

## Einführung

In .Net werden Objekte, die mit new () erstellt wurden, auf dem verwalteten Heapspeicher zugewiesen. Diese Objekte werden von dem Programm, das sie verwendet, niemals explizit abgeschlossen. Stattdessen wird dieser Prozess vom .Net Garbage Collector gesteuert.

Einige der folgenden Beispiele sind "Laborfälle", um den Garbage Collector bei der Arbeit und einige wichtige Details seines Verhaltens zu zeigen, während andere sich darauf konzentrieren, wie Klassen für die korrekte Handhabung durch den Garbage Collector vorbereitet werden.

## Bemerkungen

Der Garbage Collector zielt darauf ab, die Programmkosten in Bezug auf den zugewiesenen Speicher zu senken, dies hat jedoch Kosten in Bezug auf die Verarbeitungszeit. Um einen guten Gesamtkompromiss zu erreichen, gibt es eine Reihe von Optimierungen, die bei der Programmierung mit dem Garbage Collector berücksichtigt werden sollten:

- Wenn die Collect () - Methode explizit aufgerufen werden soll (was ohnehin selten der Fall sein sollte), sollten Sie den "optimierten" Modus in Betracht ziehen, der das Dead-Objekt nur dann finalisiert, wenn tatsächlich Speicher benötigt wird
- Anstatt die Collect () - Methode aufzurufen, sollten Sie die AddMemoryPressure () - und RemoveMemoryPressure () - Methode verwenden, die nur dann eine Speichersammlung auslöst, wenn sie tatsächlich benötigt wird
- Es kann nicht garantiert werden, dass eine Speichersammlung alle toten Objekte abschließt. Stattdessen verwaltet der Garbage Collector 3 "Generationen", ein Objekt, das manchmal von einer Generation in die nächste "überlebt"
- Abhängig von verschiedenen Faktoren, einschließlich der Feinabstimmung der Einrichtung, können mehrere Threading-Modelle angewendet werden, die zu unterschiedlichen Interferenzen zwischen dem Garbage Collector-Thread und den anderen Anwendungsthread (en) führen.

## Examples

### Ein einfaches Beispiel für die (Müll-) Sammlung

In der folgenden Klasse gegeben:

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

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

Ein Programm, das eine Instanz erstellt, auch ohne sie zu verwenden:

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

Erzeugt die folgende Ausgabe:

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

Wenn nichts anderes passiert, wird das Objekt erst abgeschlossen, wenn das Programm beendet ist (wodurch alle Objekte auf dem verwalteten Heap freigegeben werden, wodurch diese im Prozess abgeschlossen werden).

Es ist möglich, den Garbage Collector wie folgt an einem bestimmten Punkt auszuführen:

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

Welches führt zu folgendem Ergebnis:

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

Sobald der Garbage Collector aufgerufen wurde, wurde dieses Mal das nicht verwendete (auch als "tot") Objekt abgeschlossen und aus dem verwalteten Heap freigegeben.

## Lebende Objekte und tote Objekte - die Grundlagen

Faustregel: Wenn eine Speicherbereinigung auftritt, sind "lebende Objekte" diejenigen, die noch verwendet werden, während "tote Objekte" solche sind, die nicht mehr verwendet werden. .

Im folgenden Beispiel (der Einfachheit halber sind FinalizableObject1 und FinalizableObject2 Unterklassen von FinalizableObject aus dem obigen Beispiel und erben daher das Verhalten der Initialisierungs- / Finalisierungsnachrichten):

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

Die Ausgabe wird sein:

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

```
<namespace>.FinalizableObject1 finalized
```

Zum Zeitpunkt, zu dem der Garbage Collector aufgerufen wird, ist FinalizableObject1 ein totes Objekt und wird finalisiert, während FinalizableObject2 ein Live-Objekt ist und auf dem verwalteten Heap verbleibt.

## Mehrere tote Objekte

Was ist, wenn zwei (oder mehrere) ansonsten tote Objekte sich aufeinander beziehen? Dies wird im folgenden Beispiel gezeigt. Angenommen, OtherObject ist eine öffentliche Eigenschaft von FinalizableObject:

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

Dies erzeugt die folgende Ausgabe:

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

Die beiden Objekte werden finalisiert und aus dem verwalteten Heap freigegeben, obwohl sie sich gegenseitig referenzieren (da zu keinem Objekt ein anderer Verweis von einem tatsächlich aktiven Objekt vorhanden ist).

## Schwache Referenzen

Schwache Verweise sind ... Verweise auf andere Objekte (auch als "Ziele" bezeichnet), aber "schwach", da sie die Sammlung von Objekten nicht verhindern. Mit anderen Worten, schwache Referenzen zählen nicht, wenn der Garbage Collector Objekte als "live" oder "dead" bewertet.

Der folgende Code:

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

Erzeugt die Ausgabe:

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

Das Objekt wird vom verwalteten Heap freigegeben, obwohl es von der WeakReference-Variablen referenziert wird (noch im Gültigkeitsbereich, wenn der Garbage-Collector aufgerufen wurde).

Konsequenz # 1: Es ist zu jedem Zeitpunkt unsicher, anzunehmen, ob ein WeakReference-Ziel auf dem verwalteten Heap noch zugewiesen ist oder nicht.

Konsequenz # 2: Immer wenn ein Programm auf das Ziel einer Schwachstellenreferenz zugreifen muss, sollte Code für beide Fälle bereitgestellt werden, wobei das Ziel noch zugewiesen ist oder nicht. Die Methode für den Zugriff auf das Ziel ist TryGetTarget:

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

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

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

Die generische Version von WeakReference ist seit .Net 4.5 verfügbar. Alle Framework-Versionen bieten eine nicht generische, nicht typisierte Version, die auf dieselbe Weise erstellt und wie folgt geprüft wird:

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

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

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

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

## Entsorgen () vs. Finalisierer

Implementieren Sie die Dispose () - Methode (und deklarieren Sie die enthaltende Klasse als IDisposable), um sicherzustellen, dass speicherintensive Ressourcen freigegeben werden, sobald das Objekt nicht mehr verwendet wird. Der "Haken" ist, dass es keine starke Garantie gibt, dass die Dispose () -Methode jemals aufgerufen wird (im Gegensatz zu Finalizern, die immer am Ende

der Lebensdauer des Objekts aufgerufen werden).

Ein Szenario ist ein Programm, das Dispose () für explizit erstellte Objekte aufruft:

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

    // ... Use that object

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

    // ... Do other stuff

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

Ein anderes Szenario deklariert eine Klasse, die vom Framework instanziiert werden soll. In diesem Fall erbt die neue Klasse normalerweise eine Basisklasse, beispielsweise erstellt sie in MVC eine Controller-Klasse als Unterklasse von System.Web.Mvc.ControllerBase. Wenn die Basisklasse die IDisposable-Schnittstelle implementiert, ist dies ein guter Hinweis darauf, dass Dispose () ordnungsgemäß vom Framework aufgerufen wird - es gibt jedoch keine starke Garantie.

Daher ist Dispose () kein Ersatz für einen Finalizer. stattdessen sollten die beiden für unterschiedliche Zwecke verwendet werden:

- Ein Finalizer setzt schließlich Ressourcen frei, um Speicherverluste zu vermeiden, die andernfalls auftreten würden
- Dispose () gibt Ressourcen frei (möglicherweise dieselben), sobald diese nicht mehr benötigt werden, um die allgemeine Speicherzuordnung zu entlasten.

## Ordnungsgemäße Entsorgung und Fertigstellung von Objekten

Da Dispose () und Finalizer auf unterschiedliche Zwecke abzielen, sollte eine Klasse, die externe speicherintensive Ressourcen verwaltet, beide implementieren. Die Folge ist, die Klasse so zu schreiben, dass sie zwei mögliche Szenarien gut handhabt:

- Wenn nur der Finalizer aufgerufen wird
- Wenn zuerst Dispose () aufgerufen wird und später auch der Finalizer

Eine Lösung besteht darin, den Bereinigungscode so zu schreiben, dass ein ein- oder zweimaliges Ausführen das gleiche Ergebnis liefert wie das einmalige Ausführen. Die Machbarkeit hängt von der Art der Bereinigung ab, zum Beispiel:

- Das Schließen einer bereits geschlossenen Datenbankverbindung hat wahrscheinlich keine Auswirkungen und funktioniert daher
- Das Aktualisieren einiger "Verbrauchszähler" ist gefährlich und führt bei zweimaligem Aufruf

zu einem falschen Ergebnis.

Eine sicherere Lösung stellt durch Entwurf sicher, dass der Bereinigungscode unabhängig vom externen Kontext einmalig aufgerufen wird. Dies kann auf "klassische Weise" mit einer eigenen Flagge erreicht werden:

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

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

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

Alternativ bietet der Garbage Collector eine bestimmte Methode SuppressFinalize (), die das Überspringen des Finalizers ermöglicht, nachdem Dispose aufgerufen wurde:

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

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

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

Müllsammlung online lesen: https://riptutorial.com/de/dot-net/topic/9636/mullsammlung

# Kapitel 29: NuGet-Verpackungssystem

## Bemerkungen

NuGet.org :

> NuGet ist der Paketmanager für die Microsoft-Entwicklungsplattform einschließlich .NET. Die NuGet-Client-Tools bieten die Möglichkeit, Pakete zu produzieren und zu verwenden. Die NuGet Gallery ist das zentrale Paket-Repository, das von allen Paketautoren und -verbrauchern verwendet wird.

Bilder in Beispielen mit freundlicher Genehmigung von NuGet.org .

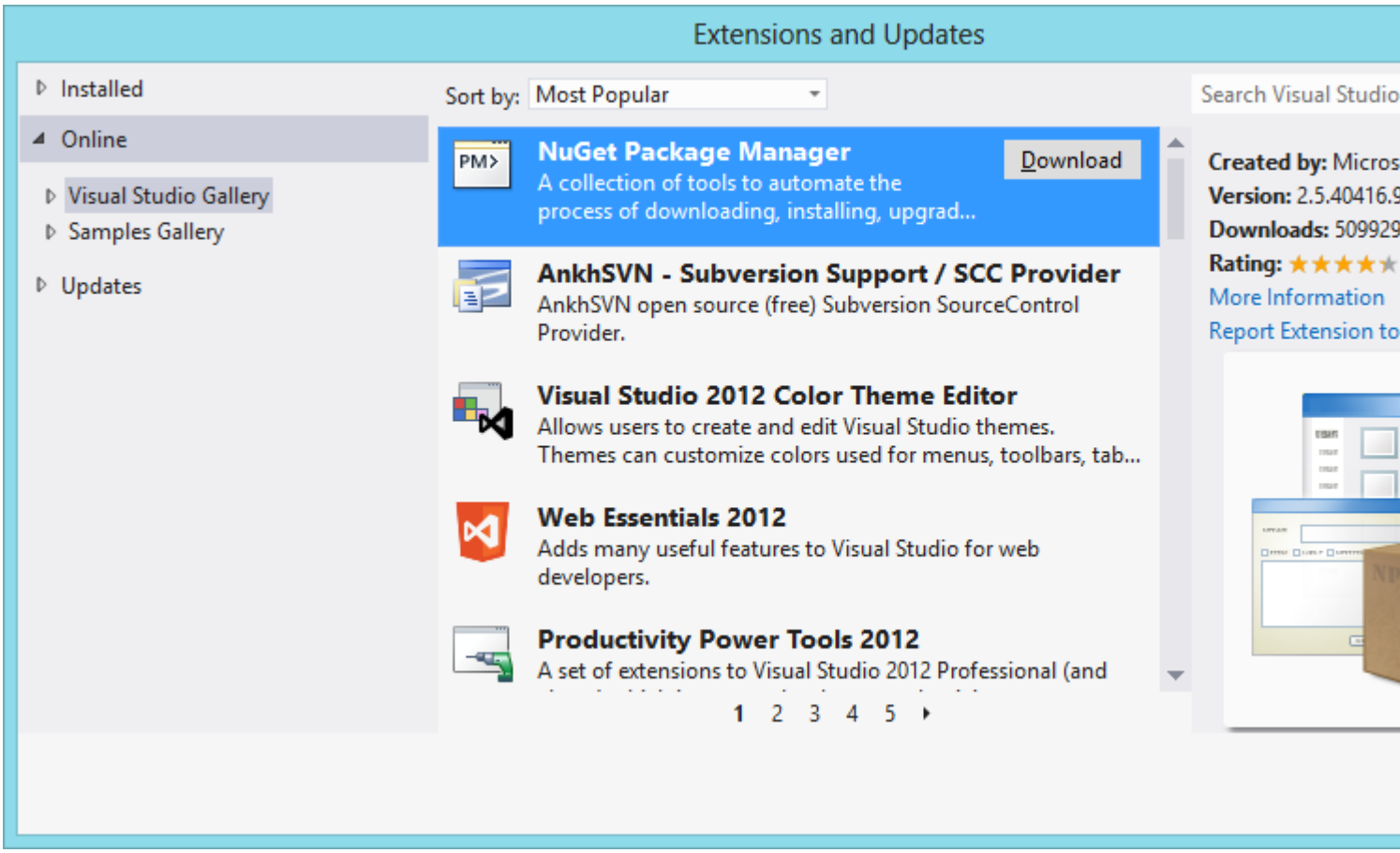
## Examples

### NuGet Package Manager installieren

Um Ihre Projektpakete verwalten zu können, benötigen Sie den NuGet Package Manager. Dies ist eine Visual Studio-Erweiterung, die in den offiziellen Dokumenten erklärt wird: Installieren und Aktualisieren von NuGet Client .

Ab Visual Studio 2012 ist NuGet in jeder Edition enthalten und kann verwendet werden: Tools -> NuGet Package Manager -> Package Manager Console.

Klicken Sie dazu im Menü Extras von Visual Studio auf Erweiterungen und Updates:




Dies installiert sowohl die GUI:

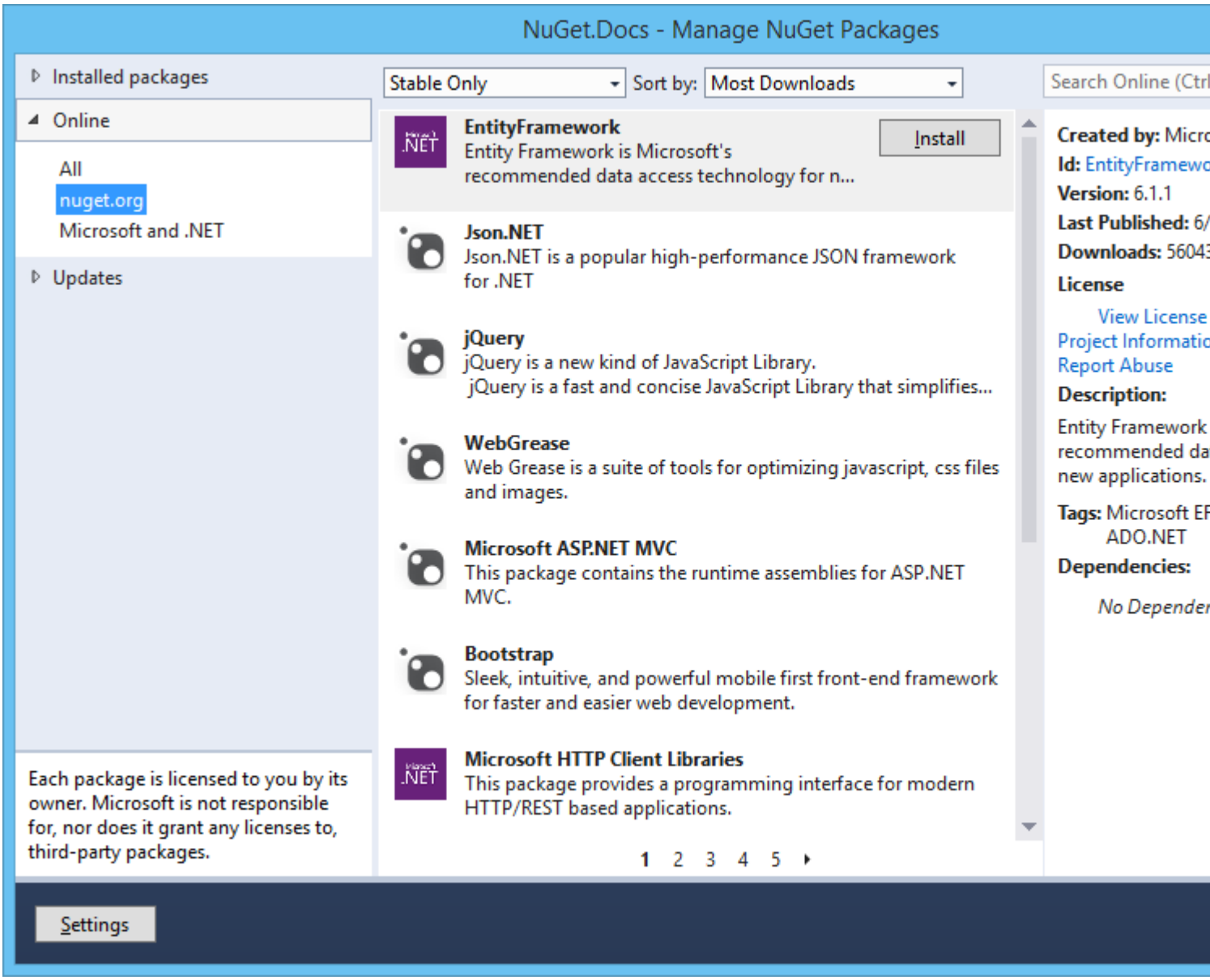
- Verfügbar durch Klicken auf "NuGet-Pakete verwalten ..." in einem Projekt oder im Ordner "Referenzen"

Und die Package Manager Console:

- Extras -> NuGet Package Manager -> Package Manager Console.

## Verwalten von Paketen über die Benutzeroberfläche

Wenn Sie mit der rechten Maustaste auf ein Projekt (oder den Ordner "Referenzen") klicken, können Sie auf die Option "NuGet-Pakete verwalten ..." klicken. Dies zeigt den Package Manager-Dialog .




## Pakete über die Konsole verwalten

Klicken Sie auf die Menüs Extras -> NuGet Package Manager -> Package Manager Console, um die Konsole in Ihrer IDE anzuzeigen. Offizielle Dokumentation hier .

Hier können Sie unter anderem `install-package` ausgeben, mit denen das eingegebene Paket im aktuell ausgewählten "Standardprojekt" installiert wird:

```
Install-Package Elmah
```

Sie können auch das Projekt angeben, in dem das Paket installiert werden soll, und das ausgewählte Projekt in der Dropdown-Liste "Standardprojekt" überschreiben:

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

## Paket aktualisieren

Verwenden Sie zum Aktualisieren eines Pakets den folgenden Befehl:

```
PM> Update-Package EntityFramework
```

Dabei ist EntityFramework der Name des zu aktualisierenden Pakets. Beachten Sie, dass das Update für alle Projekte ausgeführt wird und sich daher von `Install-Package EntityFramework` das nur in "Standardprojekt" installiert wird.

Sie können auch ein einzelnes Projekt explizit angeben:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

## Paket deinstallieren

```
PM> Uninstall-Package EntityFramework
```

## Deinstallieren eines Pakets aus einem Projekt in einer Lösung

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

## Eine bestimmte Version eines Pakets installieren

```
PM> Install-Package EntityFramework -Version 6.1.2
```

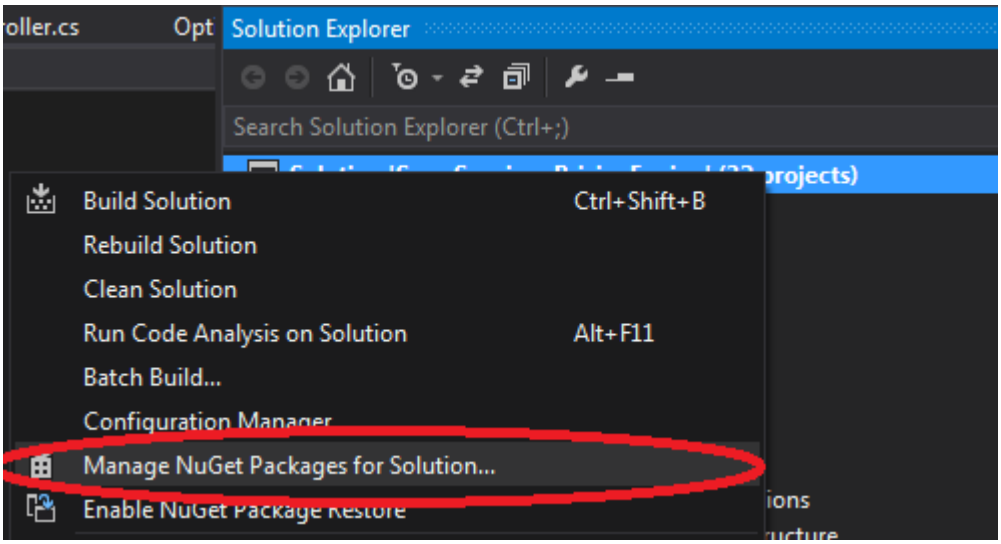
## Hinzufügen eines Paketquellen-Feeds (MyGet, Klondike, ect)

```
nuget sources add -name feedname -source http://sourcefeedurl
```

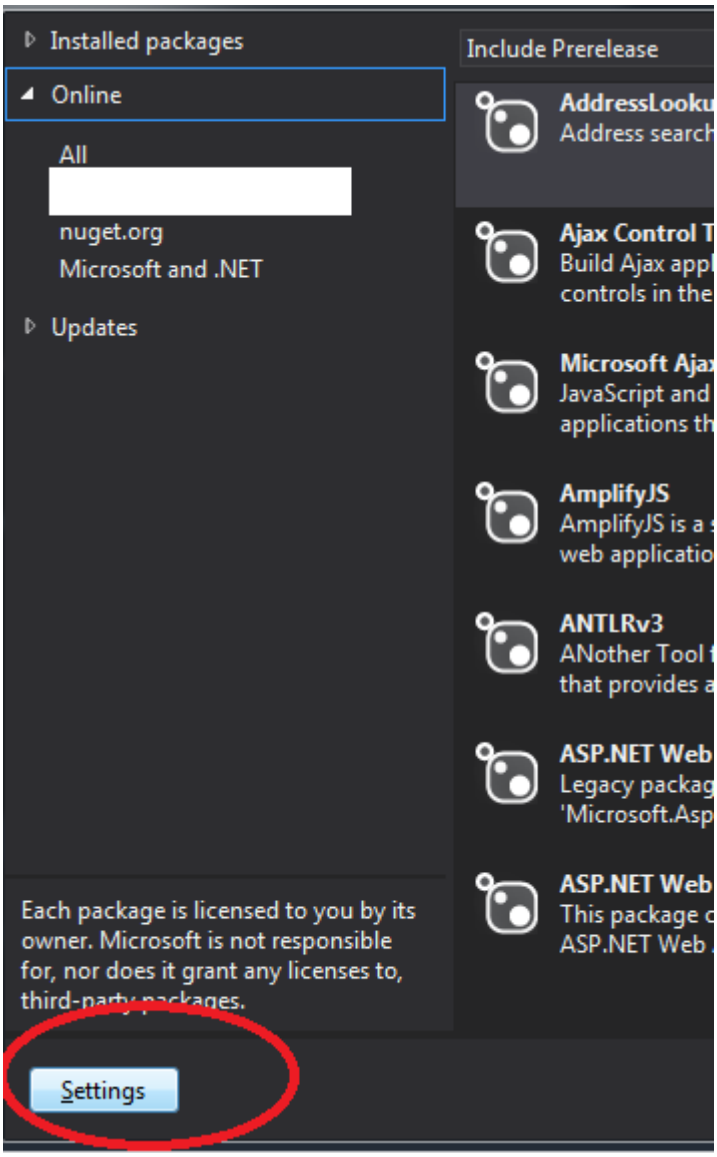
## Verwenden verschiedener (lokaler) Nuget-Paketquellen mithilfe der Benutzeroberfläche

Es ist für Unternehmen üblich, einen eigenen Nugetserver für die Verteilung von Paketen auf verschiedene Teams einzurichten.

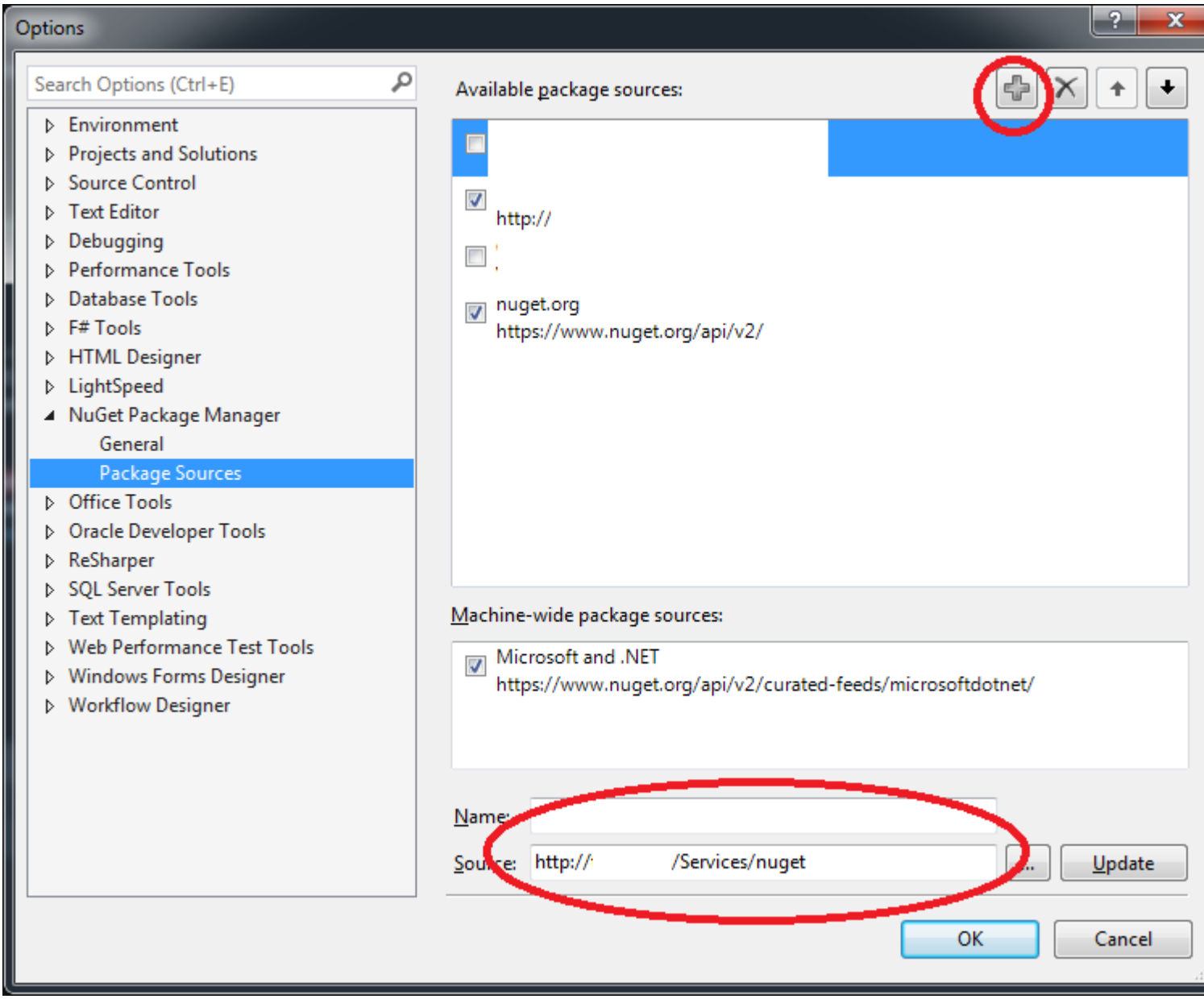
1. Gehen Sie zum Projektmappen-Explorer und klicken Sie mit der `rechten` Maustaste. Wählen Sie dann `Manage NuGet Packages for Solution`




2. In dem sich öffnenden Fenster klicken Sie auf `Settings`




3. Klicken Sie auf `+` in der oberen rechten Ecke und fügen Sie Namen und URL hinzu, die auf Ihren lokalen Nuget-Server zeigen.




## Deinstallieren Sie eine bestimmte Version des Pakets

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

NuGet-Verpackungssystem online lesen: https://riptutorial.com/de/dot-net/topic/43/nuget-verpackungssystem

# Kapitel 30: Parallele Verarbeitung mit .Net Framework

## Einführung

Dieses Thema behandelt die Multi-Core-Programmierung mit Task Parallel Library mit .NET Framework. Mit der Task-Parallel-Bibliothek können Sie Code schreiben, der für Menschen lesbar ist, und passt sich der Anzahl der verfügbaren Kerne an. So können Sie sicher sein, dass Ihre Software sich automatisch mit der Aktualisierungsumgebung aktualisiert.

## Examples

### Parallele Erweiterungen

Parallele Erweiterungen wurden zusammen mit der Task Parallel Library eingeführt, um Datenparallelität zu erreichen. Datenparallelität bezieht sich auf Szenarien, in denen dieselbe Operation gleichzeitig (dh parallel) für Elemente in einer Quellensammlung oder einem Array ausgeführt wird. .NET bietet neue Konstrukte, um Datenparallelität mithilfe von Parallel.For- und Parallel.Foreach-Konstrukten zu erreichen.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

Das oben erwähnte Parallel.ForEach-Konstrukt verwendet die mehreren Kerne und verbessert somit die Leistung auf dieselbe Weise.

Parallele Verarbeitung mit .Net Framework online lesen: https://riptutorial.com/de/dot-net/topic/8085/parallele-verarbeitung-mit--net-framework

# Kapitel 31: Plattform aufrufen

## Syntax

- [DllImport ("Example.dll")] statisch extern void SetText (Zeichenfolge inString);
- [DllImport ("Example.dll")] statisch extern void GetText (StringBuilder outString);
- [MarshalAs (UnmanagedType.ByValTStr, SizeConst = 32)] Zeichenfolgentext;
- [MarshalAs (UnmanagedType.ByValArray, SizeConst = 128)] Byte [] ByteArr;
- [StructLayout (LayoutKind.Sequential)] public struct PERSON {...}
- [StructLayout (LayoutKind.Explicit)] öffentliche Struktur MarshaledUnion {[FieldOffset (0)] ...}

## Examples

### Aufrufen einer Win32-DLL-Funktion

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int
options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Deklarieren Sie eine Funktion als `static extern` indem Sie `DllImportAttribute` mit der Eigenschaft `Value` auf .dll name setzen. Vergessen Sie nicht, den Namespace `System.Runtime.InteropServices` zu verwenden. Dann rufen Sie es als normale statische Methode auf.

Die Platform Invocation Services kümmern sich darum, die DLL zu laden und die gewünschte Funktion zu finden. In den meisten einfachen Fällen wird das P / Invoke auch Parameter marshallieren und Werte von und zur DLL zurückgeben (dh Konvertierung von .NET-Datentypen in Win32-Datentypen und umgekehrt).

### Windows-API verwenden

Verwenden Sie pinvoke.net .

Bevor Sie eine `extern` Windows-API-Funktion in Ihrem Code deklarieren, sollten Sie sie auf pinvoke.net suchen . Sie haben höchstwahrscheinlich bereits eine passende Erklärung mit allen unterstützenden Typen und guten Beispielen.

### Marshalling-Arrays

### Arrays einfacher Art

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

### Arrays von Zeichenfolgen

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

## Marshaling-Strukturen

### Einfache Struktur

C ++ - Signatur:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

C # -Definition

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

### Struktur mit Arrayfeldern unbekannter Größe. Übergeben

C ++ - Signatur

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

Wenn dieser Code von verwaltetem an nicht verwalteten Code übergeben wird

Das `data` ist als IntPtr definiert und Speicher sollte explizit zugewiesen `Marshal.AllocHGlobal()` (und befreit `Marshal.FreeHGlobal()` anschliessend):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

### Struktur mit Arrayfeldern unbekannter Größe. Empfang

C ++ - Signatur:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

Wenn solche Daten aus nicht verwaltetem Code übergeben werden und der Speicher von den nicht verwalteten Funktionen zugewiesen wird, sollte der verwaltete Aufrufer sie in eine `IntPrt` Variable empfangen und den Puffer in ein verwaltetes Array konvertieren. Im Fall von Strings gibt es eine praktische `Marshal.PtrToStringAnsi()` -Methode:

```
[StructLayout(LayoutKind.Sequential)]
```

```
public struct USER
{
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

## Gewerkschaften marschieren

### Nur Felder mit Werttyp

C ++ - Deklaration

```
typedef union
{
    char c;
    int i;
} CharOrInt;
```

C # -Deklaration

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

### Wertetyp und Referenzfelder mischen

Die Überlappung eines Referenzwerts mit dem Werttyp Eins ist nicht zulässig. Sie können also nicht einfach den ~~`FieldOffset(0) text; FieldOffset(0) i;`~~ wird nicht für kompilieren

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

Im Allgemeinen müssten Sie ein benutzerdefiniertes Marshalling verwenden. In besonderen Fällen wie dieser können jedoch einfachere Techniken verwendet werden:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Plattform aufrufen online lesen: https://riptutorial.com/de/dot-net/topic/1643/plattform-aufrufen

# Kapitel 32: Prozess- und Thread-Affinitätseinstellung

## Parameter

| Parameter | Einzelheiten |
|---|---|
| Affinität | Ganzzahl, die die Menge der Prozessoren beschreibt, auf denen der Prozess ausgeführt werden darf. Wenn Sie beispielsweise möchten, dass Ihr Prozess auf einem 8-Prozessor-System nur auf den Prozessoren 3 und 4 ausgeführt wird, wählen Sie die Affinität wie folgt aus: 00001100 = 12 |

## Bemerkungen

Die Prozessoraffinität eines Threads ist die Menge von Prozessoren, zu denen er eine Beziehung hat. Mit anderen Worten, die, auf denen es ausgeführt werden kann.

Prozessoraffinität stellt jeden Prozessor als ein Bit dar. Bit 0 steht für Prozessor Eins, Bit 1 für Prozessor Zwei usw.

## Examples

### Holen Sie sich die Prozessaffinitätsmaske

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
```

```
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Verwendungsbeispiel:

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

## Legen Sie die Prozessaffinitätsmaske fest

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Verwendungsbeispiel:

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

Prozess- und Thread-Affinitätseinstellung online lesen: https://riptutorial.com/de/dot-net/topic/4431/prozess--und-thread-affinitatseinstellung

# Kapitel 33: ReadOnlyCollections

## Bemerkungen

Eine `ReadOnlyCollection` bietet eine schreibgeschützte Ansicht für eine vorhandene Auflistung (die 'Quellauflistung').

Elemente werden nicht direkt zu einer `ReadOnlyCollection` hinzugefügt oder daraus entfernt. Sie werden stattdessen hinzugefügt und aus der Quellensammlung entfernt, und die `ReadOnlyCollection` gibt diese Änderungen an der Quelle wieder.

Die Anzahl und Reihenfolge der Elemente in einer `ReadOnlyCollection` kann nicht geändert werden. Die Eigenschaften der Elemente und die Methoden können jedoch aufgerufen werden, sofern sie im Gültigkeitsbereich liegen.

Verwenden Sie eine `ReadOnlyCollection` wenn Sie externen Code erlauben möchten, Ihre Sammlung anzuzeigen, ohne sie ändern zu können. Sie können die Sammlung jedoch weiterhin selbst ändern.

Siehe auch

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

## ReadOnlyCollections vs ImmutableCollection

Eine `ReadOnlyCollection` unterscheidet sich von einer `ImmutableCollection` dass Sie eine `ImmutableCollection` nicht bearbeiten können, wenn Sie sie erstellt haben. Sie enthält immer `n` Elemente und kann nicht ersetzt oder neu angeordnet werden. Eine `ReadOnlyCollection` kann dagegen nicht direkt bearbeitet werden. Elemente können jedoch weiterhin mit der Quellensammlung hinzugefügt / entfernt / neu angeordnet werden.

## Examples

### ReadOnlyCollection erstellen

## Verwenden des Konstruktors

Eine `ReadOnlyCollection` wird erstellt, indem ein vorhandenes `IList` Objekt an den Konstruktor übergeben wird:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

# LINQ verwenden

Zusätzlich bietet LINQ eine `AsReadOnly()` Erweiterungsmethode für `IList` Objekte:

```
var readOnlyVersion = groceryList.AsReadOnly();
```

# Hinweis

Normalerweise möchten Sie die Quellensammlung privat pflegen und öffentlichen Zugriff auf die `ReadOnlyCollection` gewähren. Während Sie eine `ReadOnlyCollection` aus einer Inline-Liste erstellen konnten, können Sie die Sammlung nach dem Erstellen nicht mehr ändern.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

Wenn Sie dies tun, möchten Sie möglicherweise eine andere Datenstruktur verwenden, z. B. eine `ImmutableCollection` .

## Aktualisieren einer ReadOnlyCollection

Eine `ReadOnlyCollection` kann nicht direkt bearbeitet werden. Stattdessen wird die Quellensammlung aktualisiert und die `ReadOnlyCollection` berücksichtigt diese Änderungen. Dies ist die `ReadOnlyCollection` der `ReadOnlyCollection` .

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                // ..but they can be added to the original
collection

itemCount = readOnlyGroceryList.Count;      // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now
"Vitamins"
```

Demo anzeigen

## Warnung: Elemente in einer ReadOnlyCollection sind an sich nicht schreibgeschützt

Wenn die Quellensammlung nicht unveränderlich ist, können Elemente, auf die über eine `ReadOnlyCollection` zugegriffen wird, geändert werden.

```
public class Item
```

```
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

Demo anzeigen

ReadOnlyCollections online lesen: https://riptutorial.com/de/dot-net/topic/6906/readonlycollections

# Kapitel 34: Reflexion

## Examples

### Was ist eine Versammlung?

Baugruppen sind der Baustein für jede CLR- Anwendung (Common Language Runtime) . Jeder von Ihnen definierte Typ wird zusammen mit seinen Methoden, Eigenschaften und seinem Bytecode innerhalb einer Assembly kompiliert und verpackt.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Assemblies sind selbstdokumentierend: Sie enthalten nicht nur Typen, Methoden und ihren IL-Code, sondern auch die Metadaten, die für die Prüfung und den Verbrauch erforderlich sind, sowohl beim Kompilieren als auch zur Laufzeit:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Assemblies haben Namen, die ihre vollständige, eindeutige Identität beschreiben:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Wenn dieser Name ein `PublicKeyToken` , wird es als *starker Name bezeichnet* . Eine Assembly mit starker Benennung bezeichnet das Erstellen einer Signatur mithilfe des privaten Schlüssels, der dem mit der Assembly verteilten öffentlichen Schlüssel entspricht. Diese Signatur wird dem Assemblymanifest hinzugefügt, das die Namen und Hashes aller Dateien enthält, aus denen die Assembly besteht, und ihr `PublicKeyToken` wird Teil des Namens. Baugruppen mit demselben starken Namen sollten identisch sein. Starke Namen werden bei der Versionierung und zur Vermeidung von Assemblykonflikten verwendet.

### So erstellen Sie ein Objekt aus T mit Reflection

Verwenden des Standardkonstruktors

```
T variable = Activator.CreateInstance(typeof(T));
```

Verwendung eines parametrisierten Konstruktors

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

## Erstellen von Objekten und Festlegen von Eigenschaften mithilfe von Reflektionen

Nehmen wir an, wir haben eine Klasse `Classy` , die die Eigenschaft Properties besitzt

```
public class Classy
{
    public string Propertua {get; set;}
}
```

`Propertua` mit Reflektion `Propertua` :

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

## Ein Attribut einer Aufzählung mit Reflektion erhalten (und zwischenspeichern)

Attribute können nützlich sein, um Metadaten in Enums zu kennzeichnen. Der Wert dieses Wertes kann langsam sein, daher müssen die Ergebnisse zwischengespeichert werden.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object,
object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
```

```
    }
```

## Vergleichen Sie zwei Objekte mit Reflexion

```
public class Equatable
{
    public string field1;

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }

    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^
field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

**Hinweis: In** diesem Beispiel wird zur Vereinfachung ein Vergleich zwischen Feldern (statische Felder und Eigenschaften werden ignoriert) durchgeführt

Reflexion online lesen: https://riptutorial.com/de/dot-net/topic/44/reflexion

# Kapitel 35: Reguläre Ausdrücke (System.Text.RegularExpressions)

## Examples

### Prüfen Sie, ob das Muster mit der Eingabe übereinstimmt

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

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

### Optionen übergeben

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

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

### Einfaches Spiel und Ersetzen

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

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

### Spiel in Gruppen zusammen

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

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

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

```
}
```

### Entfernen Sie nicht alphanumerische Zeichen aus der Zeichenfolge

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

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

### Finde alle Übereinstimmungen

# Verwenden

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

# Code

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

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

# Ausgabe

```
Carrot
Cherry
Clementine
```

Reguläre Ausdrücke (System.Text.RegularExpressions) online lesen:
https://riptutorial.com/de/dot-net/topic/6944/regulare-ausdrucke--system-text-regularexpressions-

# Kapitel 36: Sammlungen

## Bemerkungen

Es gibt verschiedene Arten der Sammlung:

- `Array`
- `List`
- `Queue`
- `SortedList`
- `Stack`
- Wörterbuch

## Examples

### Erstellen einer initialisierten Liste mit benutzerdefinierten Typen

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

Hier haben wir eine Klasse ohne Konstruktor mit zwei Eigenschaften: `Name` und eine boolesche, nullfähige Eigenschaft `Selected` . Wenn wir eine `List<Model>` initialisieren wollten, gibt es verschiedene Möglichkeiten, dies auszuführen.

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

Hier erstellen wir mehrere `new` Instanzen unserer `Model` Klasse und initialisieren sie mit Daten. Was ist, wenn wir einen Konstruktor hinzufügen?

```
public class Model
{

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

Dadurch können wir unsere Liste *etwas* anders initialisieren.

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

Was ist mit einer Klasse, bei der eine der Eigenschaften selbst eine Klasse ist?

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

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

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

Beachten Sie, dass wir den Konstruktor auf die Klasse `Model` , um das Beispiel etwas zu vereinfachen.

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

Beachten Sie, dass wir unsere `List<ExtendedModel>` mit `Collection<ExtendedModel>` , `ExtendedModel[]` , `object[]` oder einfach `[]` austauschen können.

## Warteschlange

In .Net gibt es eine Sammlung, die zum Verwalten von Werten in einer `Queue` , die das FIFO-Konzept (First-In-First-Out) verwendet . Die `Enqueue(T item)` der Warteschlangen ist die Methode `Enqueue(T item)` der Elemente in die Warteschlange `Dequeue()` und `Dequeue()` der das erste Element `Dequeue()` und aus der Warteschlange entfernt wird. Die generische Version kann wie der folgende

Code für eine Warteschlange von Zeichenfolgen verwendet werden.

Fügen Sie zuerst den Namespace hinzu:

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

und benutze es:

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

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

Es gibt eine nicht generische Version des Typs, die mit Objekten arbeitet.

Der Namensraum ist:

```
using System.Collections;
```

Fügen Sie ein Codebeispiel für eine nicht generische Warteschlange hinzu:

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

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

Es gibt auch eine Methode namens Peek (), die das Objekt am Anfang der Warteschlange zurückgibt, ohne die Elemente zu entfernen.

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

foreach (int element in queue)
{
```

```
    Console.WriteLine(i);
}
```

Die Ausgabe (ohne zu entfernen):

```
10
20
30
40
50
```

## Stapel

Es gibt eine Sammlung in .NET, die zum Verwalten von Werten in einem `Stack` , die das LIFO-Konzept (Last-In-First-Out) verwendet . Die Grundlagen von Stapeln sind die Methode `Push(T item)` der Elemente im Stack hinzugefügt werden, und `Pop()` der das letzte Element hinzugefügt und aus dem Stack entfernt wird. Die generische Version kann wie der folgende Code für eine Warteschlange von Zeichenfolgen verwendet werden.

Fügen Sie zuerst den Namespace hinzu:

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

und benutze es:

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

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

Es gibt eine nicht generische Version des Typs, die mit Objekten arbeitet.

Der Namensraum ist:

```
using System.Collections;
```

Und ein Codebeispiel eines nicht generischen Stapels:

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

```
object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

Es gibt auch eine Methode namens Peek (), die das zuletzt hinzugefügte Element zurückgibt, ohne es aus dem `Stack` zu entfernen.

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

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

Es ist möglich, die Elemente auf dem Stapel zu durchlaufen und die Reihenfolge des Stapels (LIFO) einzuhalten.

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

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

Die Ausgabe (ohne zu entfernen):

```
50
40
30
20
10
```

## Sammlungsinitialisierer verwenden

Einige Sammlungstypen können zum Zeitpunkt der Deklaration initialisiert werden. Die folgende Anweisung erstellt und initialisiert beispielsweise die `numbers` mit einigen ganzen Zahlen:

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

Intern konvertiert der C # -Compiler diese Initialisierung tatsächlich in eine Reihe von Aufrufen der Add-Methode. Daher können Sie diese Syntax nur für Auflistungen verwenden, die die `Add` Methode tatsächlich unterstützen.

> Die Klassen `Stack<T>` und `Queue<T>` dies nicht.

Für komplexe Sammlungen wie die Klasse `Dictionary<TKey, TValue>` , die Schlüssel / Wert-Paare annehmen, können Sie jedes Schlüssel / Wert-Paar als anonymen Typ in der Initialisierungsliste angeben.

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

Das erste Element in jedem Paar ist der Schlüssel und das zweite ist der Wert.

Sammlungen online lesen: https://riptutorial.com/de/dot-net/topic/30/sammlungen

# Kapitel 37: Schreiben Sie in den StdErr-Stream und lesen Sie ihn aus

## Examples

### In die Standardfehlerausgabe mit Console schreiben

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

### Aus Standardfehler des untergeordneten Prozesses lesen

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

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

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

Schreiben Sie in den StdErr-Stream und lesen Sie ihn aus online lesen: https://riptutorial.com/de/dot-net/topic/10779/schreiben-sie-in-den-stderr-stream-und-lesen-sie-ihn-aus

# Kapitel 38: Serielle Ports

## Examples

### Grundbetrieb

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

### Listet die verfügbaren Portnamen auf

```
string[] portNames = SerialPort.GetPortNames();
```

### Asynchrones Lesen

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

### Synchrone Echo-Service

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
            serialPort.Close();
        }
    }
}
```

## Asynchroner Nachrichtenempfänger

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
```

```
                    var port = (SerialPort)sender;
                    int bytesToRead = port.BytesToRead;
                    if (bytesToRead > buffer.Length)
                        Array.Resize(ref buffer, bytesToRead);
                    int bytesRead = port.Read(buffer, 0, bytesToRead);
                    ProcessBuffer(buffer, bytesRead);
                    break;
                case SerialData.Eof:
                    terminateService.Set();
                    break;
            }
        }
    }
    static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
    {
        lock (eventLock)
            if (e.EventType == SerialError.TXFull)
            {
                Console.WriteLine("Error: TXFull. Can't handle this!");
                terminateService.Set();
            }
            else
            {
                Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                var port = (SerialPort)sender;
                port.DiscardInBuffer();
                port.DiscardOutBuffer();
                unprocessedBuffer = null;
                port.Write(new byte[] { NAK }, 0, 1);
            }
    }

    static void ProcessBuffer(byte[] buffer, int length)
    {
        List<byte> message = unprocessedBuffer;
        for (int i = 0; i < length; i++)
            if (buffer[i] == ETX)
            {
                if (message != null)
                {
                    Console.WriteLine("MessageReceived: {0}",
                        Encoding.ASCII.GetString(message.ToArray()));
                    message = null;
                }
            }
            else if (buffer[i] == STX)
                message = null;
            else if (message != null)
                message.Add(buffer[i]);
        unprocessedBuffer = message;
    }
}
}
```

Dieses Programm wartet auf Nachrichten, die in `STX` und `ETX` Bytes eingeschlossen sind, und gibt den dazwischen befindlichen Text aus. Alles andere wird weggeworfen. Bei Schreibpufferüberlauf wird es angehalten. Bei anderen Fehlern werden die Eingangs- und Ausgangspuffer zurückgesetzt und auf weitere Meldungen gewartet.

Der Code veranschaulicht:

- Asynchrones Lesen der seriellen Schnittstelle (siehe Verwendung von `SerialPort.DataReceived` ).
- Fehlerbehandlung beim seriellen Port (siehe Verwendung von `SerialPort.ErrorReceived` ).
- Nicht-textnachrichtenbasierte Protokollimplementierung
- Teilweises Lesen der Nachricht.
    - Das `SerialPort.DataReceived` Ereignis kann früher auftreten, als die gesamte Nachricht (bis zu `ETX` ) kommt. Möglicherweise ist auch nicht die gesamte Nachricht im Eingabepuffer verfügbar (SerialPort.Read (..., ..., port.BytesToRead) liest nur einen Teil der Nachricht. In diesem Fall speichern wir den empfangenen Teil ( `unprocessedBuffer` ) und warten weiter auf weitere Daten.
- Umgang mit mehreren Nachrichten auf einmal.
    - Das `SerialPort.DataReceived` Ereignis kann nur auftreten, nachdem vom anderen Ende mehrere Nachrichten gesendet wurden.

Serielle Ports online lesen: https://riptutorial.com/de/dot-net/topic/5366/serielle-ports

# Kapitel 39: SpeechRecognitionEngine-Klasse zum Erkennen von Sprache

## Syntax

- SpeechRecognitionEngine ()
- SpeechRecognitionEngine.LoadGrammar (Grammatikgrammatik)
- SpeechRecognitionEngine.SetInputToDefaultAudioDevice ()
- SpeechRecognitionEngine.RecognizeAsync (Modus "RecognizeMode")
- GrammarBuilder ()
- GrammarBuilder.Append (Auswahlmöglichkeiten)
- Auswahlmöglichkeiten (params string [] Wahlmöglichkeiten)
- Grammatik (GrammarBuilder-Builder)

## Parameter

| `LoadGrammar` : Parameter | Einzelheiten |
|---|---|
| Grammatik | Die zu ladende Grammatik. Zum Beispiel ein `DictationGrammar` Objekt, um `DictationGrammar` zuzulassen. |
| **`RecognizeAsync` : Parameter** | **Einzelheiten** |
| Modus | Der `RecognizeMode` für die aktuelle Erkennung: `Single` für nur eine Erkennung, `Multiple` für mehrere. |
| **`GrammarBuilder.Append` : Parameter** | **Einzelheiten** |
| Wahlmöglichkeiten | Hängt einige Auswahlmöglichkeiten an den Grammatik-Generator an. Das heißt, wenn der Benutzer Sprache eingibt, kann der Erkenner verschiedenen "Verzweigungen" aus einer Grammatik folgen. |
| **`Choices` Konstruktor: Parameter** | **Einzelheiten** |
| Wahlmöglichkeiten | Eine Reihe von Auswahlmöglichkeiten für den Grammatik-Generator. Siehe `GrammarBuilder.Append` . |
| **`Grammar` : Parameter** | **Einzelheiten** |
| Baumeister | Der `GrammarBuilder` zum `GrammarBuilder` einer `Grammar` . |

# Bemerkungen

Um `SpeechRecognitionEngine` , muss in Ihrer Windows-Version die Spracherkennung aktiviert sein.

Sie müssen einen Verweis auf `System.Speech.dll` hinzufügen, bevor Sie die Sprachklassen verwenden können.

# Examples

## Spracherkennung für das Freitextdiktat asynchron

```
using System.Speech.Recognition;

// ...

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

## Asynchrones Erkennen von Sprache basierend auf einem eingeschränkten Satz von Phrasen

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

SpeechRecognitionEngine-Klasse zum Erkennen von Sprache online lesen: https://riptutorial.com/de/dot-net/topic/69/speechrecognitionengine-klasse-zum-erkennen-von-sprache

# Kapitel 40: Speicherverwaltung

## Bemerkungen

Leistungskritische Anwendungen in verwalteten .NET-Anwendungen können durch den GC stark beeinträchtigt werden. Wenn der GC ausgeführt wird, werden alle anderen Threads bis zum Abschluss angehalten. Aus diesem Grund wird empfohlen, die GC-Prozesse sorgfältig zu bewerten und festzulegen, wie sie bei der Ausführung minimiert werden sollen.

## Examples

### Nicht verwaltete Ressourcen

Wenn wir über die GC und den "Heap" sprechen, sprechen wir wirklich über den sogenannten *verwalteten Heap* . Objekte auf dem *verwalteten Heap* können auf Ressourcen zugreifen, die sich nicht auf dem verwalteten Heap befinden, z. B. beim Schreiben oder Lesen aus einer Datei. Unerwartetes Verhalten kann auftreten, wenn eine Datei zum Lesen geöffnet wird und dann eine Ausnahmebedingung auftritt, die das Schließen des Datei-Handles verhindert. Aus diesem Grund erfordert .NET, dass nicht verwaltete Ressourcen die `IDisposable` Schnittstelle implementieren. Diese Schnittstelle verfügt über eine einzige Methode namens `Dispose` ohne Parameter:

```
public interface IDisposable
{
    Dispose();
}
```

Beim Umgang mit nicht verwalteten Ressourcen sollten Sie sicherstellen, dass sie ordnungsgemäß entsorgt werden. Sie können dies tun, indem Sie `Dispose()` explizit in einem `finally` Block oder mit einer `using` Anweisung aufrufen.

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

oder

```
string textFromFile;
string filename = "SomeFile.txt";
```

```
using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

Letzteres ist die bevorzugte Methode und wird beim Kompilieren automatisch auf das Erstere erweitert.

## Verwenden Sie SafeHandle, wenn Sie nicht verwaltete Ressourcen umschließen

Wenn Sie Wrapper für nicht verwaltete Ressourcen schreiben, sollten Sie `SafeHandle` eher als Unterklasse `SafeHandle` anstatt `IDisposable` und einen Finalizer selbst zu implementieren. Ihre `SafeHandle` Unterklasse sollte so klein und einfach wie möglich sein, um die `SafeHandle` zu minimieren. Dies bedeutet wahrscheinlich, dass Ihre SafeHandle-Implementierung ein internes Implementierungsdetail einer Klasse enthält, das diese umschließt, um eine verwendbare API bereitzustellen. Diese Klasse stellt sicher, dass Ihr nicht verwalteter Handle freigegeben wird, auch wenn ein Programm Ihre `SafeHandle` Instanz `SafeHandle` .

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Haftungsausschluss: In diesem Beispiel wird gezeigt, wie eine verwaltete Ressource mit `SafeHandle` die `IDisposable` für Sie implementiert und die Finalizer entsprechend konfiguriert. Es ist sehr verständlich und wahrscheinlich sinnlos, auf diese Weise einen Speicherplatz zuzuweisen.

Speicherverwaltung online lesen: https://riptutorial.com/de/dot-net/topic/59/speicherverwaltung

# Kapitel 41: Stapel und Haufen

## Bemerkungen

Es ist erwähnenswert, dass bei der Deklaration eines Referenztyps der Anfangswert `null` . Dies liegt daran, dass es noch nicht auf einen Speicherort im Speicher verweist und ein einwandfreier Zustand ist.
Mit Ausnahme von nullfähigen Typen müssen Wertetypen jedoch normalerweise immer einen Wert haben.

## Examples

### Verwendete Werttypen

Werttypen enthalten einfach einen ***Wert*** .

Alle Werttypen werden von der System.ValueType- Klasse abgeleitet. Dazu gehören die meisten integrierten Typen.

Beim Erstellen eines neuen Werttyps wird der als ***Stack*** bezeichnete Speicherbereich verwendet. Der Stack wächst entsprechend um den deklarierten Typ. So wird zum Beispiel einem int immer 32 Bit Speicher auf dem Stack zugewiesen. Wenn der Werttyp nicht mehr im Gültigkeitsbereich liegt, wird der Speicherplatz auf dem Stapel freigegeben.

Der folgende Code veranschaulicht einen Werttyp, der einer neuen Variablen zugewiesen wird. Eine Struktur wird als praktische Methode zum Erstellen eines benutzerdefinierten Werttyps verwendet (die System.ValueType-Klasse kann nicht anderweitig erweitert werden).

Es ist wichtig zu verstehen, dass bei der Zuweisung eines Werttyps der Wert selbst in die neue Variable ***kopiert*** wird, was bedeutet, dass wir zwei verschiedene Instanzen des Objekts haben, die sich nicht gegenseitig beeinflussen können.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";
```

```
        Console.WriteLine(                // Outputs 'False' - because
            object.ReferenceEquals(       // personA and personB are referencing
                personA,                  // different areas of memory
                personB));

        Console.WriteLine(personA.Name);  // Outputs 'Linda'
        Console.WriteLine(personB.Name);  // Outputs 'Bob'
    }
}
```

## Verwendete Referenztypen

Referenztypen umfassen sowohl einen ***Verweis*** auf einen Speicherbereich als auch einen in diesem Bereich gespeicherten ***Wert*** .
Dies ist analog zu Zeigern in C / C ++.

Alle Referenztypen werden auf dem sogenannten ***Heap*** gespeichert.
Der Heap ist einfach ein verwalteter Speicherbereich, in dem Objekte gespeichert werden. Wenn ein neues Objekt instanziiert wird, wird ein Teil des Heaps für die Verwendung durch dieses Objekt zugewiesen und ein Verweis auf diese Position des Heaps wird zurückgegeben. Der Heap wird vom *Garbage Collector* verwaltet und gewartet und erlaubt keinen manuellen Eingriff.

Zusätzlich zu dem für die Instanz selbst erforderlichen Speicherplatz ist zusätzlicher Speicherplatz zum Speichern der Referenz selbst sowie zusätzliche temporäre Informationen erforderlich, die für die .NET CLR erforderlich sind.

Der folgende Code veranschaulicht, wie ein Referenztyp einer neuen Variablen zugewiesen wird. In diesem Fall verwenden wir eine Klasse. Alle Klassen sind Referenztypen (auch wenn sie statisch sind).

Wenn ein Referenztyp einer anderen Variablen zugewiesen wird, ist es die ***Referenz*** auf das Objekt, die kopiert wird, **nicht** der Wert selbst. Dies ist ein wichtiger Unterschied zwischen Werttypen und Referenztypen.

Dies hat zur Folge, dass wir jetzt *zwei* Verweise auf dasselbe Objekt haben.
Alle Änderungen an den Werten innerhalb dieses Objekts werden von beiden Variablen übernommen.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;
```

```
        personA.Name = "Linda";

        Console.WriteLine(               // Outputs 'True' - because
            object.ReferenceEquals(      // personA and personB are referencing
                personA,                 // the *same* memory location
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

Stapel und Haufen online lesen: https://riptutorial.com/de/dot-net/topic/9358/stapel-und-haufen

# Kapitel 42: Synchronisierungskontexte

## Bemerkungen

Ein Synchronisierungskontext ist eine Abstraktion, die es dem Benutzer ermöglicht, Code zu verwenden, um Arbeitseinheiten an einen Scheduler zu übergeben, ohne sich dessen bewusst zu sein, wie die Arbeit geplant wird.

Synchronisierungskontexte werden normalerweise verwendet, um sicherzustellen, dass Code in einem bestimmten Thread ausgeführt wird. In WPF- und Winforms-Anwendungen wird vom Präsentations-Framework ein `SynchronizationContext` bereitgestellt, der den UI-Thread darstellt. Auf diese Weise kann `SynchronizationContext` als Erzeuger-Konsument-Muster für Delegierte betrachtet werden. Ein Arbeiter - Thread *erzeugen* , ausführbaren Code (der Delegierten) und oder den *Verbrauch* von der UI Nachrichtenschleife Warteschlange.

Die Task Parallel Library bietet Funktionen zum automatischen Erfassen und Verwenden von Synchronisationskontexten.

## Examples

### Führen Sie den Code im UI-Thread aus, nachdem Sie die Hintergrundarbeit ausgeführt haben

In diesem Beispiel wird veranschaulicht, wie eine UI-Komponente mithilfe eines `SynchronizationContext` von einem Hintergrund-Thread aktualisiert wird

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

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

Wenn Sie in diesem Beispiel versucht haben, `MyTextBox.Text` direkt in der `for` Schleife zu aktualisieren, erhalten Sie einen Thread-Fehler. Durch das Posten der `UpdateCallback` Aktion im `SynchronizationContext` wird das Textfeld in demselben Thread wie der Rest der Benutzeroberfläche aktualisiert.

In der Praxis sollten Fortschrittsaktualisierungen mithilfe einer Instanz von `System.IProgress<T>` . Die Standardimplementierung `System.Progress<T>` erfasst automatisch den Synchronisationskontext, für den sie erstellt wird.

Synchronisierungskontexte online lesen: https://riptutorial.com/de/dot-net/topic/5407/synchronisierungskontexte

# Kapitel 43: System.IO

## Examples

### Lesen einer Textdatei mit StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Beachten Sie, dass diese Überladung des `StreamReader` Konstruktors eine automatische Codierungserkennung durchführt , die möglicherweise der in der Datei verwendeten Codierung entspricht.

Beachten Sie, dass es einige bequeme Methoden gibt, die den gesamten Text aus der Datei lesen, die in der `System.IO.File` Klasse verfügbar ist, nämlich `File.ReadAllText(path)` und `File.ReadAllLines(path)` .

### Lesen / Schreiben von Daten mit System.IO.File

Lassen Sie uns zunächst drei verschiedene Möglichkeiten zum Extrahieren von Daten aus einer Datei betrachten.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- In der ersten Zeile lesen wir alle Daten in der Datei als Zeichenfolge.
- In der zweiten Zeile lesen wir die Daten in der Datei in ein String-Array. Jede Zeile in der Datei wird zu einem Element im Array.
- Beim dritten lesen wir die Bytes aus der Datei.

---

Lassen Sie uns als Nächstes drei verschiedene Methoden zum **Anhängen von** Daten an eine Datei anzeigen. Wenn die von Ihnen angegebene Datei nicht vorhanden ist, erstellt jede Methode die Datei automatisch, bevor sie die Daten anfügt.

```
 File.AppendAllText(file, "Here is some data that is\nappended to the file.");
 File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the
file." });
 using (StreamWriter stream = File.AppendText(file))
 {
     stream.WriteLine("Here is some data that is");
     stream.Write("appended to the file.");
```

```
}
```

- In der ersten Zeile fügen wir einfach eine Zeichenkette am Ende der angegebenen Datei hinzu.
- In der zweiten Zeile fügen wir jedes Element des Arrays in eine neue Zeile in der Datei ein.
- In der dritten Zeile schließlich verwenden wir `File.AppendText` , um einen `File.AppendText` zu öffnen, der die angehängten Daten anfügt.

---

Und schließlich wollen wir drei verschiedene Methoden zum **Schreiben von** Daten in eine Datei sehen. Der Unterschied zwischen *anhängen* und das *Schreiben* ist , dass das Schreiben der Daten in der Datei **überschreibt** , während **fügt** in der Datei mit den Daten angehängt wird . Wenn die von Ihnen angegebene Datei nicht vorhanden ist, erstellt jede Methode die Datei automatisch, bevor sie die Daten in die Datei schreibt.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- Die erste Zeile schreibt eine Zeichenfolge in die Datei.
- Die zweite Zeile schreibt jeden String im Array in eine eigene Zeile in der Datei.
- In der dritten Zeile können Sie ein Byte-Array in die Datei schreiben.

### Serielle Ports mit System.IO.SerialPorts

## Iteration über angeschlossene serielle Ports

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

---

## Instantiieren eines System.IO.SerialPort-Objekts

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

**HINWEIS** : Dies sind nur drei der sieben Überladungen des Konstruktors für den SerialPort-Typ.

---

## Daten über den SerialPort lesen / schreiben

Der einfachste Weg ist die Verwendung der Methoden `SerialPort.Read` und `SerialPort.Write` . Sie

können jedoch auch ein `System.IO.Stream` Objekt `System.IO.Stream` , mit dem Sie Daten über den SerialPort streamen können. Verwenden Sie dazu `SerialPort.BaseStream` .

### lesen

```
int length = port.BytesToRead;
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

Sie können auch alle verfügbaren Daten lesen:

```
string curData = port.ReadExisting();
```

Oder lesen Sie einfach den ersten Zeilenumbruch in den eingehenden Daten:

```
string line = port.ReadLine();
```

### Schreiben

Der einfachste Weg, Daten über den SerialPort zu schreiben, ist:

```
port.Write("here is some text to be sent over the serial port.");
```

Sie können jedoch bei Bedarf auch Daten wie folgt senden:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

System.IO online lesen: https://riptutorial.com/de/dot-net/topic/5259/system-io

# Kapitel 44: System.IO.File-Klasse

## Syntax

- String-Quelle;
- String Ziel;

## Parameter

| Parameter | Einzelheiten |
|---|---|
| `source` | Die Datei, die an einen anderen Speicherort verschoben werden soll. |
| `destination` | Das Verzeichnis, in das Sie `source` verschieben möchten (diese Variable sollte auch den Namen (und die Dateierweiterung) der Datei enthalten |

## Examples

### Datei löschen

So löschen Sie eine Datei (wenn Sie über die erforderlichen Berechtigungen verfügen) ist so einfach wie folgt:

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

Es können jedoch viele Dinge schief gehen:

- Sie haben keine erforderlichen Berechtigungen ( `UnauthorizedAccessException` wird ausgelöst).
- Die Datei wird möglicherweise von einer anderen Person verwendet ( `IOException` wird ausgelöst).
- Datei kann nicht gelöscht werden, da ein Fehler auf niedriger Ebene vorliegt oder das Medium `IOException` ist ( `IOException` wird ausgelöst).
- Datei existiert nicht mehr ( `IOException` wird ausgelöst).

Beachten Sie, dass der letzte Punkt (Datei nicht vorhanden) normalerweise mit einem Code-Snippet wie folgt *umgangen wird* :

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

Es handelt sich jedoch nicht um eine atomare Operation, und die Datei kann von einer anderen Person zwischen dem Aufruf von `File.Exists()` und vor `File.Delete()` . Der richtige Ansatz für die Verarbeitung von E / A-Vorgängen erfordert die Ausnahmebehandlung (vorausgesetzt, bei einem

Fehlschlagen des Vorgangs kann ein alternativer Ablauf von Maßnahmen ergriffen werden):

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

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

Beachten Sie, dass diese E / A-Fehler manchmal vorübergehend sind (z. B. verwendete Datei). Wenn eine Netzwerkverbindung beteiligt ist, wird sie möglicherweise automatisch wiederhergestellt, ohne dass von unserer Seite etwas unternommen wird. Es ist üblich, eine E / A-Operation einige Male mit einer kleinen Verzögerung zwischen den einzelnen Versuchen zu *wiederholen* :

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

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

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

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

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

Hinweis: Wenn Sie diese Funktion aufrufen, wird die Datei in der Windows-Umgebung nicht wirklich gelöscht. Wenn eine andere Person die Datei mit `FileShare.Delete` öffnet, kann die Datei gelöscht werden. `FileShare.Delete` geschieht jedoch nur, wenn der Eigentümer die Datei schließt.

### Entfernen Sie unerwünschte Zeilen aus einer Textdatei

Eine Textdatei zu ändern ist nicht einfach, da der Inhalt verschoben werden muss. Für *kleine* Dateien ist die einfachste Methode, den Inhalt in den Speicher zu lesen und dann den geänderten Text zurückzuschreiben.

In diesem Beispiel lesen wir alle Zeilen aus einer Datei und löschen alle leeren Zeilen und schreiben dann in den ursprünglichen Pfad zurück:

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

Wenn die Datei zu groß ist, um sie in den Speicher zu laden, unterscheidet sich der Ausgabepfad vom Eingabepfad:

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

### Konvertieren Sie die Kodierung der Textdatei

Text wird verschlüsselt gespeichert (siehe auch Strings- Thema). In einigen Fällen müssen Sie möglicherweise die Codierung ändern. In diesem Beispiel wird (zur Vereinfachung) davon ausgegangen, dass die Datei nicht zu groß ist und vollständig im Speicher gelesen werden kann:

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

Vergessen Sie bei Konvertierungen nicht, dass die Datei möglicherweise BOM (Byte Order Mark) enthält, um besser zu verstehen, wie sie verwaltet wird, lesen Sie Encoding.UTF8.GetString berücksichtigt nicht die Präambel / BOM .

### "Berühren" Sie eine große Anzahl von Dateien (um die letzte Schreibzeit zu aktualisieren)

In diesem Beispiel wird das letzte Schreiben einer großen Anzahl von Dateien `System.IO.Directory.EnumerateFiles` (mithilfe von `System.IO.Directory.EnumerateFiles` anstelle von `System.IO.Directory.GetFiles()` ). Optional können Sie ein Suchmuster angeben (standardmäßig `"*.*"` Und schließlich eine Verzeichnisstruktur durchsuchen (nicht nur das angegebene Verzeichnis):

```
public static void Touch(string path,
                         string searchPattern = "*.*",
```

```
                        SearchOptions options = SearchOptions.None)
{
    var now = DateTime.Now;

    foreach (var filePath in Directory.EnumerateFiles(path, searchPattern, options))
    {
        File.SetLastWriteTime(filePath, now);
    }
}
```

### Aufzählen von Dateien, die älter als eine angegebene Anzahl sind

Dieses Snippet ist eine Hilfsfunktion zum Auflisten aller Dateien, die älter als ein bestimmtes Alter sind. Dies ist beispielsweise nützlich, wenn Sie alte Protokolldateien oder alte zwischengespeicherte Daten löschen müssen.

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

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

So benutzt:

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

Einige Dinge zu beachten:

- Die Suche wird mit `Directory.EnumerateFiles()` anstelle von `Directory.GetFiles()` . Die Aufzählung ist *lebendig,* dann müssen Sie nicht warten, bis alle Dateisystemeinträge abgerufen wurden.
- Wir prüfen die letzte Schreibzeit, aber Sie können die Erstellungszeit oder die letzte Zugriffszeit verwenden (z. B. zum Löschen *nicht verwendeter* zwischengespeicherter Dateien, dass die Zugriffszeit deaktiviert ist).
- Die Granularität ist für alle diese Eigenschaften nicht einheitlich (Schreibzeit, Zugriffszeit, Erstellungszeit). Weitere Informationen hierzu finden Sie in MSDN.

### Verschieben Sie eine Datei von einem Ort an einen anderen

## File.Move

Um eine Datei von einem Ort an einen anderen zu verschieben, kann dies mit einer einfachen Codezeile erreicht werden:

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

Es gibt jedoch viele Dinge, die bei dieser einfachen Operation schief gehen könnten. Was passiert beispielsweise, wenn der Benutzer, der Ihr Programm ausführt, kein Laufwerk mit der Bezeichnung 'C' hat? Was wäre, wenn sie es taten - aber sie beschlossen, es in 'B' oder 'M' umzubenennen?

Was ist, wenn die Quelldatei (die Datei, in die Sie verschieben möchten) ohne Ihr Wissen verschoben wurde - oder wenn es einfach nicht existiert?

Dies kann umgangen werden, indem zunächst geprüft wird, ob die Quelldatei vorhanden ist:

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

Dadurch wird sichergestellt, dass die Datei zu diesem Zeitpunkt existiert und an einen anderen Ort verschoben werden kann. Es kann vorkommen, dass ein einfacher Aufruf von `File.Exists` nicht ausreicht. Ist dies nicht der Fall, überprüfen Sie erneut, teilen Sie dem Benutzer mit, dass die Operation fehlgeschlagen ist, oder behandeln Sie die Ausnahme.

Eine `FileNotFoundException` ist nicht die einzige Ausnahme, der Sie wahrscheinlich begegnen.

Nachfolgend finden Sie mögliche Ausnahmen:

| Ausnahmetyp | Beschreibung |
|---|---|
| `IOException` | Die Datei ist bereits vorhanden oder die Quelldatei wurde nicht gefunden. |
| `ArgumentNullException` | Der Wert der Source- und / oder Destination-Parameter ist null. |
| `ArgumentException` | Der Wert der Source- und / oder Destination-Parameter ist leer oder enthält ungültige Zeichen. |
| `UnauthorizedAccessException` | Sie haben nicht die erforderlichen Berechtigungen, um diese Aktion auszuführen. |
| `PathTooLongException` | Quelle, Ziel oder angegebene Pfade überschreiten die maximale Länge. Unter Windows muss die Länge eines Pfads weniger als 248 Zeichen betragen, während Dateinamen weniger als 260 Zeichen umfassen dürfen. |
| `DirectoryNotFoundException` | Das angegebene Verzeichnis wurde nicht gefunden. |
| `NotSupportedException` | Die Quell- oder Zielpfade oder Dateinamen weisen ein ungültiges Format auf. |

System.IO.File-Klasse online lesen: https://riptutorial.com/de/dot-net/topic/5395/system-io-file-klasse

# Kapitel 45: System.Net.Mail

## Bemerkungen

Es ist wichtig, eine System.Net.MailMessage zu entsorgen, da jede einzelne Anlage einen Stream enthält und diese Streams so schnell wie möglich freigegeben werden müssen. Die using-Anweisung stellt sicher, dass das Disposable-Objekt auch bei Ausnahmen verfügbar ist

## Examples

### MailMessage

Hier ist das Beispiel für das Erstellen einer E-Mail-Nachricht mit Anhängen. Nach dem Erstellen senden wir diese Nachricht mit Hilfe der `SmtpClient` Klasse. Hier wird der Standardport 25 verwendet.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string>
Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;
```

```
                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
            }
        }
        catch
        {
            return false;
        }
    }
}
```

## Mail mit Anhang

`MailMessage` stellt eine E-Mail-Nachricht dar, die mithilfe der `SmtpClient` Klasse weiter gesendet werden `SmtpClient` . Der E-Mail-Nachricht können mehrere Anhänge (Dateien) hinzugefügt werden.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

System.Net.Mail online lesen: https://riptutorial.com/de/dot-net/topic/7440/system-net-mail

# Kapitel 46: System.Reflection.Emit-Namespace

## Examples

### Eine Assembly dynamisch erstellen

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
```

```
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);

        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance
method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as
the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);
```

```
        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another
constructor in the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save
the assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can
use e.g. ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

System.Reflection.Emit-Namespace online lesen: https://riptutorial.com/de/dot-net/topic/74/system-reflection-emit-namespace

# Kapitel 47: System.Runtime.Caching.MemoryCache (ObjectCache)

## Examples

### Element zum Cache hinzufügen (Set)

Die Funktion set fügt einen Cache-Eintrag in den Cache ein, indem sie eine CacheItem-Instanz verwendet, um den Schlüssel und den Wert für den Cache-Eintrag bereitzustellen.

Diese Funktion `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

### System.Runtime.Caching.MemoryCache (ObjectCache)

Diese Funktion ruft den Formularcache für vorhandene Elemente ab. Wenn das Element nicht im Cache vorhanden ist, ruft es das Element basierend auf der Funktion valueFetchFactory ab.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

            //returns existing item form cache or add the new value if it does not exist.
```

```
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
            return default(TValue);
        }
    }
```

System.Runtime.Caching.MemoryCache (ObjectCache) online lesen:
https://riptutorial.com/de/dot-net/topic/76/system-runtime-caching-memorycache--objectcache-

# Kapitel 48: Systemdiagnose

## Examples

### Stoppuhr

Dieses Beispiel zeigt, wie `Stopwatch` zum Benchmarking eines Codeblocks verwendet werden kann.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

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

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

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

### Führen Sie Shell-Befehle aus

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

Hiermit wird das Cmd-Fenster ausgeblendet.

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

## Befehl an CMD senden und Ausgabe empfangen

Diese Methode ermöglicht das Senden eines `command` an `Cmd.exe` und gibt die Standardausgabe (einschließlich Standardfehler) als Zeichenfolge zurück:

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

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

    var p = new Process {StartInfo = startInfo};

    p.Start();

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

### Verwendungszweck

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

### Ausgabe

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME ANMELDUNGSZEIT Joe.Bloggs ica-cgp # 0 2 Active 24692 + 13: 29 25/07/2016 07:50 Jim.McFlannegan ica-cgp # 1 3 Active. 25/07 / 2016 08:33 Andy.McAnderson ica-cgp # 2 4 Aktiv. 25/07/2016 08:54 John.Smith ica-cgp # 4 5 Active 14 25/07/2016 08:57 Bob.Bobbington ica-cgp # 5 6 Active 24692 + 13: 29 25/07/2016 09:05 Tim.Tom ica-cgp # 6 7 Active. 25/07/2016 09:08 Bob.Joges ica-cgp # 7 8 Active 24692 + 13: 29 25 / 07/2016 09:13 "

---

In einigen Fällen kann der Zugriff auf den betreffenden Server auf bestimmte Benutzer beschränkt sein. Wenn Sie die Anmeldeinformationen für diesen Benutzer haben, können Sie Abfragen mit dieser Methode senden:

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

```
{
    var cmdOut = string.Empty;

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

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

    var p = new Process {StartInfo = startInfo};

    p.Start();

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

Passwort erhalten:

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

    return ss;
}
```

### Anmerkungen

Beide oben genannten Methoden geben eine Verkettung von STDOUT und STDERR zurück, da `OutputDataReceived` und `ErrorDataReceived` beide an dieselbe Variable - `cmdOut` - `cmdOut` .

Systemdiagnose online lesen: https://riptutorial.com/de/dot-net/topic/3143/systemdiagnose

# Kapitel 49: Task Parallel Library (TPL)

## Bemerkungen

# Zweck und Anwendungsfälle

Zweck der Task Parallel Library ist die Vereinfachung des Schreibens und Verwaltens von Multithread-Code und parallelem Code.

Einige Anwendungsfälle *:

- Beibehalten einer Benutzeroberfläche, indem Hintergrundarbeit für separate Aufgaben ausgeführt wird
- Verteilung der Arbeitslast
- Zulassen, dass eine Clientanwendung gleichzeitig Anforderungen sendet und empfängt (rest, TCP / UDP, ect)
- Mehrere Dateien gleichzeitig lesen und / oder schreiben

* Code sollte von Fall zu Fall für Multithreading betrachtet werden. Wenn zum Beispiel eine Schleife nur wenige Iterationen durchführt oder nur einen kleinen Teil der Arbeit erledigt, kann der Mehraufwand für Parallelität die Vorteile überwiegen.

**TPL mit .Net 3.5**

Die TPL ist auch für .Net 3.5 in einem NuGet-Paket verfügbar. Sie wird Task Parallel Library genannt.

## Examples

### Grundlegende Producer-Consumer-Schleife (BlockingCollection)

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

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

Es ist erwähnenswert, wenn Sie `collection.CompleteAdding();` nicht aufrufen.CompleteAdding `collection.CompleteAdding();` Sie können der Sammlung auch dann weiter hinzufügen, wenn Ihre

Consumer-Aufgabe ausgeführt wird. Rufen Sie einfach `collection.CompleteAdding();` Wenn Sie sicher sind, gibt es keine weiteren Ergänzungen. Diese Funktion kann verwendet werden, um einen Multiple Producer zu einem einzigen Consumer-Muster zu machen, bei dem mehrere Quellen Elemente in die BlockingCollection einspeisen und ein einzelner Consumer Elemente herauszieht und mit diesen etwas tut. Wenn Ihre BlockingCollection leer ist, bevor Sie das vollständige Hinzufügen aufrufen, wird die Enumerable from `collection.GetConsumingEnumerable()` so lange blockiert, bis der Collection ein neues Element hinzugefügt wird. wird aufgerufen und die Warteschlange ist leer.

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

Task.WaitAll(producerTask, consumerTask);

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

## Aufgabe: Grundinstanziierung und Warten

Eine Aufgabe kann erstellt werden, indem die `Task` Klasse direkt instanziiert wird ...

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

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

... oder mit der statischen `Task.Run` Methode:

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

Beachten Sie, dass nur im ersten Fall `Start` explizit aufgerufen werden muss.

## Aufgabe: WaitAll und Variablenerfassung

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

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

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

## Aufgabe: WaitAny

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

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

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

Task.WaitAll(allTasks);
```

**Hinweis:** Die letzte `WaitAll` ist notwendig, da `WaitAny` keine Ausnahmen verursacht.

## Task: Ausnahmen behandeln (mit Wait)

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

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

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

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

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

## Aufgabe: Behandlung von Ausnahmen (ohne zu warten)

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

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

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

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

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

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

## Aufgabe: Abbrechen mit Annullierungstoken

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

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

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

cancellationTokenSource.Cancel();
try
{
```

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

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

Als Alternative zu `ThrowIfCancellationRequested` kann die Stornierungsanforderung mit `IsCancellationRequested` erkannt und eine `OperationCanceledException` manuell ausgelöst werden:

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

Beachten Sie, wie das Annullierungstoken im Parameter `cancellationToken` an den `cancellationToken` wird. Dies ist erforderlich, damit die Task in den Status `Canceled` `Faulted` , nicht in den `Faulted` , wenn `ThrowIfCancellationRequested` aufgerufen wird. Aus demselben Grund wird das Annullierungstoken im zweiten Fall explizit im Konstruktor von `OperationCanceledException` .

## Task.WhenAny

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

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

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

## Task.WhenAll

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

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

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

## Parallel.Einruf

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

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

## Parallel.ForEach

In diesem Beispiel wird `Parallel.ForEach` , um die Summe der Zahlen zwischen 1 und 10000 mithilfe mehrerer Threads zu berechnen. Um Thread-Sicherheit zu erreichen, wird `Interlocked.Add` verwendet, um die Zahlen zu summieren.

```
using System.Threading;

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

## Parallel.für

In diesem Beispiel wird `Parallel.For` , um die Summe der Zahlen zwischen 1 und 10000 mithilfe mehrerer Threads zu berechnen. Um Thread-Sicherheit zu erreichen, wird `Interlocked.Add` verwendet, um die Zahlen zu summieren.

```
using System.Threading;

int Foo()
```

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

## Fließender Ausführungskontext mit AsyncLocal

Wenn Sie einige Daten von der übergeordneten Aufgabe an ihre `AsyncLocal` Aufgaben übergeben müssen, damit sie bei der Ausführung logisch übertragen werden, verwenden Sie die `AsyncLocal` Klasse :

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

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

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

        user.Value = "user from task1";

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

    task1.Wait();

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

**Anmerkung:** Wie aus dem obigen Beispiel `AsynLocal.Value` hat `AsynLocal.Value` eine `copy on read` Semantik. Wenn Sie jedoch einen Referenztyp übergeben und seine Eigenschaften ändern, wirken sich dies auf andere Aufgaben aus. Daher `AsyncLocal` es sich bei `AsyncLocal` oder unveränderliche Typen zu verwenden.

## Parallel.ForEach in VB.NET

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

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

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

## Aufgabe: Rückgabe eines Wertes

Eine Task, die einen Wert `Task< TResult >` hat den Rückgabetyp `Task< TResult >` wobei TResult der Wertetyp ist, der zurückgegeben werden muss. Sie können das Ergebnis einer Aufgabe über ihre Result-Eigenschaft abfragen.

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

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

        return sum;
    });

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

Wenn die Task asynchron ausgeführt wird und die Task wartet, wird das Ergebnis zurückgegeben.

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

Task Parallel Library (TPL) online lesen: https://riptutorial.com/de/dot-net/topic/55/task-parallel-library--tpl-

# Kapitel 50: TPL-API-Übersichten (Task Parallel Library)

## Bemerkungen

Die Task Parallel Library umfasst eine Reihe öffentlicher Typen und APIs, die das Hinzufügen von Parallelität und Parallelität zu einer Anwendung erheblich vereinfachen. .Netz. TPL wurde in .Net 4 eingeführt und ist die empfohlene Methode zum Schreiben von Multithreading- und Parallelcode.

TPL kümmert sich um Arbeitsplanung, Thread-Affinität, Löschungsunterstützung, Statusverwaltung und Lastverteilung, sodass der Programmierer sich auf die Lösung von Problemen konzentrieren kann, anstatt sich mit den üblichen Details auf niedriger Ebene zu beschäftigen.

## Examples

### Führen Sie die Arbeit als Reaktion auf einen Schaltflächenklick aus und aktualisieren Sie die Benutzeroberfläche

In diesem Beispiel wird veranschaulicht, wie Sie auf einen Schaltflächenklick reagieren können, indem Sie einige Arbeiten an einem Arbeitsthread ausführen und anschließend die Benutzeroberfläche aktualisieren, um den Abschluss anzuzeigen

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the
UI thread.

}
```

TPL-API-Übersichten (Task Parallel Library) online lesen: https://riptutorial.com/de/dot-net/topic/5164/tpl-api-ubersichten--task-parallel-library-

# Kapitel 51: TPL-Datenfluss

## Bemerkungen

# In den Beispielen verwendete Bibliotheken

`System.Threading.Tasks.Dataflow`

`System.Threading.Tasks`

`System.Net.Http`

`System.Net`

# Unterschied zwischen Post und SendAsync

Um einem Block Elemente hinzuzufügen, können Sie entweder `Post` oder `SendAsync` .

`Post` wird versuchen, den Artikel synchron hinzuzufügen und einen `bool` zurückzusenden, der `bool` , ob er erfolgreich war oder nicht. Es kann nicht erfolgreich sein, wenn zum Beispiel ein Block seine `BoundedCapcity` erreicht `BoundedCapcity` und noch keinen Platz für neue Elemente hat. `SendAsync` hingegen gibt eine nicht abgeschlossene `Task<bool>` , auf die Sie `await` . Diese Aufgabe wird in der Zukunft mit einem `true` Ergebnis abgeschlossen, wenn der Block seine interne Warteschlange gelöscht hat, und er kann mehr Elemente annehmen oder ein `false` Ergebnis, wenn er dauerhaft abnimmt (z. B. als Ergebnis einer Stornierung).

## Examples

### In einem ActionBlock buchen und auf Abschluss warten

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

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

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

### Blöcke verknüpfen, um eine Pipeline zu erstellen

```
var httpClient = new HttpClient();

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

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

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

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

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

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

## Synchroner Producer / Consumer mit BufferBlock

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

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

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

```
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
        {
            var consumer = new Consumer();
            while(true)
            {
                consumer.Consume(buffer.Receive());
            }
        });

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

## Asynchroner Producer-Consumer mit begrenztem Pufferblock

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

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

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

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

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

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

TPL-Datenfluss online lesen: https://riptutorial.com/de/dot-net/topic/784/tpl-datenfluss

# Kapitel 52: Unit-Tests

## Examples

### Hinzufügen eines MSTest-Einheitentestprojekts zu einer vorhandenen Lösung

- Klicken Sie mit der rechten Maustaste auf die Lösung, Neues Projekt hinzufügen
- Wählen Sie im Testbereich ein Unit-Test-Projekt aus
- Wählen Sie einen Namen für die Assembly. Wenn Sie das Projekt `Foo` testen, kann der Name `Foo.Tests`
- Fügen Sie einen Verweis auf das getestete Projekt in den Verweisen auf Unit-Test-Projekte hinzu

### Erstellen einer Beispieltestmethode

Für MSTest (das Standardtest-Framework) müssen Sie Ihre `[TestClass]` mit einem Attribut `[TestClass]` und die Testmethoden mit einem Attribut `[TestMethod]` und öffentlich sein.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

Unit-Tests online lesen: https://riptutorial.com/de/dot-net/topic/5365/unit-tests

# Kapitel 53: VB-Formulare

## Examples

### Hallo Welt in VB.NET-Formularen

So zeigen Sie ein Meldungsfeld an, wenn das Formular angezeigt wurde:

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

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

Load () wird zuerst und nur einmal aufgerufen, wenn das Formular zum ersten Mal geladen wird. Show () wird jedes Mal aufgerufen, wenn der Benutzer das Formular startet. Activate () wird jedes Mal aufgerufen, wenn der Benutzer das Formular aktiviert.

Load () wird ausgeführt, bevor Show () aufgerufen wird, aber seien Sie gewarnt: Wenn Sie msgBox () in show aufrufen, kann msgBox () ausgeführt werden, bevor Load () beendet ist. **Es ist im Allgemeinen eine schlechte Idee, sich auf die Reihenfolge der Ereignisse zwischen Load (), Show () und ähnlichem zu verlassen.**

### Für Anfänger

Einige Dinge, die alle Anfänger wissen / tun sollten, damit sie einen guten Start mit VB .Net haben:

Stellen Sie die folgenden Optionen ein:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Verwenden Sie &, nicht + für die Verkettung von Zeichenfolgen. Strings sollten etwas genauer untersucht werden, da sie häufig verwendet werden.

Verbringen Sie etwas Zeit, um die Wert- und Referenztypen zu verstehen.

Verwenden Sie niemals Application.DoEvents . Achten Sie auf die 'Vorsicht'. Wenn Sie an einem Punkt angelangt sind, an dem dies etwas zu sein scheint, fragen Sie nach.

Die Dokumentation ist dein Freund.

### Forms Timer

Die Windows.Forms.Timer- Komponente kann verwendet werden, um Benutzerinformationen bereitzustellen, die **nicht** zeitkritisch sind. Erstellen Sie ein Formular mit einer Schaltfläche, einer Beschriftung und einer Timer-Komponente.

Es könnte beispielsweise verwendet werden, um dem Benutzer die Uhrzeit periodisch anzuzeigen.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

Dieser Timer ist jedoch nicht für das Timing geeignet. Ein Beispiel wäre die Verwendung für einen Countdown. In diesem Beispiel simulieren wir einen Countdown von drei Minuten. Dies kann durchaus eines der langweiligsten Beispiele sein.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
```

```
    End Sub

    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Nach dem Anklicken von button1 vergehen ungefähr drei Minuten, und label1 zeigt die Ergebnisse. Zeigt label1 180? Wahrscheinlich nicht. Auf meiner Maschine zeigte es 182,5!

Der Grund für die Diskrepanz liegt in der Dokumentation: "Die Windows Forms-Timerkomponente ist Single-Threading und auf eine Genauigkeit von 55 Millisekunden beschränkt." Deshalb sollte es nicht für das Timing verwendet werden.

Wenn Sie den Timer und die Stoppuhr etwas anders verwenden, können wir bessere Ergebnisse erzielen.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
```

```
End Class
```

Es gibt andere Timer, die bei Bedarf verwendet werden können. Diese Suche sollte in dieser Hinsicht helfen.

VB-Formulare online lesen: https://riptutorial.com/de/dot-net/topic/2197/vb-formulare

# Kapitel 54: Vernetzung

## Bemerkungen

Siehe auch: HTTP-Clients

## Examples

### Grundlegender TCP-Chat (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
```

```
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

## Einfacher SNTP-Client (UdpClient)

Weitere Informationen zum SNTP-Protokoll finden Sie unter RFC 2030 .

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from
UTC");
```

```
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)

        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

Vernetzung online lesen: https://riptutorial.com/de/dot-net/topic/35/vernetzung

# Kapitel 55: Verschlüsselung / Kryptographie

## Bemerkungen

.NET Framework bietet die Implementierung vieler kryptografischer Algorithmen. Sie umfassen im Wesentlichen symmetrische Algorithmen, asymmetrische Algorithmen und Hashes.

## Examples

### RijndaelManaged

Erforderlicher Namensraum: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

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

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

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

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

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

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

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

### Verwendungszweck

```
var textToEncrypt = "hello World";

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

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

### Hinweis:

- Rijndael ist der Vorgänger des standardmäßigen symmetrischen kryptographischen Algorithmus AES.

## Daten mit AES verschlüsseln und entschlüsseln (in C #)

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

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

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

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

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

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

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

        byte[] encrypted;

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

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

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

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

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

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

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

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

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

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

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

        return plaintext;
    }
}
}
```

Dieses Beispiel stammt von MSDN .

Es ist eine Konsolen-Demoanwendung, die zeigt, wie eine Zeichenfolge mit der Standard- **AES-**Verschlüsselung verschlüsselt und anschließend entschlüsselt wird.

( **AES = Advanced Encryption Standard** , eine Spezifikation, die 2001 vom US-amerikanischen National Institute of Standards und Technology (NIST) für die Verschlüsselung elektronischer Daten festgelegt wurde und nach wie vor der De-facto-Standard für symmetrische Verschlüsselung ist.)

**Anmerkungen:**

- In einem echten Verschlüsselungsszenario müssen Sie einen geeigneten Verschlüsselungsmodus auswählen (kann der Eigenschaft `Mode` zugewiesen werden, indem Sie einen Wert aus der `CipherMode` Enumeration `CipherMode` ). Verwenden `CipherMode.ECB` **niemals** `CipherMode.ECB` (Electronic Codebook-Modus), da dies einen schwachen Chiffrestream hervorruft
- Um einen guten (und nicht einen schwachen) `Key` zu erstellen, verwenden Sie entweder einen kryptographischen Zufallsgenerator oder das obige Beispiel ( **Erstellen eines Schlüssels aus einem Kennwort** ). Die empfohlene **Schlüsselgröße** ist 256 Bit. Unterstützte Schlüsselgrößen sind über die `LegalKeySizes` Eigenschaft verfügbar.
- Um den Initialisierungsvektor `IV` zu initialisieren, können Sie ein SALT verwenden, wie im obigen Beispiel gezeigt ( **Random SALT** ).
- Unterstützte Blockgrößen sind über die Eigenschaft `SupportedBlockSizes` verfügbar. Die Blockgröße kann über die Eigenschaft `BlockSize` zugewiesen werden

**Verwendung:** siehe Main () - Methode.

## Erstellen eines Schlüssels aus einem Passwort / zufälligem SALT (in C #)

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

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

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

        byte[] salt = CreateRandomSalt(7);

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

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

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

```
            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

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

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

        Console.ReadLine();
    }

    #region Helper methods

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

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

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

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

        // return the bytes.
        return randBytes;
    }

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

```
        }

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

    #endregion
}
```

Dieses Beispiel stammt aus MSDN.

Es ist eine Konsolen-Demo. Sie zeigt, wie Sie einen sicheren Schlüssel basierend auf einem benutzerdefinierten Kennwort erstellen und ein zufälliges SALT basierend auf dem kryptografischen Zufallsgenerator erstellen.

**Anmerkungen:**

- Die integrierte Funktion `PasswordDeriveBytes` verwendet den Standard-PBKDF1-Algorithmus, um einen Schlüssel aus dem Kennwort zu generieren. Standardmäßig werden 100 Iterationen verwendet, um den Schlüssel zu generieren, um Brute-Force-Angriffe zu verlangsamen. Das zufällig erzeugte SALT verstärkt den Schlüssel weiter.
- Die Funktion `CryptDeriveKey` konvertiert den von `PasswordDeriveBytes` generierten Schlüssel in einen mit dem angegebenen Verschlüsselungsalgorithmus (hier "TripleDES") kompatiblen Schlüssel, indem der angegebene Hash-Algorithmus (hier "SHA1") verwendet wird. Die Schlüsselgröße beträgt in diesem Beispiel 192 Bytes, und der Initialisierungsvektor IV wird vom Triple-DES-Krypto-Provider übernommen
- Normalerweise wird dieser Mechanismus verwendet, um einen stärker zufällig generierten Schlüssel durch ein Kennwort zu schützen, das große Datenmengen verschlüsselt. Sie können es auch verwenden, um mehrere Passwörter verschiedener Benutzer bereitzustellen, um Zugriff auf dieselben Daten zu gewähren (die durch einen anderen Zufallsschlüssel geschützt werden).
- Leider unterstützt `CryptDeriveKey` derzeit kein AES. Sehen Sie hier.
  **HINWEIS:** Zur Problemumgehung können Sie einen zufälligen AES-Schlüssel zur Verschlüsselung der mit AES zu schützenden Daten erstellen und den AES-Schlüssel in einem TripleDES-Container speichern, der den von `CryptDeriveKey` generierten Schlüssel `CryptDeriveKey` . Dies beschränkt jedoch die Sicherheit auf TripleDES, nutzt die größeren Schlüsselgrößen von AES nicht und schafft eine Abhängigkeit von TripleDES.

**Verwendung:** Siehe Main () - Methode.

## Verschlüsselung und Entschlüsselung mittels Kryptographie (AES)

Entschlüsselungscode

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

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

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

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

        return cipherText;
    }
```

Verschlüsselungscode

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

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

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

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

Verwendungszweck

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

Verschlüsselung / Kryptographie online lesen: https://riptutorial.com/de/dot-net/topic/7615/verschlusselung---kryptographie

# Kapitel 56: Wörterbücher

## Examples

### Wörterbuch auflisten

Sie können auf drei verschiedene Arten durch ein Wörterbuch aufzählen:

#### Verwenden von KeyValue-Paaren

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

#### Schlüssel verwenden

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

#### Werte verwenden

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

### Wörterbuch mit einem Collection-Initialisierer initialisieren

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

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

## Hinzufügen zu einem Wörterbuch

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

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

Alternativ können sie über einen Indexer hinzugefügt / eingestellt werden. (Ein Indexer sieht intern aus wie eine Eigenschaft mit Get und Set, nimmt jedoch einen Parameter eines beliebigen Typs, der zwischen den Klammern angegeben ist.)

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

Im Gegensatz zur `Add` Methode, die eine Ausnahme auslöst, ersetzt der Indexer nur den vorhandenen Wert, wenn ein Schlüssel bereits im Wörterbuch enthalten ist.

Für threadsicheres Wörterbuch verwenden Sie `ConcurrentDictionary<TKey, TValue>` :

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

## Einen Wert aus einem Wörterbuch erhalten

Diesen Setup-Code gegeben:

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

Sie können den Wert für die Eingabe mit der Taste 1 lesen möchten Wenn Schlüssel nicht vorhanden Wert bekommen werfen `KeyNotFoundException` , so können Sie für das mit dem ersten Scheck wollen `ContainsKey` :

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

Dies hat einen Nachteil: Sie durchsuchen Ihr Wörterbuch zweimal (einmal, um die Existenz zu prüfen und einmal, um den Wert zu lesen). Bei einem großen Wörterbuch kann sich dies auf die Leistung auswirken. Glücklicherweise können beide Operationen zusammen ausgeführt werden:

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

### Erstellen Sie ein Wörterbuch mit Case-Insensivitve-Tasten.

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

### ConcurrentDictionary (ab .NET 4.0)

> Stellt eine thread-sichere Sammlung von Schlüssel / Wert-Paaren dar, auf die mehrere Threads gleichzeitig zugreifen können.

## Instanz erstellen

Das Erstellen einer Instanz funktioniert ähnlich wie bei `Dictionary<TKey, TValue>` , z. B .:

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

## Hinzufügen oder Aktualisieren

Sie werden vielleicht überrascht sein, dass es keine `Add` Methode gibt, stattdessen `AddOrUpdate` mit zwei Überladungen:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Fügt ein Schlüssel / Wert-Paar hinzu, wenn der Schlüssel noch nicht vorhanden ist, oder aktualisiert ein Schlüssel / Wert-Paar, wenn der Schlüssel verwendet wird ist bereits vorhanden.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` - *Verwendet die angegebenen Funktionen, um ein Schlüssel / Wert-Paar zum Schlüssel hinzuzufügen, wenn der Schlüssel noch nicht vorhanden ist, oder bis Aktualisieren Sie ein Schlüssel / Wert-Paar, wenn der Schlüssel bereits vorhanden ist.*

Einen Wert hinzufügen oder aktualisieren, unabhängig davon, welcher Wert vorhanden war, wenn er für den angegebenen Schlüssel bereits vorhanden war (1):

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

Einen Wert hinzufügen oder aktualisieren, aber jetzt den Wert in update ändern, basierend auf dem vorherigen Wert (1):

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

Mit der Überladung (2) können wir auch eine neue Fabrik hinzufügen:

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

# Wert bekommen

Einen Wert zu erhalten ist derselbe wie beim `Dictionary<TKey,TValue>` :

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

# Einen Wert abrufen oder hinzufügen

Es gibt zwei Überladungen, die einen Wert fadensicher abrufen **oder hinzufügen** .

Holen Sie sich den Wert mit Taste 2 oder fügen Sie den Wert "Second" hinzu, wenn der Schlüssel nicht vorhanden ist:

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

Verwenden einer Factory zum Hinzufügen eines Werts, wenn kein Wert vorhanden ist:

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

## IEnumerable to Dictionary (≥ .NET 3.5)

Erstellen Sie ein Wörterbuch <TKey, TValue> aus einem IEnumerable <T> :

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

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

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


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

## Aus einem Wörterbuch entfernen

Diesen Setup-Code gegeben:

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

Verwenden Sie die `Remove` Methode, um einen Schlüssel und den zugehörigen Wert zu entfernen.

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

Durch Ausführen dieses Codes werden der Schlüssel `2` und sein Wert aus dem Wörterbuch entfernt. `Remove` gibt einen booleschen Wert zurück, der angibt, ob der angegebene Schlüssel gefunden und aus dem Wörterbuch entfernt wurde. Wenn der Schlüssel nicht im Wörterbuch vorhanden ist, wird nichts aus dem Wörterbuch entfernt, und es wird false zurückgegeben (es wird keine Ausnahme ausgelöst).

Es ist **falsch** , einen Schlüssel zu versuchen und zu entfernen, indem der Wert für den Schlüssel auf `null` .

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

Der Schlüssel wird dadurch nicht entfernt. Der vorherige Wert wird einfach durch den Wert `null` .

Um alle Schlüssel und Werte aus einem Wörterbuch zu entfernen, verwenden Sie die `Clear` Methode.

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

Nach der Ausführung von `Clear` der `Count` des Wörterbuchs 0, die interne Kapazität bleibt jedoch unverändert.

## ContainsKey (TKey)

Um zu überprüfen, ob ein `Dictionary` einen bestimmten Schlüssel hat, können Sie die Methode `ContainsKey(TKey)` und den Schlüssel des Typs `TKey` . Die Methode gibt einen `bool` Wert zurück, wenn der Schlüssel im Wörterbuch vorhanden ist. Für probe:

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

Und prüfen Sie, ob ein `C2` im Wörterbuch vorhanden ist:

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

Die ContainsKey-Methode ist in der generischen Version `Dictionary<TKey, TValue>` .

### Wörterbuch zur Liste

Erstellen einer Liste von KeyValuePair:

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

Erstellen einer Schlüsselliste:

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

Erstellen einer Liste von Werten:

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

### ConcurrentDictionary, erweitert mit Lazy'1, reduziert doppelte Berechnungen

## Problem

ConcurrentDictionary glänzt, wenn es darum geht, vorhandene Schlüssel aus dem Cache (meistens ohne Sperren) zurückzugeben und auf einer granularen Ebene zu kämpfen. Was aber, wenn die Objekterstellung wirklich teuer ist und die Kosten für das Kontextwechseln überwiegen und einige Cache-Fehler auftreten?

Wenn derselbe Schlüssel von mehreren Threads angefordert wird, wird eines der Objekte, die aus kollidierenden Operationen resultieren, schließlich der Auflistung hinzugefügt, und die anderen werden verworfen, wodurch die CPU-Ressource für das Erstellen des Objekts und der Speicherressource für das temporäre Speichern des Objekts verschwendet wird . Andere Ressourcen könnten ebenfalls verschwendet werden. Das ist wirklich schlimm.

## Lösung

Wir können `ConcurrentDictionary<TKey, TValue>` mit `Lazy<TValue>` . Die Idee ist, dass die ConcurrentDictionary-Methode GetOrAdd nur den Wert zurückgeben kann, der der Auflistung

tatsächlich hinzugefügt wurde. Das Verlieren von Lazy-Objekten könnte auch in diesem Fall verschwendet werden, was aber kein großes Problem darstellt, da das Lazy-Objekt selbst relativ teuer ist. Die Value-Eigenschaft des verlierenden Lazy wird nie angefordert, da wir nur die Value-Eigenschaft der der Collection tatsächlich hinzugefügten Eigenschaft anfordern können - derjenigen, die von der GetOrAdd-Methode zurückgegeben wird:

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

Das Zwischenspeichern von XmlSerializer-Objekten kann besonders teuer sein und auch beim Start der Anwendung gibt es viele Konflikte. Dazu gehört noch mehr: Wenn es sich um benutzerdefinierte Serialisierer handelt, wird auch für den Rest des Prozesslebenszyklus ein Speicherverlust entstehen. Der einzige Vorteil von ConcurrentDictionary besteht in diesem Fall darin, dass für den Rest des Prozesslebenszyklus keine Sperren vorhanden sind, der Anwendungsstart und die Speicherverwendung jedoch nicht akzeptabel sind. Dies ist ein Job für unser ConcurrentDictionary, ergänzt mit Lazy:

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

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

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

Wörterbücher online lesen: https://riptutorial.com/de/dot-net/topic/45/worterbucher

# Kapitel 57: XmlSerializer

## Bemerkungen

Verwenden Sie den `XmlSerializer` zum Parsen von `HTML` . Dafür stehen spezielle Bibliotheken wie das HTML Agility Pack zur Verfügung

## Examples

### Objekt serialisieren

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

### Objekt deserialisieren

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

### Verhalten: Ordnen Sie den Elementnamen der Eigenschaft zu

```
<Foo>
    <Dog/>
</Foo>
```

- 

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

### Verhalten: Ordnen Sie den Arraynamen der Eigenschaft zu (XmlArray).

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

- 

```
public class Store
{
    [XmlArray("Articles")]
    public List<Product> Products {get; set; }
}
```

### Formatierung: Benutzerdefiniertes DateTime-Format

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat,
CultureInfo.InvariantCulture); }
    }
}
```

### Effizientes Erstellen mehrerer Serialisierer mit dynamisch festgelegten abgeleiteten Typen

## Woher wir kamen

Manchmal können wir nicht alle erforderlichen Metadaten bereitstellen, die für das XmlSerializer-Framework in Attribut erforderlich sind. Nehmen wir an, wir haben eine Basisklasse von serialisierten Objekten, und einige der abgeleiteten Klassen sind der Basisklasse unbekannt. Wir können kein Attribut für alle Klassen platzieren, die zum Entwurfszeitpunkt des Basistyps nicht bekannt sind. Wir könnten ein anderes Team haben, das einige der abgeleiteten Klassen entwickelt.

## Was können wir tun

Wir können den `new XmlSerializer(type, knownTypes)` , aber das wäre eine O (N ^ 2) `new XmlSerializer(type, knownTypes)` für N Serialisierer, um zumindest alle in Argumenten angegebenen Typen zu ermitteln:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

In diesem Beispiel kennt der Basistyp die abgeleiteten Typen nicht, was in OOP normal ist.

## Effizient machen

Glücklicherweise gibt es eine Methode, die dieses spezielle Problem anspricht - bekannte Typen für mehrere Serialisierer effizient bereitzustellen:

System.Xml.Serialization.XmlSerializer.FromTypes-Methode (Typ [])

> Mit der FromTypes-Methode können Sie effizient ein Array von XmlSerializer-Objekten zum Verarbeiten eines Arrays von Type-Objekten erstellen.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Hier ist ein komplettes Codebeispiel:

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

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
```

```
            Serialize(sampleObject);
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
}
```

Ausgabe:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
This error was anticipated,
we have not supplied a derived class.
System.InvalidOperationException: There was an error generating the XML document. --->
System.InvalidOperationException: The type Program+Derived was not expected. Use the
XmlInclude or SoapInclude attribute to specify types that are not known statically.
   at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String
n, String ns, Base o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   --- End of inner exception stack trace ---
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
   at Program.Serialize[T](T o)
```

```
   at Program.Main()
Now trying to serialize with all of the type information:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Base xsi:type="Derived">
    <JustSomePropInBase>0</JustSomePropInBase>
    <JustSomePropInDerived>0</JustSomePropInDerived>
  </Base>
</Container>
Slides down well this time!
```

## Was ist in der Ausgabe?

In dieser Fehlermeldung wird empfohlen, was wir vermeiden wollten (oder was wir in einigen Szenarien nicht tun können). Dabei werden abgeleitete Typen aus der Basisklasse referenziert:

```
Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.
```

So bekommen wir unsere abgeleitete Klasse im XML:

```
<Base xsi:type="Derived">
```

`Base` entspricht dem im `Container` deklarierten Eigenschaftstyp. `Derived` ist der Typ der tatsächlich gelieferten Instanz.

Hier ist eine Arbeitsbeispiel- Geige

XmlSerializer online lesen: https://riptutorial.com/de/dot-net/topic/31/xmlserializer

# Kapitel 58: Zeichenketten

## Bemerkungen

In .NET-Zeichenfolgen ist `System.String` eine Folge von Zeichen `System.Char` . Jedes Zeichen ist eine UTF-16-codierte `System.String` . Diese Unterscheidung ist wichtig, da *die* Definition der *Zeichen in gesprochener Sprache und die* Definition von *Zeichen in* .NET (und vielen anderen Sprachen) unterschiedlich sind.

Ein *Zeichen* , das korrekt als Graphem bezeichnet werden sollte , wird als Glyphe angezeigt und durch einen oder mehrere Unicode- Codepunkte definiert. Jeder Codepunkt wird dann in einer Folge von Codeeinheiten codiert. Jetzt sollte klar sein, warum ein einzelnes `System.Char` nicht immer ein Graphem darstellt. Sehen wir uns in der Realität an, wie sie sich unterscheiden:

- Ein Graphem kann aufgrund der Kombination von Zeichen zu zwei oder mehr Codepunkten führen: `à` setzt sich aus zwei Codepunkten zusammen: *U + 0061 LATEINER KLEINER BUCHSTABE A* und *U + 0300* . Dies ist der häufigste Fehler, weil `"à ".Length == 2` während Sie `1` erwarten können.
- Es gibt Zeichen dupliziert, zum Beispiel `à` eine einzige Codepunkt *U + 00E0 Kleines a mit Gravis* oder zwei Code-Punkte sein können , wie oben erläutert. Offensichtlich müssen sie dasselbe vergleichen: `"\u00e0" == "\u0061\u0300"` (auch wenn `"\u00e0".Length != "\u0061\u0300".Length` ). Dies ist möglich, da die *Zeichenfolge normalisiert wird* , indem die Methode `String.Normalize()` verwendet wird.
- Eine Unicode - Sequenz , die eine zusammengesetzte oder zerlegt Sequenz enthalten kann, beispielsweise Zeichen  *U + D55C HAN CHARACTER* kann einen einzigen Codepunkt (codiert als einzelne Code-Einheit in UTF-16) oder eine zerlegt Sequenz seiner Silben , und  . Sie müssen gleich verglichen werden.
- Ein Codepunkt kann in mehr als eine `Codeeinheit` codiert werden: Das Zeichen  *U + 2008A HAN CHARACTER* ist als zwei `System.Char` Codierung ( `"\ud840\udc8a"` ) `"\ud840\udc8a"` auch wenn es sich nur um einen Codepunkt handelt: UTF-16 Kodierung hat keine feste Größe! Dies ist eine Quelle unzähliger Fehler (auch schwerwiegender Sicherheitsfehler). Wenn Ihre Anwendung beispielsweise eine maximale Länge anwendet und den String blind verkürzt, können Sie einen ungültigen String erstellen.
- Einige Sprachen haben digraph und trigraphs, zum Beispiel in der Tschechischen `ch` ist ein Standalone - Brief (nach `h` und bevor `ich` dann , wenn eine Liste von Strings Bestellung Sie *fyzika* vor *Chemie* hat.

Es gibt viel mehr Probleme bei der Textverarbeitung, siehe beispielsweise Wie kann ich einen Unicode-fähigen Zeichenvergleich durchführen? für eine breitere Einführung und mehr Links zu verwandten Argumenten.

Im Allgemeinen können Sie beim Umgang mit *internationalem* Text diese einfache Funktion verwenden, um Textelemente in einer Zeichenfolge aufzulisten (um Unicode-Ersatzzeichen und die Kodierung nicht zu beschädigen):

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

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

# Examples

## Zähle verschiedene Charaktere

Wenn Sie unterschiedliche Zeichen zählen müssen, können Sie aus den im Abschnitt " *Bemerkungen"* erläuterten Gründen nicht einfach die `Length` -Eigenschaft verwenden, da es sich bei der Länge des Arrays von `System.Char` nicht um Zeichen, sondern um Codeeinheiten (nicht Unicode-Codepunkte) handelt noch Grapheme). Wenn Sie beispielsweise einfach `text.Distinct().Count()` schreiben `text.Distinct().Count()` Sie falsche Ergebnisse, richtigen Code:

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

Ein weiterer Schritt besteht darin, das **Vorkommen jedes Zeichens** zu **zählen.** Wenn die Leistung kein Problem darstellt, können Sie dies einfach wie folgt tun (in diesem Beispiel unabhängig von Fall):

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

## Zeichen zählen

Wenn Sie *Zeichen* zählen müssen, können Sie aus den im Abschnitt " *Bemerkungen"* erläuterten Gründen nicht einfach die Length -Eigenschaft verwenden, da es sich bei der Länge des Arrays von `System.Char` nicht um Zeichen, sondern um Codeeinheiten (nicht um Unicode-Codepunkte) handelt Graphemen). Korrekter Code lautet dann:

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

Eine kleine Optimierung kann die `EnumerateCharacters()` Erweiterungsmethode speziell für diesen Zweck neu schreiben:

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

```
            return 0;

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

        return count;
    }
}
```

## Zählen Sie Vorkommen eines Zeichens

Aus den im Abschnitt " *Anmerkungen" genannten* Gründen können Sie dies nicht einfach tun (es sei denn, Sie möchten das Vorkommen einer bestimmten Code-Einheit zählen):

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

Sie benötigen eine komplexere Funktion:

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

Beachten Sie, dass der Zeichenkettenvergleich (im Gegensatz zum Zeichenvergleich, der kulturinvariant ist) immer nach Regeln für eine bestimmte Kultur durchgeführt werden muss.

## String in Blöcke mit fester Länge aufteilen

Wir können nicht einen String in beliebigen Punkten brechen (weil ein `System.Char` nicht allein gültig sein kann , weil es sich um eine Kombination von Zeichen oder ein Teil eines Surrogat ist) , dann Code, berücksichtigen müssen (beachten Sie, dass mit der *Länge* meine ich die Anzahl der *Grapheme* nicht die Anzahl *Code-Einheiten* ):

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

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

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

### Konvertieren Sie die Zeichenfolge in eine andere Kodierung

.NET-Zeichenfolgen enthalten `System.Char` (UTF-16-Codeeinheiten). Wenn Sie Text mit einer anderen Kodierung speichern (oder verwalten) möchten, müssen Sie mit einem Array von `System.Byte` .

Konvertierungen werden von Klassen durchgeführt, die von `System.Text.Encoder` und `System.Text.Decoder` abgeleitet `System.Text.Encoder` und zusammen in eine andere Codierung konvertieren können (von einem Byte *X*- codierten Array `byte[]` in einen UTF-16-codierten `System.String` und einen Vice) -versa).

Da der Encoder / Decoder normalerweise sehr nahe beieinander arbeitet, werden sie in einer von `System.Text.Encoding` abgeleiteten Klasse zusammengefasst. `System.Text.Encoding` bieten abgeleitete Klassen Konvertierungen zu / von gängigen Codierungen (UTF-8, UTF-16 usw.).

# Beispiele:

## Konvertieren Sie eine Zeichenfolge in UTF-8

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

---

## Konvertieren Sie UTF-8-Daten in einen String

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

---

## Kodierung einer vorhandenen Textdatei ändern

Dieser Code liest den Inhalt einer UTF-8-codierten Textdatei und speichert sie als UTF-16 codiert ab. Beachten Sie, dass dieser Code nicht optimal ist, wenn die Datei groß ist, da der gesamte Inhalt in den Speicher gelesen wird:

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

### Virtuelle Object.ToString () - Methode

Alles in .NET ist ein Objekt, daher hat jeder Typ die `ToString()` Methode, die in der `Object` Klasse definiert ist, die überschrieben werden kann. Die Standardimplementierung dieser Methode gibt nur den Namen des Typs zurück:

```
public class Foo
{
}
```

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

`ToString()` wird implizit aufgerufen, wenn der Wert mit einer Zeichenfolge verknüpft wird:

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

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

Das Ergebnis dieser Methode wird auch von Debugging-Tools ausgiebig verwendet. Wenn Sie diese Methode aus irgendeinem Grund nicht überschreiben möchten, sondern anpassen möchten, wie der Debugger den Wert Ihres Typs anzeigt, verwenden Sie das DebuggerDisplay-Attribut ( MSDN ):

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

## Unveränderlichkeit von Saiten

Strings sind unveränderlich. Sie können die vorhandene Zeichenfolge einfach nicht ändern. Bei jeder Operation der Zeichenfolge wird eine neue Instanz der Zeichenfolge mit einem neuen Wert erstellt. Das bedeutet, wenn Sie ein einzelnes Zeichen in einer sehr langen Zeichenfolge ersetzen müssen, wird Speicher für einen neuen Wert reserviert.

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

Wenn Sie viele Operationen mit einem String-Wert durchführen müssen, verwenden Sie die `StringBuilder` Klasse, die für eine effiziente Bearbeitung von Strings ausgelegt ist:

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

## Vergleichende Zeichenketten

Trotz `String` ist ein Referenztyp `==` Operator vergleicht Zeichenfolgenwerte statt Referenzen.

Wie Sie vielleicht wissen, ist `string` nur ein Array von Zeichen. Wenn Sie jedoch der Meinung sind, dass die Gleichheitsprüfung und der Vergleich von Zeichenfolgen Zeichen für Zeichen vorgenommen werden, sind Sie falsch. Diese Operation ist kulturspezifisch (siehe Anmerkungen unten): Einige Zeichenfolgen können je nach Kultur als gleich behandelt werden.

Denken Sie zweimal nach, bevor Sie die Gleichheitsprüfung kurzschließen, indem Sie die `Length` Eigenschaften von zwei Strings vergleichen!

Verwenden Sie Überladungen der `String.Equals` Methode, die zusätzlichen `StringComparison` Enumerationswert akzeptieren, wenn Sie das Standardverhalten ändern müssen.

Zeichenketten online lesen: https://riptutorial.com/de/dot-net/topic/2227/zeichenketten

# Kapitel 59: Zip-Dateien lesen und schreiben

## Einführung

Die **ZipFile-** Klasse befindet sich im Namespace **System.IO.Compression** . Es kann zum Lesen und Schreiben von Zip-Dateien verwendet werden.

## Bemerkungen

- Sie können auch einen MemoryStream anstelle eines FileStream verwenden.
- Ausnahmen

| Ausnahme | Bedingung |
|---|---|
| ArgumentException | Der Stream wurde bereits geschlossen oder die Fähigkeiten des Streams stimmen nicht mit dem Modus überein (zB: Versuch, in einen Read-Only-Stream zu schreiben) |
| ArgumentNullException | *Eingabestrom* ist null |
| ArgumentOutOfRangeException | *Modus* hat einen ungültigen Wert |
| InvalidDataException | Siehe nachstehende Liste |

Wenn eine **InvalidDataException** ausgelöst wird, kann dies drei Ursachen haben:

- Der Inhalt des Streams konnte nicht als ZIP-Archiv interpretiert werden
- *Modus* ist Update und ein Eintrag fehlt im Archiv oder ist beschädigt und kann nicht gelesen werden
- *Modus* ist Update und ein Eintrag ist zu groß, um in den Speicher zu passen

Alle Informationen wurden dieser MSDN-Seite entnommen

## Examples

### ZIP-Inhalt auflisten

Dieses Snippet listet alle Dateinamen eines ZIP-Archivs auf. Die Dateinamen sind relativ zur Zip-Wurzel.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
```

```
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
    }
}
```

## Extrahieren von Dateien aus ZIP-Dateien

Das Extrahieren aller Dateien in ein Verzeichnis ist sehr einfach:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

Wenn die Datei bereits vorhanden ist, wird eine **System.IO.IOException** ausgelöst.

Extrahieren bestimmter Dateien:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Jede dieser Methoden führt zum gleichen Ergebnis.

## ZIP-Datei aktualisieren

Um eine ZIP-Datei zu aktualisieren, muss die Datei stattdessen mit ZipArchiveMode.Update geöffnet werden.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

Es besteht auch die Möglichkeit, direkt in eine Datei im Archiv zu schreiben:

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Zip-Dateien lesen und schreiben online lesen: https://riptutorial.com/de/dot-net/topic/9943/zip-dateien-lesen-und-schreiben

# Credits

| S. No | Kapitel | Contributors |
|---|---|---|
| 1 | Erste Schritte mit .NET Framework | Adriano Repetti, Alan McBee, ale10ander, Andrew Jens, Andrew Morton, Andrey Shchekin, Community, Daniel A. White, Ehsan Sajjad, harriyott, hillary.fraley, Ian, James Thorpe, Jamie Rees, Joel Martinez, Kevin Montrose, Lirrik, MarcinJuraszek, matteeyah, naveen, Nicholas Sizer, Paweł Izdebski, Peter, Peter Gordon, Peter Hommel, PSN, Richard Lander, Rion Williams, Robert Columbia, RubberDuck, SeeuD1, Serg Rogovtsev, Squidward, Stephen Leppik, Steven Doggart, svick, ɬolɐəz əuɬ qoq |
| 2 | .NET Core | Mihail Stancescu |
| 3 | Abhängigkeitsspritze | Phil Thomas, Scott Hannen |
| 4 | ADO.NET | Akshay Anand, Andrew Morton, Daniel A. White, DavidG, Drew, elmer007, Hamid, Harjot, Heinzi, Igor, user2321864 |
| 5 | Akronym Glossar | Tanveer Badar |
| 6 | Ausdrucksbäume | Akshay Anand, George Polevoy, Jim, n.podbielski, Pavel Mayorov, RamenChef, Stephen Leppik, Stilgar, wangengzheng |
| 7 | Ausnahmen | Adi Lester, Akshay Anand, Alan McBee, Alfred Myers, Arvin Baccay, BananaSft, CodeCaster, Dave R., Kritner, Mafii, Matt, Rob, Sean, starbeamrainbowlabs, STW, Yousef Al-Mulla |
| 8 | Benutzerdefinierte Typen | Alan McBee, DrewJordan, matteeyah |
| 9 | CLR | Gajendra, starbeamrainbowlabs, Theodoros Chatzigiannakis |
| 10 | Code-Verträge | JJS, Matthew Whited, RamenChef |
| 11 | Datei Eingabe / Ausgabe | ale10ander, Alexander Mandt, Ingenioushax, |

| | | |
|---|---|---|
| | | Nitram |
| 12 | DateTime-Analyse | GalacticCowboy, John |
| 13 | die Einstellungen | Alan McBee |
| 14 | Einfädeln | Behzad, Martijn Pieters, Mellow |
| 15 | Fortschritt verwenden und IProgress | DLeh |
| 16 | Für jeden | Dr Rob Lang, just.ru, Lucas Trzesniewski |
| 17 | Globalisierung in ASP.NET MVC mithilfe von Smart Internationalisierung für ASP.NET | Scott Hannen |
| 18 | HTTP-Clients | CodeCaster, Konamiman, MuiBienCarlota |
| 19 | HTTP-Server | Devon Burriss, Konamiman |
| 20 | JIT-Compiler | Krikor Ailanjian |
| 21 | JSON in .NET mit Newtonsoft.Json | DLeh |
| 22 | JSON-Serialisierung | Akshay Anand, Andrius, Eric, hasan, M22an, PedroSouki, Thriggle, Tolga Evcimen |
| 23 | Laden Sie Datei- und POST-Daten auf den Webserver hoch | Aleks Andreev |
| 24 | LINQ | A. Raza, Adil Mammadov, Akshay Anand, Alexander V., Benjamin Hodgson, Blachshma, Bradley Grainger, Bruno Garcia, Carlos Muñoz, CodeCaster, dbasnett, DoNot , dotctor, Eduardo Molteni, Ehsan Sajjad, GalacticCowboy, H. Pauwelyn, Haney, J3soon, jbtule, jnovo, Joe Amenta, Kilazur, Konamiman, MarcinJuraszek, Mark Hurd, McKay, Mellow, Mert Gülsoy, Mike Stortz, Mr.Mindor, Nate Barbettini, Pavel Voronin, Ruben Steins, Salvador Rubio Martinez, Sammi, Sergio Domínguez, Sidewinder94 |
| 25 | Managed Extensibility Framework | Joe Amenta, Kirk Broadhurst, RamenChef |
| 26 | Mit SHA1 in C # arbeiten | mahdi abasi |
| 27 | Müllsammlung | avat |
| 28 | NuGet-Verpackungssystem | Andrey Shchekin, Anik Saha, Ashtonian, |

| | | |
|---|---|---|
| | | CodeCaster, Daniel A. White, Matas Vaitkevicius, Ozair Kafray |
| 29 | Parallele Verarbeitung mit .Net Framework | Yahfoufi |
| 30 | Plattform aufrufen | Dmitry Egorov, Imran Ali Khan |
| 31 | Prozess- und Thread-Affinitätseinstellung | MSE, RamenChef |
| 32 | ReadOnlyCollections | tehDorf |
| 33 | Reflexion | Aleks Andreev, Bjørn-Roger Kringsjå, demonplus, Jean-Baptiste Noblot, Jigar, JJP , Kirk Broadhurst, Lorenzo Dematté, Matas Vaitkevicius, NetSquirrel, Pavel Mayorov, Peter, smdrager, Terry, user1304444, void |
| 34 | Reguläre Ausdrücke (System.Text.RegularExpressions) | BrunoLM, Denuath, Matt dc, tehDorf |
| 35 | Sammlungen | Alan McBee, Aman Sharma, Anik Saha, Daniel A. White, demonplus, Felipe Oriani, harriyott, Ian, Mark C., Ravi A., Virtlink |
| 36 | Schreiben Sie in den StdErr-Stream und lesen Sie ihn aus | Aleks Andreev |
| 37 | Serielle Ports | Dmitry Egorov |
| 38 | SpeechRecognitionEngine-Klasse zum Erkennen von Sprache | ProgramFOX, RamenChef |
| 39 | Speicherverwaltung | Big Fan, binki, DrewJordan |
| 40 | Stapel und Haufen | Hywel Rees |
| 41 | Synchronisierungskontexte | DLeh, Gusdor |
| 42 | System.IO | CodeCaster, Daniel A. White, demonplus, Filip Frącz, RoyalPotato |
| 43 | System.IO.File-Klasse | Adriano Repetti, delete me |
| 44 | System.Net.Mail | demonplus, Steve, vicky |
| 45 | System.Reflection.Emit-Namespace | Luaan, NikolayKondratyev, RamenChef, toddmo |

| | | |
|---|---|---|
| 46 | System.Runtime.Caching.MemoryCache (ObjectCache) | Guanxi, RamenChef |
| 47 | Systemdiagnose | Adi Lester, Bassie, Fredou, Ogglas, Ondřej Štorc, RamenChef |
| 48 | Task Parallel Library (TPL) | Adi Lester, Aman Sharma, Andrew, i3arnon, Jacobr365, JamyRyals, Konamiman, Mathias Müller, Mert Gülsoy, Mikhail Filimonov, Pavel Mayorov, Pavel Voronin, RamenChef, Thomas Bledsoe, TorbenJ |
| 49 | TPL-API-Übersichten (Task Parallel Library) | Gusdor, Jacobr365 |
| 50 | TPL-Datenfluss | i3arnon, Jacobr365, Nikola.Lukovic, RamenChef |
| 51 | Unit-Tests | Axarydax |
| 52 | VB-Formulare | ale10ander, dbasnett |
| 53 | Vernetzung | Konamiman |
| 54 | Verschlüsselung / Kryptographie | Alexander Mandt, Daniel A. White, demonplus, Jagadisha B S, lokusking, Matt |
| 55 | Wörterbücher | Adriano Repetti, Bjørn-Roger Kringsjå, Daniel Plaisted, Darrel Lee, Felipe Oriani, George Duckett, George Polevoy, hatchet, Hogan, Ian, LegionMammal978, Luke Bearl, Olivier Jacot-Descombes, RamenChef, Ringil, Robert Columbia, Stephen Byrne, the berserker, Tomáš Hübelbauer |
| 56 | XmlSerializer | Aphelion, George Polevoy, RamenChef, Rowland Shaw, Thomas Levesque, void, Yogi |
| 57 | Zeichenketten | Adriano Repetti, Alexander Mandt, Matt, Pavel Voronin, RamenChef |
| 58 | Zip-Dateien lesen und schreiben | Arxae |