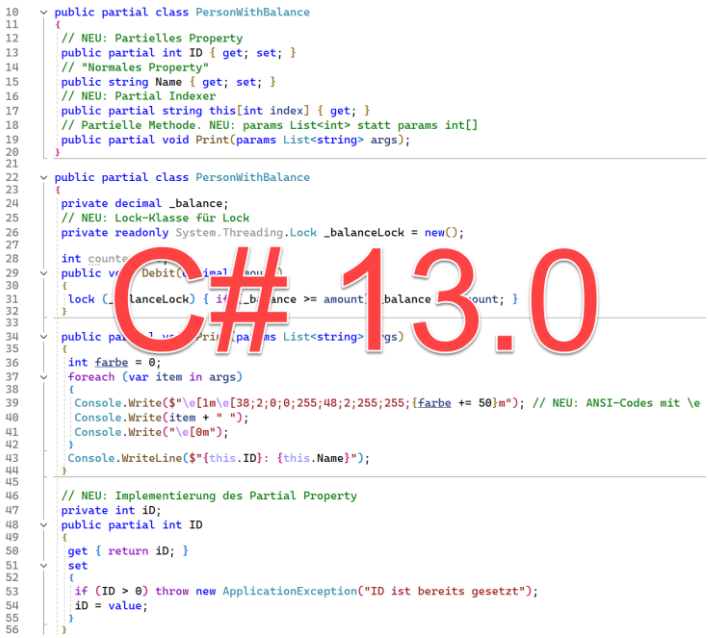
Dr. Holger Schwichtenberg

C# 13.0 Crashkurs

**Die Syntax der Programmiersprache C#
für die Softwareentwicklung
in .NET Framework und .NET bis einschließlich Version 9.0**



www.IT-Visions.de
Dr. Holger Schwichtenberg

Buchversion/Auflage: 13.0.0 vom 01.11.2024
Verlag: www.IT-Visions.de, Fahrenberg 40b, D-45257 Essen
Sprachliche Korrektur: Matthias Bloch, Heike Rickert, Dorothea Fleischer
ISBN: 978-3-934-27944-5
Bezugsquellen: *www.IT-Visions.de/Buch/CS13*

1 Inhaltsverzeichnis (Hauptkapitel)

2 Inhaltsverzeichnis (Details)

3 Vorwort

Liebe Leserinnen und Leser,

der "C# Crashkurs" ist ein prägnanter Überblick über die Syntax der Programmiersprache C# in der aktuellen Version 13.0, die zusammen mit .NET 9.0 am 12. November 2024 erschienen ist.

Dieses Buch ist geeignet für **Softwareentwickler, die von einer anderen objektorientierten Programmiersprache (z.B. C++, Java, JavaScript, Visual Basic .NET, Delphi oder PHP) auf C# umsteigen wollen** oder bereits C# einsetzen und ihr Wissen erweitern, insbesondere die neusten Sprachfeatures kennenlernen wollen. Wir schulen bei www.IT-Visions.de jedes Jahr hunderte Entwickler auf C# bzw. die neuste Version der Sprache um. Da es viele Umsteiger von Visual Basic .NET zu C# gibt, werden hier die Unterschiede von C# gegenüber Visual Basic .NET an einigen Stellen im Buch hervorgehoben.

Für Neueinsteiger, die mit C# erstmals überhaupt eine objektorientiere Programmiersprache (OOP) erlernen wollen, ist dieses Werk hingegen nicht geeignet, denn es werden die OO-Grundkonzepte nicht erklärt, da die meisten Softwareentwickler heutzutage diese aus anderen Sprachen kennen und das Buch nicht mit diesen Grundlagen unnötig in die Länge gezogen werden soll.

Dieser Crashkurs erhebt nicht den Anspruch, alle syntaktischen Details zu C# aufzuzeigen, sondern nur die **in der Praxis wichtigsten Sprachkonstrukte**.

In diesem Buch werden bewusst alle Syntaxbeispiele anhand von Konsolenanwendungen gezeigt. So brauchen Sie als Leser kein Wissen über irgendeine (manchmal kurzlebige) GUI-Bibliothek und die Beispiele sind prägnant fokussiert auf die Syntax.

Dieses Buch wird vertrieben:

- PDF-E-Book bei Leanpub.com ab 29,99 Dollar (der Autor erhält 19,99 Dollar):
  www.leanpub.com/CSharp13
- Gedruckt (Print-on-Demand) bei Amazon.de für 39,99 Euro (der Autor erhält 15,38 Euro):
  www.amazon.de/exec/obidos/ASIN/3934279449/itvisions-21
- Kindle-E-Book bei Amazon.de für 29,99 Euro (der Autor erhält 9,81 Euro):
  www.amazon.de/exec/obidos/ASIN/B0CM47LGY8/itvisions-21
- Als Teil des E-Book-Buch-Abos zusammen mit anderen aktuellen Fachbüchern ab 99,00 Euro im Jahr inkl. aller Updates (der Autor erhält den kompletten Preis):
  www.IT-Visions.de/BuchAbo

Tipp: Käufer bei Leanpub.com können jederzeit Aktualisierungen des PDF-Buchs (gleiche Hauptversion) kostenfrei dort beziehen. Käufer bei Amazon erhalten die PDF-Ausgabe einmalig kostenfrei (siehe Kapitel "Über dieses Fachbuch"). E-Book-Abonnenten haben jederzeit Zugriff auf alle aktuellsten Ausgaben der Fachbücher von Dr. Holger Schwichtenberg.

Da solch niedrige Preise leider nicht nennenswert dazu beitragen können, den Lebensunterhalt meiner Familie zu bestreiten, ist dieses Projekt ein Hobby. Dementsprechend kann ich nicht garantieren, wann es Updates zu diesem Buch geben wird. Ich werde dann an diesem Buch arbeiten, wenn ich neben meinem Beruf als Softwarearchitekt, Berater und Dozent und meinen sportlichen Betätigungen noch etwas Zeit für das Fachbuchautorenhobby übrig habe.

Falls mir in diesem Buch oder den zugehörigen Downloads menschliche Fehler passiert sind, möchte ich mich dafür schon jetzt in aller Form entschuldigen bei Ihnen. Bitte geben Sie mir einen freundlichen, genau beschriebenen Hinweis auf meine Fehler. Ich freue mich immer über

konstruktives Feedback und Verbesserungsvorschläge. Bitte verwenden Sie dazu das Kontaktformular: *www.dotnet-doktor.de/Leserfeedback*

> **Tipp:** Ich belohne Sie mit E-Books für gemeldete Fehler, siehe Kapitel "Über dieses Buch / Ihre Belohnung, wenn Sie helfen, dieses Buch zu verbessern".

Ich helfe Ihnen gerne, Ihren eigenen Programmcode zu schreiben, aber ich hoffe, Sie verstehen, dass ich dies nicht ehrenamtlich tun kann. Wenn Sie **technische Hilfe** zu Entity Framework und Entity Framework Core oder anderen Themen rund um die Entwicklung und den Betrieb von Anwendungen (Desktop, Web und Mobile) sowie Server und Cloud benötigen, stehe ich Ihnen im Rahmen meiner beruflichen Tätigkeit für die Firma *www.IT-Visions.de* (Beratung, Schulung, Support, Softwareentwicklung) gerne zur Verfügung. Bitte wenden Sie sich für ein Angebot an das jeweilige Kundenteam. Bitte kontaktieren Sie die Firmen aber nicht für Feedback und Verbesserungsvorschläge zu diesem Buch, da dieses Buch reine Privatsache ist.

Auf der von mir ehrenamtlich betriebenen **Leser-Website** unter *www.IT-Visions.de/Leser*, können Sie die Beispiele zu diesem Buch herunterladen. Dort müssen Sie sich registrieren. Bei der Registrierung wird ein Losungswort abgefragt. Bitte geben Sie dort bei der Registrierung das Losungswort **AWAY** ein.

Herzliche Grüße aus Essen, dem Herzen der Metropole Ruhrgebiet

Holger Schwichtenberg

4 Über den Autor

- Studienabschluss Diplom-Wirtschaftsinformatik an der Universität Essen
- Promotion an der Universität Essen im Fachgebiet komponentenbasierter Softwareentwicklung
- Seit 1996 in der IT tätig als Softwareentwickler, Softwarearchitekt, Berater, Dozent und Fachjournalist
- Fachlicher Leiter des Expertenteams bei *www.IT-Visions.de* in Essen
- Über 95 Fachbücher bei verschiedenen Verlagen, u.a. Carl Hanser Verlag, O'Reilly, APress, Microsoft Press und Addison Wesley sowie im Selbstverlag

www.IT-Visions.de
Dr. Holger Schwichtenberg

- Mehr als 1500 Beiträge in Fachzeitschriften und Online-Portalen
- Gutachter in den Wettbewerbsverfahren der EU vs. Microsoft (2006-2009)
- Ständiger Mitarbeiter der Zeitschriften iX (seit 1999), dotnetpro (seit 2000) und Windows Developer (seit 2010) sowie beim Online-Portal heise.de (seit 2008)
- Regelmäßiger Sprecher auf nationalen und internationalen Fachkonferenzen (z.B. BASTA!, Developer Week, .NET Developer Conference, MD DevDays, Microsoft TechEd, Microsoft Summit, Microsoft IT Forum, OOP, .NET Architecture Camp, IT Tage, enterJS, Advanced Developers Conference, DOTNET Cologne, iterate=>ruhr, Community in Motion, DOTNET-Konferenz, VS One, NRW.Conf, Windows Forum, Container Conf)
- Auszeichnungen und Zertifikate von Microsoft:
  - Microsoft Most Valuable Professional (MVP), ununterbrochen ausgezeichnet seit 2004
  - Microsoft Certified Solution Developer (MCSD)
- Thematische Schwerpunkte:
  - Softwarearchitektur, mehrschichtige Softwareentwicklung, Softwarekomponenten
  - Visual Studio, Continous Integration (CI) und Continous Delivery (CD) mit Azure DevOps
  - Microsoft .NET (.NET Framework, .NET Core, modernes .NET), C#, Visual Basic
  - .NET-Architektur, Auswahl von .NET-Techniken
  - Einführung von .NET, Migration auf .NET
  - Webanwendungsentwicklung und Cross-Plattform-Anwendungen mit HTML/CSS, JavaScript/ TypeScript und C# sowie Webframeworks wie Angular, Vue.js, Svelte, ASP.NET (Core) und Blazor
  - Verteilte Systeme/Webservices mit .NET, insbesondere WebAPI, gRPC und WCF/CoreWCF
  - Relationale Datenbanken, XML, Datenzugriffsstrategien
  - Objekt-Relationales Mapping (ORM), insbesondere ADO.NET Entity Framework und Entity Framework Core
  - PowerShell
  - Architektur- und Code-Reviews
  - Performance-Analysen und -Optimierung
  - Entwicklungsrichtlinien

- Ehrenamtliche Community-Tätigkeiten:
  - Vortragender für die International .NET Association (INETA) und .NET Foundation
  - Betrieb diverser Community-Websites:
    www.dotnet-lexikon.de
    www.dotnetframework.de
    www.windows-scripting.de
    www.aspnetdev.de
    u.a.
- Firmenwebsite: *www.IT-Visions.de*
- Weblog: *www.dotnet-doktor.de*
- Kontakt für Anfragen zu Schulung und Beratung sowie Softwareentwicklungsprojekten:
  E-Mail *kundenteam@IT-Visions.de*
  Telefon *0201 / 64 95 90 – 50*
- Kontakt für Feedback zu diesem Buch:
  www.dotnet-doktor.de/Leserfeedback

5 Über dieses Buch

5.1 Versionsgeschichte dieses Buchs

Die Versionsgeschichte dieses Buch finden Sie in einem eigenen Kapitel am Ende des Buchs.

Hinweis: Die Versionsgeschichte ist eine wichtige Referenz für die Leser, die sich aktuelle Versionen des Buchs beschaffen (z.B. über Leanpub.com) und wissen wollen, was sich geändert hat. Wenn Sie das Buch erstmalig lesen, müssen Sie die Versionsgeschichte nicht lesen.

5.2 Hinweis zu den Vertriebswegen

Dieses Fachbuch wird vertrieben auf folgenden Wegen (Ich nenne neben dem Verkaufspreis auch, wie viel – bzw. wenig – ich als Autor von den jeweiligen Händlern erhalte. Der Rest ist Gewinn der Händler):

- Gedruckt (Print-on-Demand) bei Amazon.de für 39,99 Euro (der Autor erhält 15,38 Euro):
  www.amazon.de/exec/obidos/ASIN/3934279449/itvisions-21
- Kindle-E-Book bei Amazon.de für 29,99 Euro (der Autor erhält 9,81 Euro):
  www.amazon.de/exec/obidos/ASIN/B0CM47LGY8/itvisions-21
- PDF-E-Book **inkl. aller Buch-Updates** bei Leanpub.com ab 29,99 Dollar (der Autor erhält 19,99 Dollar):
  www.leanpub.com/CSharp13
- Als Teil des E-Book-Buch-Abos zusammen mit anderen aktuellen Fachbüchern ab 99,00 Euro im Jahr **inkl. aller Buch-Updates** (der Autor erhält den kompletten Preis):
  www.IT-Visions.de/BuchAbo

Hinweise: Ich habe mich für den Vertriebsweg des gedruckten Buchs über Amazon entschieden, weil ich dort ständig Updates zu dem Buch einreichen kann. Per Print-on-Demand erhalten Leser dann immer das topaktuelle Buch. Oft liefert Amazon dennoch am Tag nach der Bestellung das Buch schon aus. Der Vertrieb dieses Buch über klassische IT-Verlage, die leider heutzutage immer noch größere Auflagen vorproduzieren, ist für ein sehr agiles Softwareprodukt wie C# keine Alternative mehr.

Ich nenne dabei auch den Erlös für den Autor, weil ich sehr häufig Leser treffe, die fälschlicherweise denken, der wesentliche Teil des Buchpreises komme dem Autor zu Gute. Das ist leider nicht so, außer bei Leanpub.com oder eigenen Vertriebswegen wie meinem Buchabo. Daher denke ich, dass es sinnvoll ist, dies transparent zu machen.

5.3 Bezugsquelle des PDF-E-Books für Amazon-Kunden

Wenn Sie dieses Buch in gedruckter Form oder als Kindle-Ausgabe bei Amazon erworben haben, können Sie zusätzlich eine PDF-Version des Buchs **kostenfrei** erhalten.

Leiten Sie dazu Ihren Kaufbeleg von Amazon an folgende E-Mail-Adresse weiter:

PDFBuchZugabe@dotnet-doktor.de

Geben Sie dabei bitte Vorname, Name, Firma und E-Mail-Adresse an.

Sie erhalten dann binnen 1-2 Wochen das auf Sie personalisierte PDF-Dokument. Dieses Angebot gilt innerhalb von 6 Monaten nach dem Kauf des Buchs bei Amazon.

5.4 Bezugsquelle für Aktualisierungen

Sie können jederzeit Aktualisierungen des PDF-Buchs (gleiche Hauptversion!) kostenfrei bei Leanpub.com beziehen.

Käufer der Kindle- oder Druck-Version können die aktuelle PDF-Version zum Preis von 9,99 Dollar (zzgl. 7% Mehrwertsteuer) unter folgender Webadresse beziehen:

https://leanpub.com/CSharp13/c/AWAY

Hinweise: Leider erlauben Amazon u.a. Buchhändler aufgrund der Buchpreisbindungsgesetze in Deutschland den Autoren grundsätzlich nicht, dass Leser eine Aktualisierung im Kindle-Format oder in gedruckter Form vergünstigt erhalten.

Bitte beachten Sie auch, dass die ISBN-Regularien erfordern, dass man bei einer Titeländerung bei neuer Produktversion eine neue ISBN vergeben und damit auch ein neues Buchprojekt bei Amazon und Leanpub erstellt werden muss.

5.5 Hinweise zur Breite und Tiefe dieses Buchs – Sie haben Einfluss!

Ein Fachbuch, das ein riesengroßes Themengebiet wie C# behandelt, kann nicht jedes Teilgebiet und jeden Aspekt der Programmiersprache behandeln, zumindest nicht in gleicher Tiefe. Dann würde solch ein Fachbuch über eintausend Seiten, in einigen Fällen sogar mehrere tausend Seiten umfassen.

Ich denke, dass ich nach aktuellem Stand der Technik und meinem Wissenstand etwa 1.000 Seiten zur C#-Syntax und -Tools sowie 3.000 Seiten zu den C#-Bibliotheken schreiben könnte. Würden Sie so ein dickes (und entsprechend teures) Buch kaufen und lesen wollen?

Wie jeder Fachautor lese auch ich immer wieder Kritik, dass ein Leser ein bestimmtes Thema nicht oder nicht in ausreichender Tiefe behandelt sei in dem Buch. Das ist aus der Sicht des einzelnen Lesers sicherlich gerechtfertigt, aber wie jeder Fachautor muss ich eben zwingend eine Auswahl der Themen treffen. Gerne dokumentiere ich hier, wie ich persönlich diese Auswahl für meine Bücher treffe:

- Ich behandele im Buch die Themen, die wir in unserer Firma selbst in der Praxis brauchen.
- Ich behandele zusätzlich die Themen, die unsere Kunden in Beratungsgesprächen behandelt haben möchten.

Folglich sind die Themen, die ich im Buch nicht oder nur kurz behandele, für uns und unsere Kunden nicht relevant bzw. so selbsterklärend, dass es keine Fragen dazu gibt.

Natürlich kann das für Sie anders sein. Sie können mir immer gerne schreiben, wenn Sie ein Thema im Buch behandelt haben möchten. Ich sammele diese Anregungen und wenn es mehrere Zuschriften zu einem Thema gibt, dann kommt das Thema weit oben auf die Prioritätenliste. Ich denke, das ist ein faires Verfahren.

5.6 Geplante Themen

Folgende Themen sind für kommenden Ausgaben dieses Buchs geplant:

- Aliase für referenzierte Assemblies
- Checked Operators (seit C# 11.0)
- Covariant Return Types (seit C# 9.0)
- Dekompilierung mit ILSpy u.a. [*https://blog.ndepend.com/in-the-jungle-of-net-decompilers*]
- Deployment von modernen .NET-Anwendungen mit dotnet publish
- Extension Method GetEnumerator() (seit C# 9.0)
- Implicit Cast Operator [*learn.microsoft.com/dotnet/csharp/language-reference/keywords/implicit*]
- Inkrementelle Source-Generatoren (seit C# 10.0)
- Interceptoren (experimentell seit C# 12.0)
- Operatoren für Unsigned Right Shift >>> und >>>= (seit C# 11.0)
- Laufzeitcodegenerierung / Refection Emit
- Nullable-Annotationen wie [AllowNull], [DisallowNull], [return: NotNullIfNotNull("xy")], [DoesNotReturn], [return: MaybeNull], MaybeNullWhen(bool), NotNullWhen(bool)
- Ref Fields und ref scoped (seit C# 11.0)
- Span<T> / Memory<T> (seit C# 7.2)
- Statische Codeanalyse
- Unmanaged Constructed Types (seit C# 8.0)
- Visual Studio Code als Alternative zu Visual Studio

Eventuell, wenn der Autor die Zeit findet, kommen irgendwann auch diese über die Sprachsyntax und den Compiler hinausgehenden Themen hinzu:

- Clean Code-Programmierung mit C#
- Design Pattern in C#

5.7 Programmcodebeispiele zu diesem Buch

Die Programmcodebeispiele zu diesem Buch können Sie auf der auf der von mir ehrenamtlich betriebenen Leserwebsite *www.IT-Visions.de/Leser* herunterladen. Dort müssen Sie sich registrieren. Bei der Registrierung wird ein Losungswort abgefragt. Bitte geben Sie dort das Losungswort **AWAY** ein.

Alle Programmbeispiele aus diesem Buch sind in einer Visual Studio 2022-Projektmappe mit zwei Projekten enthalten. Es muss seit C# 8.0 zwei Projekte geben, weil einige Sprachfeatures von C# 8.0 nicht mehr im klassischen .NET Framework laufen und C# seit Version 9.0 gar nicht mehr dort läuft. Die beiden Projekte enthalten:

- CSharpSprachsyntax_NETClassic (.NET Framework 4.8): Alle Sprachfeatures von C# 1.0 bis 7.3 und solche von C# 8.0, die auch auf klassischen .NET Framework laufen
- CSharpSprachsyntax_NET (.NET 6.0): Alle Sprachfeatures von C# 8.0, die NICHT auf .NET Framework laufen sowie alle Sprachfeatures **ab** C# 9.0

Die Beispiele sind in Unterordnern nach Sprachversionen aufgeteilt. Dies heißt, dass Sie zum Beispiel Sprachfeatures von C# 12.0 im Ordner CS120 finden bzw. C# 11.0 in CS110.

Wie im Vorwort bereits erwähnt handelt es sich um den Anwendungstyp "Konsolenanwendung". So brauchen Sie als Leser kein Wissen über irgendeine GUI-Bibliothek und die Beispiele sind prägnant fokussiert auf die Syntax. Bitte beachten Sie das nächste Kapitel zum Hilfsklasse "CUI".

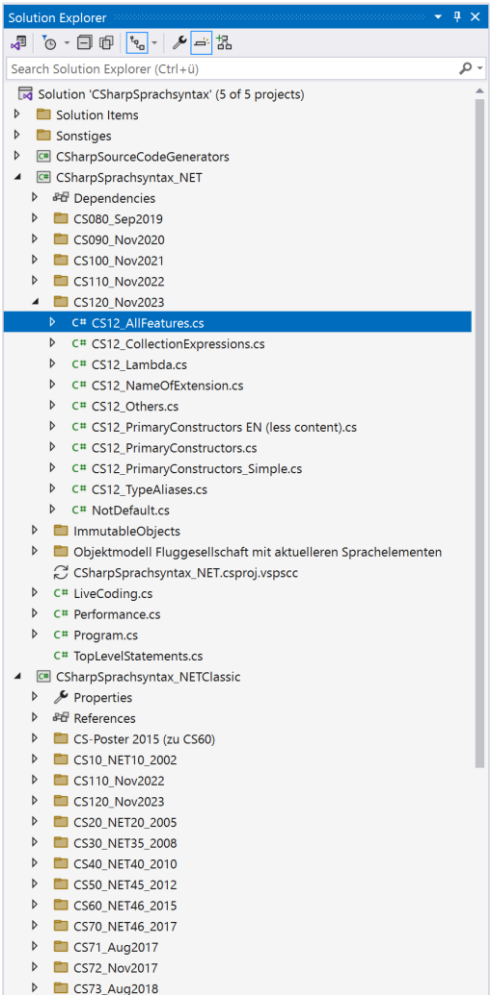


Abbildung: Programmcodebeispiele zu diesem Buch in zwei Visual Studio-Konsolenanwendungen (EXE) plus Hilfsbibliotheken (DLLs)

5.8 Hilfsklasse zur Konsolenausgabe (CUI)

Für die Bildschirmausgabe an der Konsole wird in diesem Buch oft nicht nur Console.WriteLine() verwendet, sondern auch Hilfsroutinen kommen zur Anwendung, die farbige Bildschirmausgaben erzeugen. Diese Hilfsroutinen sind in der Klasse *ITVisions.CUI* (CUI besteht dabei für Commandline User Interface) implementiert. Diese Klasse ist Teil des NuGet-Pakets ITV.AppUtil…nupkg, welches bei den herunterladbaren Projekten zu diesem Buch in Form mitgeliefert und via <packageSource> in der Datei NuGet.config einbezogen wird.

Diese wichtigsten Hilfsroutinen in der Klasse *CUI* sind im Folgenden zum besseren Verständnis abgedruckt.

Listing: Klasse CUI mit Hilfsroutinen für die Bildschirmausgabe an der Konsole

```
using System;
using System.Runtime.InteropServices;
using System.Web;
using ITVisions.UI;
using System.Diagnostics;

namespace ITVisions
{
 /// <summary>
 /// Helper utilities for console UIs
 /// (C) Dr. Holger Schwichtenberg 2002-2018
 /// </summary>
 public static class CUI
 {
  public static bool IsDebug = false;
  public static bool IsVerbose = false;

  #region Print only under certain conditions
 public static void PrintDebug(object s)
  {
   PrintDebug(s, System.Console.ForegroundColor);
  }

  public static void PrintVerbose(object s)
  {
   PrintVerbose(s, System.Console.ForegroundColor);
  }
  #endregion

  #region Issues with predefined colors
  public static void MainHeadline(string s)
  {
   Print(s, ConsoleColor.Black, ConsoleColor.Yellow);

  }
  public static void Headline(string s)
  {
   Print(s, ConsoleColor.Yellow);
  }
  public static void HeaderFooter(string s)
```

```
{
 Console.ForegroundColor = ConsoleColor.Green;
 Console.WriteLine(s);
 Console.ForegroundColor = ConsoleColor.Gray;
}

public static void SubHeadline(string s)
{
 Print(s, ConsoleColor.White);
}

public static void PrintSuccess(object s)
{
 Print(s, ConsoleColor.Green);
}

public static void H1(string s)
{
 MainHeadline(s);
}

public static void H2(string s)
{
 Headline(s);
}

public static void H3(string s)
{
 SubHeadline(s);
}

public static void PrintGreen(string s)
{
 Print(s, ConsoleColor.Green);
}

public static void PrintYellow(string s)
{
 Print(s, ConsoleColor.Yellow);
}

public static void PrintRed(string s)
{
 Print(s, ConsoleColor.Red);
}

public static void PrintSuccess(object s)
{
 Print(s, ConsoleColor.Green);
}

public static void PrintStep(object s)
{
```

```
   Print(s, ConsoleColor.Cyan);
  }

  public static void PrintDebugSuccess(object s)
  {
   PrintDebug(s, ConsoleColor.Green);
  }

  public static void PrintVerboseSuccess(object s)
  {
   PrintVerbose(s, ConsoleColor.Green);
  }

  public static void PrintWarning(object s)
  {
   Print(s, ConsoleColor.Cyan);
  }

  public static void PrintDebugWarning(object s)
  {
   PrintDebug(s, ConsoleColor.Cyan);
  }

  public static void PrintVerboseWarning(object s)
  {
   PrintVerbose(s, ConsoleColor.Cyan);
  }

  public static void PrintError(object s)
  {
   Print(s, ConsoleColor.White, ConsoleColor.Red);
  }

  public static void PrintDebugError(object s)
  {
   PrintDebug(s, ConsoleColor.White, ConsoleColor.Red);
  }

  public static void PrintVerboseError(object s)
  {
   Print(s, ConsoleColor.White, ConsoleColor.Red);
  }

  public static void Print(object s)
  {
   PrintInternal(s, null);
  }
  #endregion

  #region Print with selectable color

  public static void Print(object s, ConsoleColor farbe, ConsoleColor?
hintergrundfarbe = null)
```

```
  {
   PrintInternal(s, farbe, hintergrundfarbe);
  }

  public static void PrintDebug(object s, ConsoleColor farbe, ConsoleColor?
hintergrundfarbe = null)
  {
   if (IsDebug || IsVerbose) PrintDebugOrVerbose(s, farbe, hintergrundfarbe);
  }

  public static void PrintVerbose(object s, ConsoleColor farbe)
  {
   if (!IsVerbose) return;
   PrintDebugOrVerbose(s, farbe);
  }
  #endregion

  #region Print with additional data

  /// <summary>
  /// Print with Thread-ID
  /// </summary>
  public static void PrintWithThreadID(string s, ConsoleColor c =
ConsoleColor.White)
  {
   var ausgabe = String.Format("Thread #{0:00} {1:}: {2}",
System.Threading.Thread.CurrentThread.ManagedThreadId,
DateTime.Now.ToLongTimeString(), s);
   CUI.Print(ausgabe, c);
  }

  /// <summary>
  ///  Print with time
  /// </summary>
  public static void PrintWithTime(object s, ConsoleColor c = ConsoleColor.White)
  {
   CUI.Print(DateTime.Now.Second + "." + DateTime.Now.Millisecond + ":" + s);
  }

  private static long count;
  /// <summary>
  /// Print with counter
  /// </summary>
  private static void PrintWithCounter(object s, ConsoleColor farbe,
ConsoleColor? hintergrundfarbe = null)
  {
   count += 1;
   s = $"{count:0000}: {s}";
   CUI.Print(s, farbe, hintergrundfarbe);
  }

  #endregion

  #region internal helper routines
```

```
  private static void PrintDebugOrVerbose(object s, ConsoleColor farbe,
ConsoleColor? hintergrundfarbe = null)
  {
   count += 1;
   s = $"{count:0000}: {s}";
   Print(s, farbe, hintergrundfarbe);
   Debug.WriteLine(s);
   Trace.WriteLine(s);
   Trace.Flush();
  }

  /// <summary>
  /// Output to console, trace and file
  /// </summary>
  /// <param name="s"></param>
  [DebuggerStepThrough()]
  private static void PrintInternal(object s, ConsoleColor? farbe = null,
ConsoleColor? hintergrundfarbe = null)
  {
   if (s == null) return;

   if (HttpContext.Current != null)
   {
    try
    {
 if (farbe != null)
     {
      HttpContext.Current.Response.Write("<span style='color:" +
farbe.Value.DrawingColor().Name + "'>");
     }
     if (!HttpContext.Current.Request.Url.ToString().ToLower().Contains(".asmx")
&& !HttpContext.Current.Request.Url.ToString().ToLower().Contains(".svc") &&
!HttpContext.Current.Request.Url.ToString().ToLower().Contains("/api/"))
HttpContext.Current.Response.Write(s.ToString() + "<br>");

     if (farbe != null)
     {
      HttpContext.Current.Response.Write("</span>");
     }
    }
    catch (Exception)
    {
    }
   }
   else
   {
    object x = 1;
    lock (x)
    {
     ConsoleColor alteFarbe = Console.ForegroundColor;
     ConsoleColor alteHFarbe = Console.BackgroundColor;

     if (farbe != null) Console.ForegroundColor = farbe.Value;
```

```
    if (hintergrundfarbe != null) Console.BackgroundColor =
hintergrundfarbe.Value;

    //if (farbe.ToString().Contains("Dark")) Console.BackgroundColor =
ConsoleColor.White;
    //else Console.BackgroundColor = ConsoleColor.Black;

    Console.WriteLine(s);
    Console.ForegroundColor = alteFarbe;
    Console.BackgroundColor = alteHFarbe;
   }
  }
 }
 #endregion

 #region Set the position of the console window
 [DllImport("kernel32.dll", ExactSpelling = true)]
 private static extern IntPtr GetConsoleWindow();
 private static IntPtr MyConsole = GetConsoleWindow();

 [DllImport("user32.dll", EntryPoint = "SetWindowPos")]
 public static extern IntPtr SetWindowPos(IntPtr hWnd, int hWndInsertAfter, int
x, int Y, int cx, int cy, int wFlags);

 // Set the position of the console window without size
 public static void SetConsolePos(int xpos, int ypos)
 {
  const int SWP_NOSIZE = 0x0001;
  SetWindowPos(MyConsole, 0, xpos, ypos, 0, 0, SWP_NOSIZE);
 }

 // Set the position of the console window with size
 public static void SetConsolePos(int xpos, int ypos, int w, int h)
 {
  SetWindowPos(MyConsole, 0, xpos, ypos, w, h, 0);
 }
 #endregion
}
}
```

5.9 Qualitätssicherung der Programmcodebeispiele

Ich versichere Ihnen, dass die Programmcodebeispiele auf zwei meiner Entwicklungssysteme kompilierten und liefen, bevor ich sie per Kopieren & Einfügen in das Manuskript zu diesem Buch übernommen habe und auf der Leser-Website zum Download veröffentlicht habe.

Dennoch gibt es leider Gründe, warum die Beispiele bei Ihnen als Leser nicht laufen:

- Eine abweichende Systemkonfiguration (in der heutigen komplexen Welt der vielen Varianten und Versionen von Betriebssystemen und Anwendungen nicht unwahrscheinlich). Es ist einem Autor nicht möglich, alle Konfigurationen durchzutesten.

- Änderungen, die sich seit der Erstellung der Beispiele ergeben haben (von den vielen Breaking Changes, die die neueren .NET-Versionen immer wieder durch Microsoft erhalten, können auch Beispiele betroffen sein, was nicht immer leicht zu entdecken ist)
- Schließlich sind auch menschliche Fehler des Autors möglich. Bitte bedenken Sie, dass das Fachbuchschreiben – wie im Vorwort erwähnt – nur ein Hobby ist. Es gibt nur sehr wenige Menschen in Deutschland, die hauptberuflich als Fachbuchautor arbeiten und so professionell Programmcodebeispiele erstellen und testen können wie kommerziellen (bezahlten) Programmcode.

Falls dennoch Beispiele bei Ihnen nicht laufen, kontaktieren Sie mich bitte unter

www.dotnet-doktor.de/Leserfeedback

mit einer sehr genauen Fehlerbeschreibung. Ich bemühe mich, Ihnen binnen zwei Wochen zu antworten. Im Einzelfall kann es wegen dienstlicher oder privater Abwesenheit aber auch länger dauern.

5.10 Ihre Belohnung, wenn Sie helfen, dieses Buch zu verbessern!

Wenn Sie Fehler in diesem Buch finden, bin ich Ihnen nicht nur wirklich sehr dankbar, sondern Sie bekommen auch eine Belohnung in Form von aktualisierten oder weiteren E-Books.

Fehlerart	E-Book-Guthaben
Inhaltlicher Fehler	Pro Fehler 5 Euro
Sprachlicher Fehler	Pro Fehler 2 Euro

Ein Beispiel: Wenn Sie zwei inhaltliche Fehler und zehn Rechtschreibfehler in diesem Buch finden, dann haben Sie bei mir 30 Euro gut. Dafür können Sie dann eins meiner selbstverlegten Bücher als E-Book bekommen.

Die selbstverlegten Bücher finden Sie unter *www.IT-Visions.de/Verlag*

Melden Sie die Fehler unter *www.dotnet-doktor.de/Leserfeedback*

Schreiben Sie dabei, welches E-Book Sie wünschen. Das Buch schicke ich Ihnen dann per E-Mail zu.

Tipp: Auch Fehler auf meiner persönlichen Website *www.dotnet-doktor.de* und der Firmenwebsite *www.IT-Visions.de* zählen mit!

Ich freue mich auf Ihre Fehlermeldung!

Holger Schwichtenberg

P.S. Die Fehlermeldung zählt nur, wenn nicht ein anderer Leser dies bereits gemeldet hat und es daher in der aktuellen Auflage schon korrigiert ist.

6 Fakten zu C#

6.1 Der Name C#

C# wird gesprochen „C Sharp“. Das # könnte man auch in ein vierfaches Pluszeichen aufspalten (also C++++, eine Weiterentwicklung von C++). Ursprünglich sollte die Sprache "Cool" heißen. Eine Zeit lang wurde auch "C# .NET" verwendet; das ist heute aber nicht mehr üblich. Microsoft spricht aber gelegentlich noch von "Visual C#", z.B. meldet sich der Kommandozeilencompiler von C# auch in der aktuellen Version mit "Microsoft (R) Visual C# Compiler".

6.2 Ursprünge von C#

C# ist das Ergebnis eines Projektes bei Microsoft, welches im Dezember 1998 gestartet wurde, nachdem die Firma Sun Microsoft die Veränderung der von Sun entwickelten Programmiersprache Java verboten hatte. Vater von C# ist Anders Hejlsberg [*de.wikipedia.org/wiki/Anders_Hejlsberg*], der zuvor auch Turbo Pascal und Borland Delphi erschaffen hat. Er war früher bei Borland und arbeitet seit 1996 bei Microsoft. Heutzutage ist er auch verantwortlich für die Sprache TypeScript.

6.3 .NET als Basis für C#

Die Programmiersprache C# ist sehr eng verbunden mit der Softwareentwicklungsplattform Microsoft .NET. C#-Programmcode läuft immer auf Basis einer .NET-Laufzeitumgebung und benötigt Klassen aus der .NET-Basisklassenbibliothek. So besitzt C# selbst keine Datentypen: Alle Datentypen, die man in C# verwendet, z.B. string, sind in Wirklichkeit Klassen aus der .NET-Basisklassenbibliothek (string → System.String). Auch andere Sprachkonstrukte in C# basieren auf Schnittstellen und Klassen der .NET-Basisklassenbibliothek, z.B. foreach(…) { … } basiert auf der Schnittstelle System.Collections.IEnumerable und await foreach(…) { … } basiert auf System.Collections.Generic.IAsyncEnumerable<T>. Der Range-Operator (1..10) erfordert die Klasse System.Range usw.

Im Laufe der Geschichte von .NET (seit dem Jahr 2001) gab es zahlreiche Implementierungen von .NET (.NET Framework, Mono, .NET Compact Framework, .NET Framework Client Profile, .NET Micro Framework, Silverlight, XNA, .NET Profile für Windows Runtime, .NET Core, Universal Windows Platform). Derzeit sind noch in signifikantem Umfang in Einsatz:

- .NET Framework
- .NET Core
- Universal Windows Platform (UWP)
- Mono/Xamarin
- .NET ab Version 5.0

Hinweis: Mit .NET 6.0 führt Microsoft diese Implementierungen zu einer einheitlichen Plattform zusammen. Alle anderen Implementierungen werden nicht mehr entwickelt.

Zumindest das ".NET Framework" wird aber noch viele Jahre eine Bedeutung im Markt haben, weil Microsoft dafür zumindest noch Updates im Bereich Fehlerbehebung, Zuverlässigkeit und Sicherheit liefert. Für alle anderen Implementierungen wird auch dieser Support bald enden.

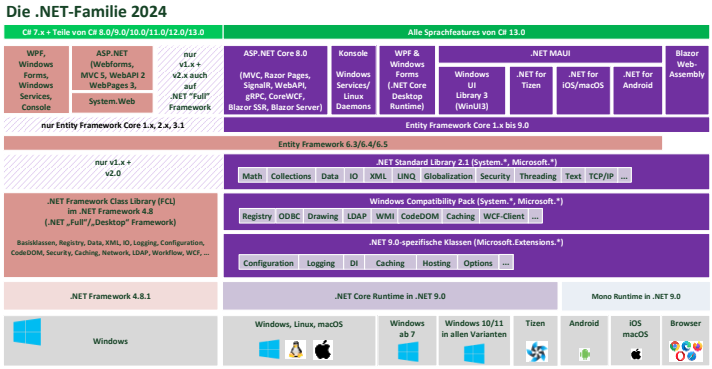


Abbildung: Die .NET-Familie mit .NET Framework 4.8 und .NET 9.0

6.4 Status der Programmiersprache C#

Früher gab es einen wahren Glaubenskrieg in der .NET-Entwicklergemeinde um die Wahl der »richtigen« Programmiersprache. C# oder Visual Basic .NET hieß die Frage, die viele Projektteams bewegt hat. Auch wenn Visual Basic .NET in allen wesentlichen Punkten syntaktisch ebenbürtig war, hat C# klar gewonnen.

C# ist heute nicht nur eine von vielen Programmiersprachen für .NET, es hat sich durchgesetzt als DIE Programmiersprache für .NET. Gegenwärtig gibt es nur noch wenige .NET-Projekte, die Visual Basic .NET, F# oder C++/CLI oder exotischere Sprachen verwenden.

Während früher viele .NET-Fachbücher in zwei verschiedenen Editionen zu C# und Visual Basic .NET erschienen sind, gibt es heutzutage nur noch eine Variante zu C#.

In der Dokumentation der .NET-Klassenbibliothek gibt es aber mittlerweile neben C# auch wieder Beispiele in Visual Basic .NET,

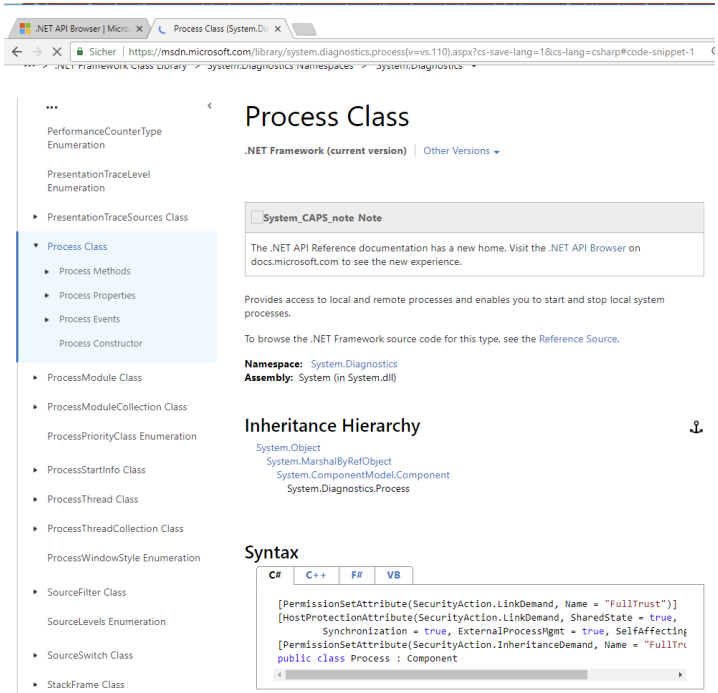


Abbildung: Beispiele in vier Sprachen in der alten MSDN-Dokumentation der .NET-Klassen in verschiedenen Registerkarten

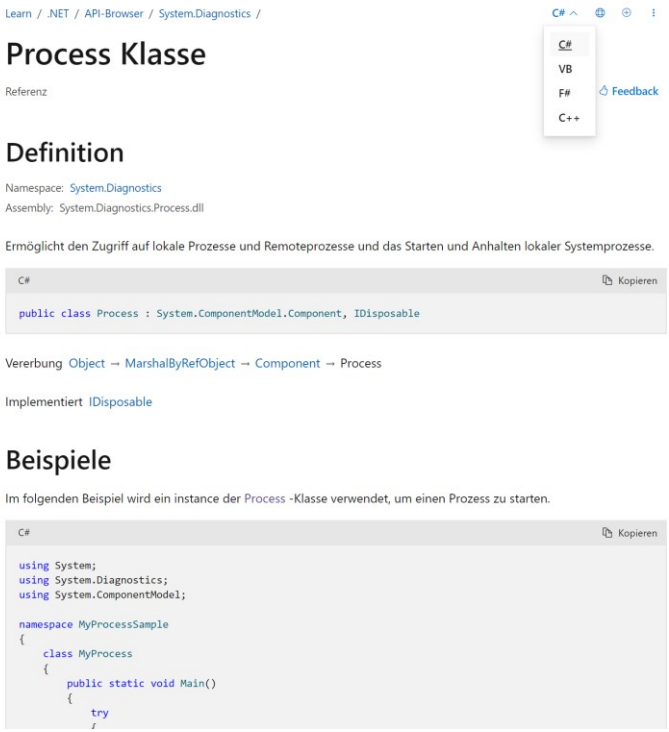


Abbildung: Neue Dokumentation auf learn.microsoft.com mit Auswahl der Sprachen per Dropdown oben auf der Seite

6.5 Dokumentation zu C#

Die offizielle Dokumentation zu C# finden Sie unter

https://learn.microsoft.com/en-us/dotnet/csharp

Weitere Dokumentation finden Sie in zwei GitHub-Projekten:

https://github.com/dotnet/csharplang

https://github.com/dotnet/roslyn

6.6 Versionsgeschichte

Hinsichtlich der Versionsnummern der Sprache C# herrschte früher etwas Verwirrung. Es gab einerseits eine offizielle Zählung mit Versionsnummer (parallel zum .NET Framework), andererseits mit Jahreszahlen (parallel zu Visual Studio). Intern wird eine dritte Zählung für den Compiler verwendet. Die erste Version von C# im Rahmen des .NET Framework 1.0 trug intern die Versionsnummer 7.0. Zu .NET 1.1 gab es dann C# 7.1, im .NET Framework 2.0 und 3.0 meldet sich der C#-Compiler mit Version 8.0. Ab .NET Framework 3.5 hat Microsoft dies aber bereinigt. Dort meldet sich der Compiler nun auch mit Version 3.5.

Die folgende Liste dokumentiert die Versionsgeschichte von C# einschließlich der verschiedenen Namen, die es jeweils gibt.

- C# 1.0 ist erschienen am 05.01.2002 (in Visual Studio.NET 2002+2003 / .NET Framework 1.0 und 1.1. Erste Version des ISO-Standards für C#.)
- C# 2.0 ist erschienen am 07.11.2005 (C# 2005 / in Visual Studio.NET 2005 / .NET Framework 2.0 und 3.0. Zweite Version des ISO-Standards für C#.)
- C# 3.0 ist erschienen am 15.08.2008 (C# 2008 / in Visual Studio.NET 2008 / .NET Framework 3.5)
- C# 4.0 ist erschienen am 12.04.2010 (C# 2010 / in Visual Studio.NET 2010 / .NET Framework 4.0)
- C# 5.0 ist erschienen am 12.08.2012 (C# 2012 / in Visual Studio.NET 2012 / .NET Framework 4.5)
- C# 6.0 ist erschienen am 20.07.2015 (C# 2015 / in Visual Studio.NET 2015 / .NET Framework 4.6)
- C# 7.0 ist erschienen am 05.03.2017 (C# 2017 / in Visual Studio 2017 v15.0)
- C# 7.1 ist erschienen am 14.08.2017 (in Visual Studio 2017 v15.3)
- C# 7.2 ist erschienen am 15.11.2017 (in Visual Studio 2017 v15.5)
- C# 7.3 ist erschienen am 02.08.2018 (in Visual Studio 2017 v15.7)
- C# 8.0 ist erschienen am 23.09.2019 (in Visual Studio 2019 v16.3)
- C# 9.0 ist erschienen am 10.11.2020 (in Visual Studio 2019 v16.8)
- C# 10.0 ist erschienen am 08.11.2021 (in Visual Studio 2022, v17.0)
- C# 11.0 ist erschienen am 08.11.2022 (in Visual Studio 2022, v17.4)
- C# 12.0 ist erschienen am 14.11.2023 (in Visual Studio 2022, v17.8)
- C# 13.0 ist erschienen am 12.11.2024 (in Visual Studio 2022, v17.12)

Version der Sprachsyntax mit Versionsnummer	Ausgeliefert mit	Version der Sprachsyntax mit Jahreszahl	Interne Versionsnummer des C#-Compilers
C# 1.0	.NET Framework 1.0	Visual C# 2002	7.0 (alter Compiler)
C# 1.1	.NET Framework 1.1	Visual C# 2003	7.1 (alter Compiler)
C# 2.0	.NET Framework 2.0	Visual C# 2005	8.0 (alter Compiler)
C# 2.0	.NET Framework 3.0	Visual C# 2005	8.0 (alter Compiler)
C# 3.0	.NET Framework 3.5	Visual C# 2008	3.5 (alter Compiler)

Version der Sprachsyntax mit Versionsnummer	Ausgeliefert mit	Version der Sprachsyntax mit Jahreszahl	Interne Versionsnummer des C#-Compilers
C# 4.0	.NET Framework 4.0	Visual C# 2010	4.0 (alter Compiler)
C# 5.0	.NET Framework 4.5	Visual C# 2012	4.5 (alter Compiler)
C# 6.0	.NET Framework 4.6 / .NET Core 1.0	Visual C# 2015	1.x (Neuer Compiler)
C# 7.0	Visual Studio 2017 15.0 / .NET Core 2.0	Visual C# 2017	2.0 (Neuer Compiler)
C# 7.1	Visual Studio 2017 15.4 / .NET Core 2.0	Visual C# 2017	2.3 (Neuer Compiler)
C# 7.2	Visual Studio 2017 15.5 / .NET Core 2.0	Visual C# 2017	2.7 (Neuer Compiler)
C# 7.3	Visual Studio 2017 15.7 / .NET Core 2.1	Visual C# 2017	2.8 + 2.9 + 2.10 (Neuer Compiler)
C# 8.0	Visual Studio 2019 16.3 / .NET Core 3.x	Visual C# 2018	3.3 bis 3.7 (Neuer Compiler)
C# 9.0	Visual Studio 2019 16.8 / .NET 5.0	Visual C# 2020	ab v3.8 (Neuer Compiler)
C# 10.0	Visual Studio 2022 17.0 / .NET 6.0	Visual C# 2022	ab v4.0 (Neuer Compiler)
C# 11.0	Visual Studio 2022 17.4 / .NET 7.0	Visual C# 2023	ab v4.4 (Neuer Compiler)
C# 12.0	Visual Studio 2022 17.8 / .NET 8.0	Visual C# 2023	ab v4.8 (Neuer Compiler)
C# 13.0	Visual Studio 2022 17.12 / .NET 9.0	Visual C# 2024	ab v4.11 (Neuer Compiler)

Tabelle: Verschiedene Versionsnummernzählungen für die Sprache C#

6.7 Standardisierung

Microsoft hat einige Teile des .NET Framework unter dem Namen Common Language Infrastructure (CLI) standardisieren lassen. Die CLI wurde erstmals im Dezember 2001 von der European Computer Manufacturers Association (ECMA) standardisiert (ECMA-Standard 335, Arbeitsgruppe TC49 / TG3, früher: TC39 / TG3, siehe [ECMA01]); mit kleinen Änderungen wurde der Standard im Dezember 2002 von der weltweit wichtigsten Standardisierungsorganisation, der International Standardization Organization (ISO), übernommen als ISO / IEC 23271.

Die Begriffe lauten in den Standards zum Teil allerdings anders als bei Microsoft: Was im .NET Framework Microsoft Intermediate Language (MSIL) heißt, entspricht im Standard der Common Intermediate Language (CIL). Anstelle der Framework Class Library (FCL) spricht man von der CLI Class Library. Von der Standardisierung ausgenommen sind jedoch z.B. die

Datenbankschnittstelle ADO.NET und die Benutzeroberflächen-Bibliotheken Windows Forms und ASP.NET Webforms. Auch die neueren .NET-Bibliotheken (WPF, WCF und WF) sind nicht standardisiert.

Auch die Programmiersprache C# ist von beiden Gremien akzeptiert (ECMA-334 bzw. ISO / IEC 23270). Die Standardisierung bezieht sich aber auf ältere Versionen. Die letzten C#-Versionen hat Microsoft nicht mehr standardisieren lassen. Die Standardisierung von C# ist alllerdings auf dem Stand C# 6.0 stehengeblieben [*www.ecma-international.org/publications-and-standards/standards/ecma-334/*].

MICROSOFT VISUAL C# VERSION	CORRESPONDING ECMA STANDARD	CORRESPONDING ISO/IEC STANDARD
V1.0	ECMA-334:2003	ISO/IEC 23270:2003
V2.0	ECMA-334:2006	ISO/IEC 23270:2006
V3.0	none	none
V4.0	none	none
V5.0	ECMA-334:2017	ISO/IEC 23270:2018
V6.0	ECMA-334:2022	none
V7.0	TBD	TBD

Abbildung: Standard der C#-Standardisierung [Quelle: www.ecma-international.org/publications-and-standards/standards/ecma-334, Stand: 29.10.2023]

Ein weiterer, von Microsoft initiierter Standard ist von der ECMA im Dezember 2005 unter ECMA-372 (Arbeitsgruppe TC49 / TG5, früher: TC39 / TG5) verabschiedet worden: C++ / CLI ist eine Spracherweiterung für C++ (ISO / IEC 14882:2003), die eine elegantere Nutzung von C++ auf der CLI-Plattform ermöglicht, als dies bisher mit den Managed Extensions for C++ (alias Managed C++) möglich war.

6.8 Implementierung des C#-Compilers

Die ursprüngliche Version des C#-Compilers (csc.exe) wurde in C++ implementiert. Auch der C#-Compiler im Mono-Projekt ist in C++ geschrieben.

Mit dem Projekt "Roslyn" (alias: .NET Compiler Platform) hat Microsoft selbst den Compiler neu in C# implementiert. Die erste Version des neuen Compilers war C# 6.0.

6.9 Open Source

Bereits zu C# 1.0 gab es eine quelloffene Version im Projekt "Rotor" im Rahmen der Standardisierung von C#. Diese war jedoch nicht "Open Source", sondern nur "Shared Source", d.h. der Quellcode durfte betrachtet, aber nicht weiterverwendet werden. Seit C# 6.0 ist der neue Compiler im Rahmen der .NET Compiler Platform "Roslyn" ein Open Source-Projekt auf Github.

Projekt für das Design der Programmiersprache:

github.com/dotnet/csharplang

Projekt für die Implementierung der Programmiersprache:

github.com/dotnet/roslyn

6.10 Parität und Co-Evolution mit Visual Basic .NET

Im Jahr 2010 hatte Microsoft verkündet, die Programmiersprache C# und Visual Basic .NET hinsichtlich ihrer Funktionalität anzugleichen. »Die Sprachen sollen sich in Stil und Gefühl unterscheiden, nicht in ihrem Funktionsumfang«, schrieb Mads Torgersen, Produktmanager für C# damals. Scott Wiltamuth führt den Begriff "Co-Evolution" ein [*blogs.msdn.microsoft.com/scottwil/2010/03/09/vb-and-c-coevolution*].

Einige Jahre hat Microsoft diese Strategie tatsächlich umgesetzt und bestehende Sprachfeatures, die nur eine Sprache hatte, in der anderen Sprache nachgerüstet und neue Sprachfeatures gleichzeitig oder zumindest zeitnah in beiden Sprachen veröffentlicht.

Im Jahr 2017 hat Microsoft sich von Parität und Co-Evolution wieder verabschiedet. Die parallel zu C# 7.0 erschienene Version 15 von Visual Basic .NET bietet daher lediglich Tupel und binäre Literale als neue Sprachfeatures an. Zudem kann Visual Basic .NET 15 C#-Methoden nutzen, die Zeiger mit ref liefern, selbst aber solche Methoden nicht implementieren.

Im März 2020 hat Microsoft verkündet, die Programmiersprache Visual Basic .NET hinsichtlich der Syntax nicht mehr weiter zu entwickeln, diese Sprache aber zumindest bei einigen Projektarten in .NET weiterhin zu unterstützen [*devblogs.microsoft.com/vbteam/visual-basic-in-net-core-3-0/*]. Zentrale Aussagen darin waren:

- "Going forward, we do not plan to evolve Visual Basic as a language."
- "Future features of .NET Core that require language changes may not be supported in Visual Basic. "
- "Due to differences in the platform, there will be some differences between Visual Basic on .NET Framework and .NET Core."

Visual Basic .NET ist dennoch nach C# weiterhin die zweitwichtigste Programmiersprache in der .NET-Welt. Telemetriedaten [*blogs.msdn.microsoft.com/dotnet/2017/02/01/the-net-language-strategy*] von Microsoft zeigen einerseits, dass Visual Basic .NET hauptsächlich zur Programmierung mit älteren .NET-Techniken wie Windows Forms und ASP.NET Webforms zum Einsatz kommt. Andererseits beginnen viele neue .NET-Entwickler mit Visual Basic .NET, bevor sie sich an C# herantrauen.

6.11 Popularität von C#

Für die Beliebtheit von Programmiersprachen gibt es verschiedene Erhebungen. Sehr beliebt ist der Tiobe Index [*www.tiobe.com/tiobe-index*], der monatlich durch eine Auswertung von Internetseiten ermittelt wird. Hier liegt C# in der Regel seit längerem auf Platz 5, hinter Python, C++, C und Java. Knapp hinter C# liegt Visual Basic .NET, hier nur als "Visual Basic" bezeichnet, aber abzugrenzen von "Visual Basic Classic" auf Platz 22 (hier nicht mehr im Bild).

Oct 2024	Oct 2023	Change	Programming Language	Ratings	Change
1	1		Python	21.90%	+7.08%
2	3	↑	C++	11.60%	+0.93%
3	4	↑	Java	10.51%	+1.59%
4	2	↓	C	8.38%	-3.70%
5	**5**		**C#**	**5.62%**	**-2.09%**
6	6		JavaScript	3.54%	+0.64%
7	7		Visual Basic	2.35%	+0.22%
8	11	↑	Go	2.02%	+0.65%
9	16	⇈	Fortran	1.80%	+0.78%
10	13	↑	Delphi/Object Pascal	1.68%	+0.38%
11	9	↓	SQL	1.64%	-0.15%
12	14	↑	MATLAB	1.48%	+0.22%
13	20	⇈	Rust	1.45%	+0.53%
14	12	↓	Scratch	1.41%	+0.05%
15	8	⇊	PHP	1.21%	-0.69%
16	10	⇊	Assembly language	1.13%	-0.51%
17	17		R	1.09%	+0.12%
18	19	↑	Ruby	0.99%	+0.07%
19	24	⇈	COBOL	0.99%	+0.23%
20	15	⇊	Swift	0.98%	-0.09%

Abbildung: Beliebtheit der Programmiersprachen (Quelle: www.tiobe.com/tiobe-index)

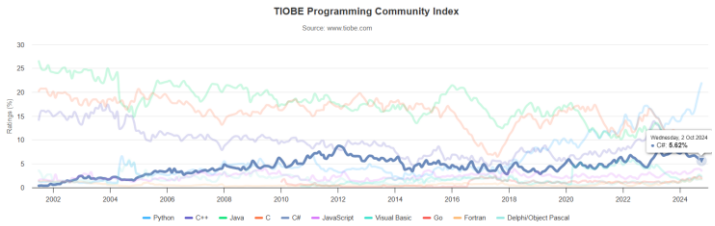


Abbildung: Beliebtheit von C# von 2002 bis 2024 (Quelle: www.tiobe.com/tiobe-index)

Das Ranking der IEEE (Institute of Electrical and Electronics Engineers) basiert auf der Auswertung mehrerer Datenquellen (CareerBuilder, GitHub, Google, Hacker News, IEEE, Reddit, Stack Overflow und Twitter).

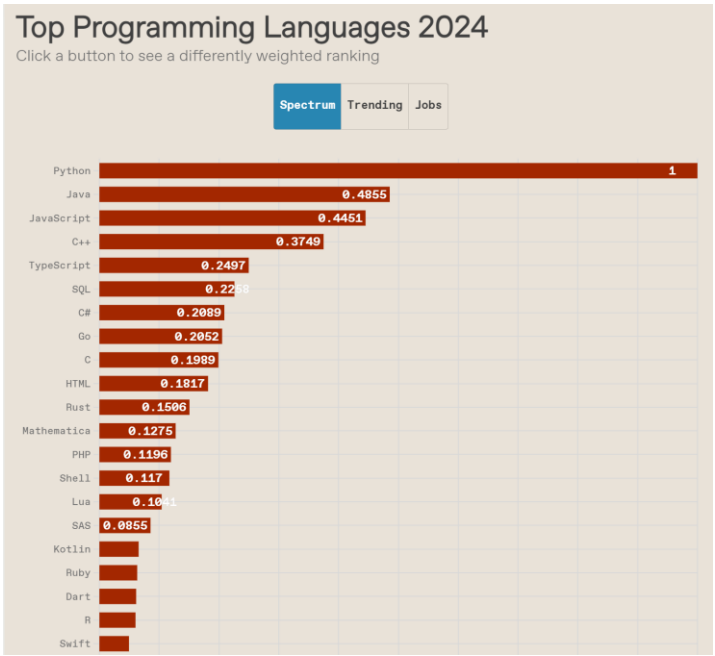


Abbildung: IEEE-Ranking 2023 [spectrum.ieee.org/top-programming-languages/#toggle-gdpr] ("The "Spectrum" ranking is weighted towards the profile of the typical IEEE member, the "Trending" ranking seeks to spot languages that are in the zeitgeist, and the "Jobs" ranking measures what employers are looking for.")

Auch das IT-Marktforschungsunternehmen RedMonk liefert ein Programmiersprachenranking basierend auf GitHub und Stackoverflow.com. C# liegt dort zusammen mit C++ und CSS auf Platz 5. Davor sind JavaScript, Python, Java und PHP.

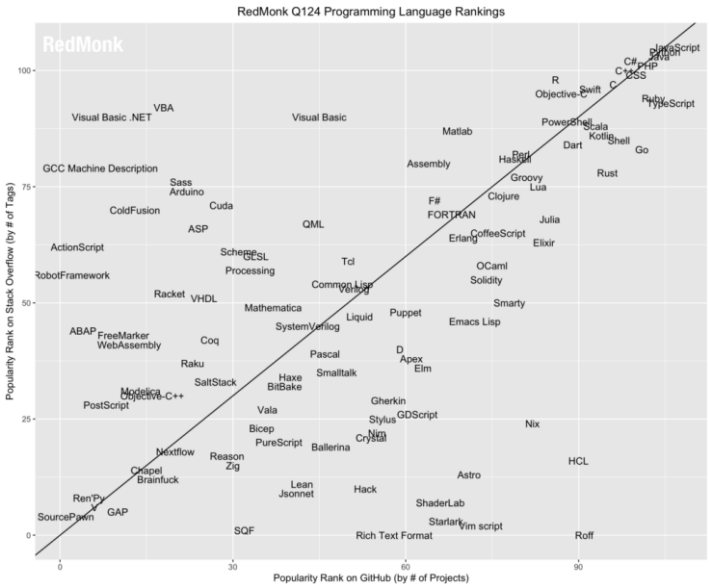


*Abbildung: Programmiersprachen-Ranking von RedMonk, Stand Januar 2024: Diagramm korreliert GitHub-Pull-Requests (x-Achse) zum Rang bei Stack Overflow (y-Achse) [https://redmonk.com/sogrady/2024/03/08/language-rankings-1-24*https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/*]*

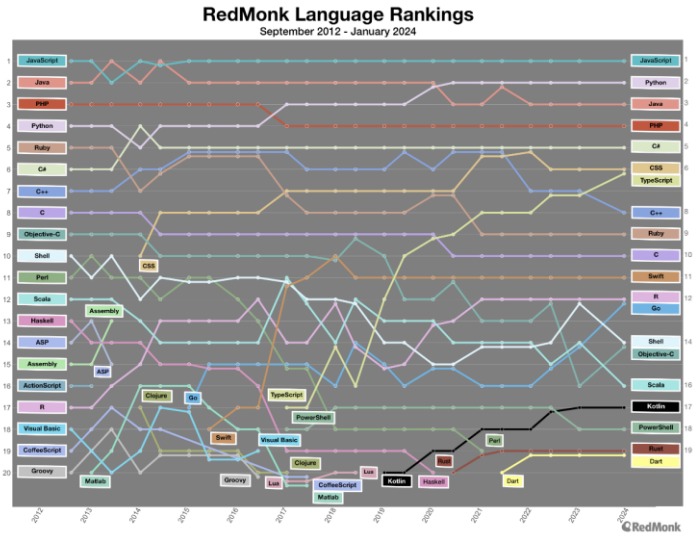


Abbildung: Jahresauswertungen von RedMonk 2012 bis 2024
[https://redmonk.com/rstephens/2024/03/08/top20-jan2024]

Seit dem Jahr 2017 gibt es eine Umfrage "The State of Developer Ecosystem" der Firma JetBrains. C# liegt im Jahr 2023 bei der Beliebtheit auf Platz 9, mit 21% weit hinter den Webtechniken wie JavaScript, TypeScript und HTML/CSS. Auch Python, SQL, Java, Shell-Sprachen und sogar C++ sind bei der Umfrage beliebter.

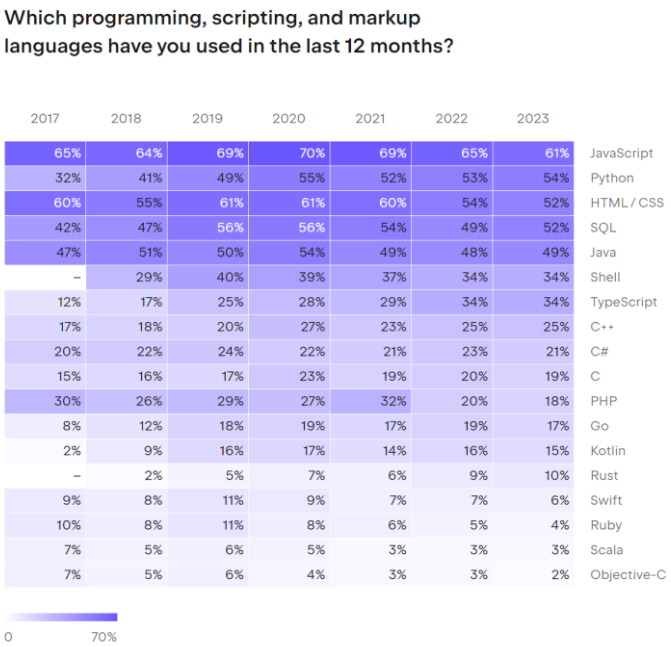


Abbildung: Umfrage "The State of Developer Ecosystem" 2023
[https://www.jetbrains.com/lp/devecosystem-2023]

Eine weitere viel beachtete Statistik ist die jährliche Umfrage von Stackoverflow.com. In der Jahresumfrage 2023 (2023, 2022, 2021, 2020, 2019, 2018) mit rund 65.000 Teilnehmern (weltweit) war C# auf Platz 8 (8, 8, 8, 7, 7, 8) der Liste der am meisten eingesetzten Programmier- und Markupsprachen mit 27,1%, (27,62%, 27,98%, 27,86%, 31,4 %, 31,9%, 35,35%).

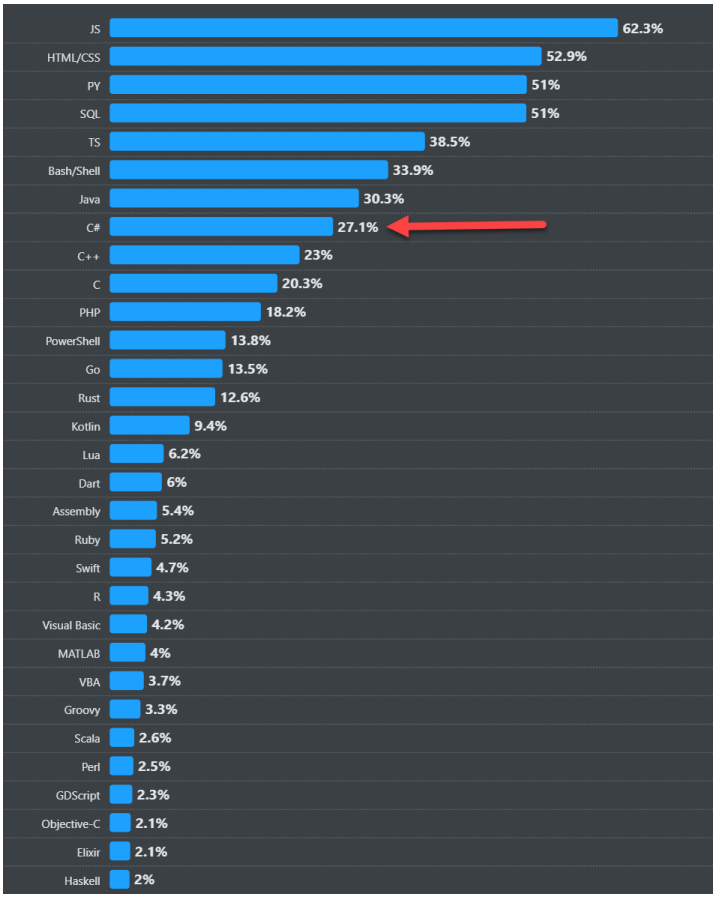


Abbildung: Einsatzhäufigkeit von C# in der Jahresumfrage 2024 von stackoverflow.com [https://survey.stackoverflow.co/2024]

In der Stackoverflow-Umfrage wird auch nach "Desired" (blauer Kreis, bei C# 21,6%) und "Admired" (roter Kreis, bei C# 64,1%) gefragt.

- "Admired": Ist im Einsatz und Entwickler/Entwicklerin möchte es weiterhin nutzen
- "Desired": Bisher nicht im Einsatz, aber Entwickler/Entwicklerin möchte es gerne nutzen

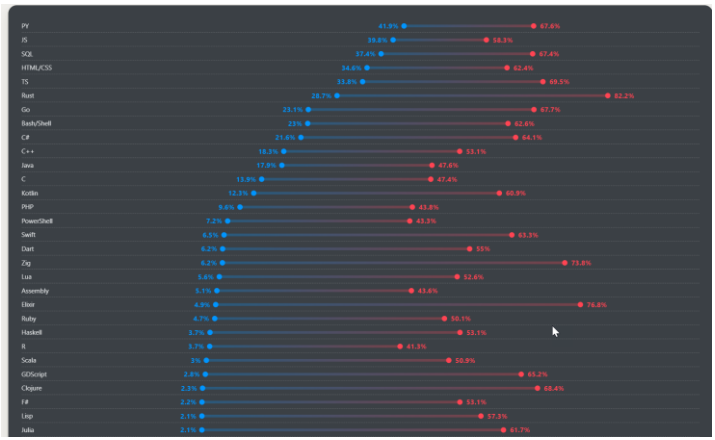


Abbildung: Liebe und Abneigung zu C# in der Jahresumfrage 2024 von stackoverflow.com [https://survey.stackoverflow.co/2024]

Eine weitere Umfrage unter Entwicklern liefert SlashData. Das Analystenteam von SlashData berichtet mit seinem Report "State Of The Developer Nation" vierteljährlich darüber, mit welchen Programmiersprachen die weltweite Gemeinschaft der Softwareentwickler arbeitet. Hier liegt C# auf Platz 4. JavaScript, TypeScript und CoffeeScript sind zusammengefasst auf Platz 1.

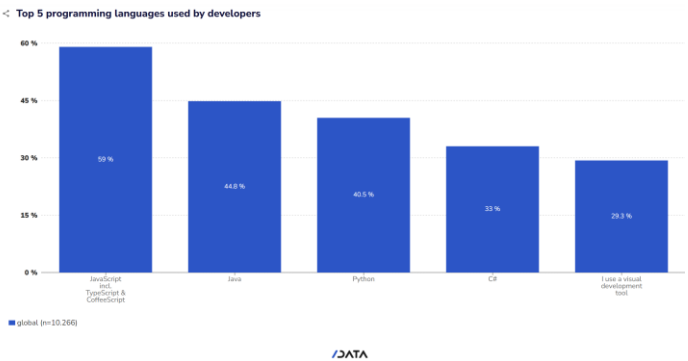


Abbildung: SlashData-Umfrage, Stand 1. Quartal 2022
[Quelle: https://www.developernation.net/developer-reports/dn26]

Falls es Ihnen bei der Programmierung auf Energie-Effizienz ankommt, sollten Sie sich diese Studie durchlesen: "Energy Efficiency across Programming Languages - How Does Energy, Time, and Memory Relate?" [*https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf*]

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Abbildung: C# liegt bei der Studie zum Vergleich der Programmiersprachen im Mittelfeld

6.12 Editoren für C#

Microsoft liefert für C# selbst drei Editoren:

- **Visual Studio**: nur für Windows. Kostenfreie Community-Version nur für Open Source-Projekte, Studierende und kleine Unternehmen.
  visualstudio.microsoft.com/de/downloads
- **Visual Studio for Mac**: kostenfrei (Nachfolger des früheren Xamarin Studio, wird aber am 31.8.2024 eingestellt)
  visualstudio.microsoft.com/de/vs/mac

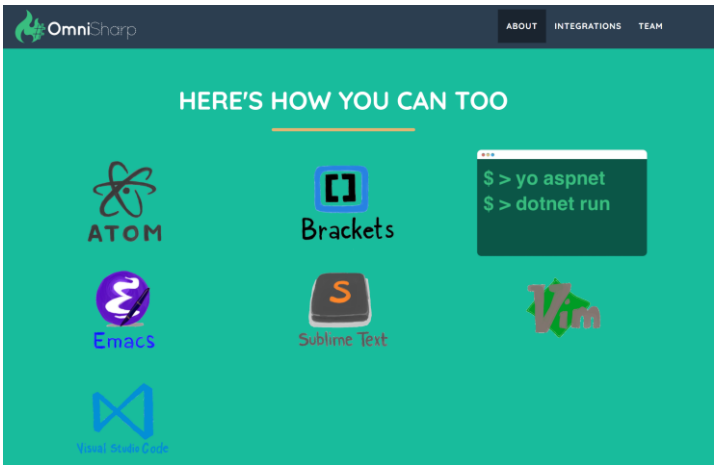
- **Visual Studio Code**: kostenfrei für Windows, macOS und Linux.
  code.visualstudio.com
  Die C#-Erweiterung "C# for Visual Studio Code" muss installiert sein!
  marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp

  Projektmappen-Explorer und Test-Explorer bekommt man über eine weitere Erweiterung, das **C# Dev Kit**:

  https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit

  Beide Erweiterungen beinhalten aber nicht alle Werkzeuge aus dem großen Visual Studio, z.B. keine grafischen UI-Designer

Zudem liefert Microsoft mit OmniSharp [*www.omnisharp.net*] eine Basis für die Integration in anderen (plattformneutralen) Editoren wie ATOM, Brackets, Emacs, Sublime und Vim (siehe Abbildung). Hier wird nicht nur Syntax-Farbeinfärbung, sondern auch Eingabeunterstützung (IntelliSense) angeboten. Auch die Visual Studio Code-Erweiterung für C# basiert auf OmniSharp.



Es gibt weitere einfache Editoren, die für die C#-Syntax nur Einfärbung, aber keine Eingabeunterstützung bieten.

Einen weiteren professionellen C#-Editor mit vielen Eingabeunterstützung- und Refactoring-Funktionen liefert die Firma JetBrains mit ihrem Produkt "Rider" (kostenpflichtig, *www.jetbrains.com/rider*).

6.13 C# 13.0

C# 13.0 ist zusammen mit Visual Studio 2022 Version 17.12 und .NET 9.0 am 12. November 2024 erschienen.

Wie schon bei .NET 6.0/C# 10.0 und .NET 7.0/C# 11.0 sowie .NET 8.0/C# 12.0 verwendet Microsoft bei .NET 9.0/C# 13.0 an vielen, aber nicht allen Stellen die Versionsnummer ohne ".0". Hier wird einheitlich die Schreibweise mit ".0" verwendet.

Anders als .NET 8.0 besitzt die 9.0-Version nur einen "Standard-Term-Support" (STS) für 18 Monate statt 36 Monaten. Nach aktuellem Stand gibt es dafür dann also Unterstützung und Updates von November 2024 bis Mai 2026.

6.14 Support für C# 13.0

C# 13.0 wird offiziell von Microsoft erst ab .NET 9.0 unterstützt ("C# 13.0 is supported only on .NET 9 and newer versions." [*learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version*].

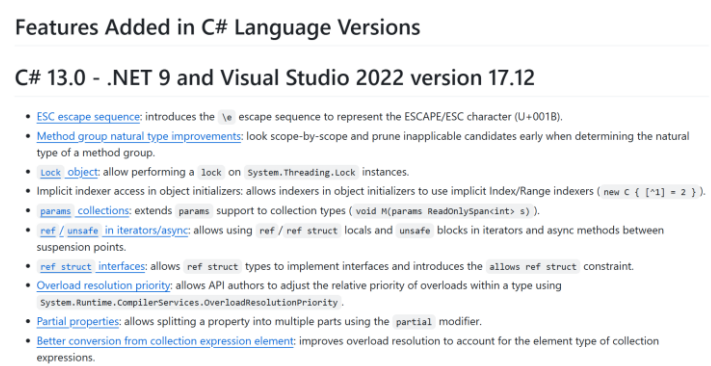
Tipp: Man kann allerdings auf eigene Verantwortung dennoch die einige (aber nicht alle!) C# 13.0-Sprachfeatures auch in älteren .NET-Versionen einschließlich .NET Framework, .NET Core und Xamarin nutzen. Dazu muss man die <LangVersion> in der Projektdatei (.csproj) auf "13.0" erhöhen. Dies wird im Kapitel "Erste C#-Schritte/Festlegen der Compilerversion" beschrieben.

Bitte beachten Sie aber, dass es für den Einsatz von C# 13.0-Sprachfeatures in .NET-Versionen vor 9.0 keinen technischen Support von Microsoft gibt, d.h. falls Sie Probleme damit haben, können Sie nicht Ihren Support-Vertrag nutzen, um Microsoft um Hilfe zu ersuchen. Dennoch ist der Einsatz höherer C#-Versionen in älteren .NET-Projekten in einigen Unternehmen gängige und problemlose Praxis.

Notwendige Visual Studio-Version für C# 13.0 ist Visual Studio 2022 v17.12 oder höher. Eine Verwendung von C# 14.0 auch mit einer aktuellen Version von Visual Studio Code und anderen OmniSharp-kompatiblen Editoren [*www.omnisharp.net*] ist möglich.

6.15 Neuerungen in C# 13.0

In C# 13.0 sind gegenüber Version 12.0 zehn Neuerungen erschienen. In C# 12.0 gab es sieben Neuerungen gegenüber C# 11.0. In C# 11.0 gegenüber 10.0 sowie Version 10.0 gegenüber 9.0 gab es jeweils 16 Neuerungen.



Features Added in C# Language Versions

C# 13.0 - .NET 9 and Visual Studio 2022 version 17.12

- ESC escape sequence: introduces the `\e` escape sequence to represent the ESCAPE/ESC character (U+001B).
- Method group natural type improvements: look scope-by-scope and prune inapplicable candidates early when determining the natural type of a method group.
- `Lock` object: allow performing a `lock` on `System.Threading.Lock` instances.
- Implicit indexer access in object initializers: allows indexers in object initializers to use implicit Index/Range indexers (`new C { [^1] = 2 }`).
- `params` collections: extends `params` support to collection types (`void M(params ReadOnlySpan<int> s)`).
- `ref` / `unsafe` in iterators/async: allows using `ref` / `ref struct` locals and `unsafe` blocks in iterators and async methods between suspension points.
- `ref struct` interfaces: allows `ref struct` types to implement interfaces and introduces the `allows ref struct` constraint.
- Overload resolution priority: allows API authors to adjust the relative priority of overloads within a type using `System.Runtime.CompilerServices.OverloadResolutionPriority`.
- Partial properties: allows splitting a property into multiple parts using the `partial` modifier.
- Better conversion from collection expression element: improves overload resolution to account for the element type of collection expressions.

Abbildung: Übersicht über die Neuerungen in C# 13.0 | Quelle: Microsoft [github.com/dotnet/csharplang/blob/main/Language-Version-History.md]

Sie finden in diesem Buch:

- Partielle Properties und partielle Indexer im Kapitel "Partielle Klassen, Methoden, Properties und partielle Indexer"
- Prioritäten für Methodenüberladungen im Kapitel "Methoden"
- Generische Mengen in Verbindung mit dem Schlüsselwort params im Kapitel "Methoden/Parameterlisten"
- Konsolenausgabenformatierung mit ANSI-Codes mit neuem Escape-Zeichen \e im Kapitel "Datentypen"
- System.Threading.Lock im Kapitel "Exklusive Zugriffe auf Ressourcen mit lock()"
- Einsatz von Range-Indexern bei der Mengeninitialisierung im Kapitel "Objektmengen-Initialisierung mit Index"
- Neuerungen für ref struct im Kapitel "Strukturen/Strukturen ausschließlich auf dem Stack (ref struct)"

6.16 C# 13.0 in älteren .NET-Versionen

Nur diejenigen neuen Sprachfeatures funktionieren auch in .NET-Versionen vor .NET 9.0, die keine Abhängigkeit von erst in .NET 9.0 eingeführten Basisbibliotheksklassen haben. Sofern man <LangVersion>latest</LangVersion> setzt in der Projektdatei, sind in älteren Versionen folgende neuen Sprachfeatures von C# 13.0 möglich:

- Partielle Properties und partielle Indexer
- Generische Mengen in Verbindung mit dem Schlüsselwort params
- Neuerungen für ref struct, außer der Verwendung als Typargument
- Escape-Zeichen \e

6.17 Breaking Changes in C# 13.0

Es gibt einige wenige Breaking Changes im Verhalten des Compilers in C# 13.0 gegenüber C# 12.0. Dies sind jedoch Sonderfälle von geringer Bedeutung (z.B. Verbot der Annotation [InlineArray] auf record struct) und werden hier daher nicht näher besprochen. Sie finden die Informationen unter

https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/breaking-changes/compiler%20breaking%20changes%20-%20dotnet%209

6.18 Vertagte neue Sprachfeatures

Folgende Sprachfeatures waren für C# 13.0 geplant, wurden aber dann auf C# 14.0 (November 2025) vertagt:

- Tupel-Dekonstruktion (int x, string y) = default statt (default, default)
- Automatische Konvertierung zwischen Array, Span<T>, ReadOnlySpan<T>
- Semi-Auto-Properties mit neuem Schlüsselwort field
- Extension Types: Eine weiterentwickelte Form der Extension Methods, bei der man nicht nur Instanzmethoden, sondern Methoden und Properties sowohl auf Instanz- als auch Klassenebene ("static") ergänzen kann. Dazu will Microsoft das neue Schlüsselwort extension einführen, siehe nächstes Bild.

```
1   public implicit extension StringExtensions for string
2   {
        0 references
3     public int Dots { get; set; }
4     public string Truncate(int count)
5     {
6      if (this == null) return "";
7      if (this.Length <= count) return this;
8      return this.Substring(0, count) + Dots;
9     }
10    public static string Create(int count)
11    {
12     return new string('.', count);
13    }
14  }
15
16  // Verwendung des Extension Types
17  string str = "Hello World";
18  str.Dots = "...";
19  string truncated = str.Truncate(5); // Hello...
20
21  string str2 = String.Create(5); // "....."
```

Abbildung: So sollten Extension Types in C# 13.0 aussehen

Die kommende Version C# 14.0 soll im November 2025 zusammen mit .NET 10.0 erscheinen.

Die Liste der Sprachfeatures, an denen Microsoft aktiv arbeitet, findet man unter

https://github.com/dotnet/roslyn/blob/main/docs/Language%20Feature%20Status.md

Working Set

Feature	Branch	State	Developer	Reviewer	IDE Buddy	LDM Champ
Default in deconstruction	decon-default	In Progress	jcouv	gafter		jcouv
Roles/Extensions	roles	In Progress	jcouv	AlekseyTs, jjonescz	CyrusNajmabadi	MadsTorgersen
Null-conditional assignment	null-conditional-assignment	In Progress	RikkiGibson	cston, jjonescz	TBD	RikkiGibson
`field` keyword in properties	field-keyword	Merged into 17.12p3	Youssef1313, cston	333fred, RikkiGibson	CyrusNajmabadi	CyrusNajmabadi
First-class Span Types	FirstClassSpan	Merged into 17.13p1	jjonescz	cston, 333fred		333fred, stephentoub

Abbildung: Sprachfeatures in Arbeit für C# 14.0

Sprachfeatures, die sich bereits in der Entwicklung befinden aber noch nicht Teil des Sprachcompilers sind, können Sie ausprobieren auf dieser Website:

sharplab.io

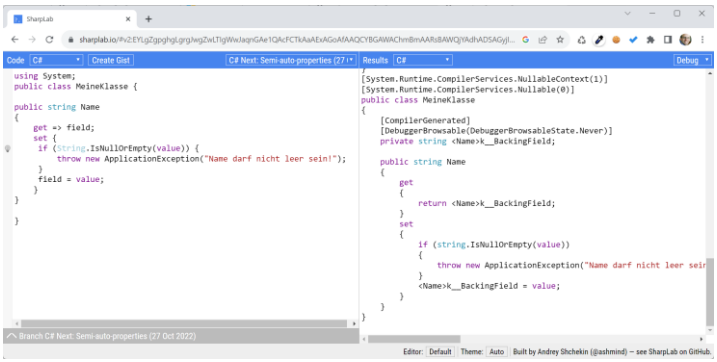


Abbildung: sharplab.io mit dem nun für C# 13.0 geplanten Sprachfeatures "Semi-Auto-Properties" mit dem neuen Schlüsselwort field für den Zugriff auf das automatisch generierte Backing-Field eines Properties

6.19 Vorschläge für kommende Sprachfeatures

Weitere Vorschläge für kommende Sprachfeatures findet man unter

github.com/dotnet/csharplang/tree/main/proposals

Jedermann kann Vorschläge für neue Sprachfeatures einreichen; die Hürden zur Annahme sind aber recht hoch.

7 Grundkonzepte von C#

Konzeptionell wurde C# vor allem von C++ und Java beeinflusst; man kann aber auch Parallelen zu Visual Basic und Delphi finden.

7.1 Sprachtypus

Im Gegensatz zur Programmiersprache C++, die eine hybride Sprache aus objektorientierten und nicht-objektorientierten Konzepten darstellt, ist C# ebenso wie Java eine rein objektorientierte Sprache, d.h. alle Datentypen basieren auf Klassen und alle Anweisungen erfolgen in Klassen.

C# unterstützt alle zentralen Konzepte der Objektorientierung einschließlich Schnittstellen, Vererbung und Polymorphismus. Schon in C# 2.0 wurde auch die Unterstützung für generische Klassen und partielle Klassen hinzugefügt. Außerdem besitzt C# Konzepte der funktionalen Programmierung (Delegates und Lambda-Ausdrücke). Man nennt C# daher auch "Multi-Paradigmen-Sprache".

7.2 Groß- und Kleinschreibung

Ein wesentlicher Unterschied zwischen C# und Visual Basic .NET ist die Tatsache, dass C# im Gegensatz zu Visual Basic .NET zwischen Groß- und Kleinschreibung unterscheidet. Dies gilt sowohl für die Schlüsselwörter der Sprache als auch für alle Bezeichner (a und A sind verschiedene Variablen!). Die Schlüsselwörter der Sprache C# werden komplett in Kleinbuchstaben geschrieben.

7.3 Schlüsselwörter der Sprache

Die folgende Liste zeigt die vordefinierten Schlüsselwörter der Programmiersprache C#. Diese Namen dürfen in der gleichen Groß-/Kleinschreibung nicht als Bezeichner verwendet werden (Quelle: *learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index*).

abstract	event	namespace	static
as	explicit	new	string
base	extern	null	struct
bool	false	object	switch
break	finally	operator	this
byte	fixed	out	throw
case	float	override	true
catch	for	params	try
char	foreach	private	typeof
checked	goto	protected	uint
class	if	public	ulong
const	implicit	readonly	unchecked
continue	in	ref	unsafe
decimal	int	return	ushort
default	interface	sbyte	using
delegate	internal	sealed	virtual
do	is	short	void
double	lock	sizeof	volatile
else	long	stackalloc	while
enum			

Darüberhinaus gibt es weitere sogenannte Kontext-Schlüsselworte, die eine besondere Bedeutung in bestimmten Zusammenhängen haben, die aber dennoch auch als Bezeichner verwendet werden dürfen.

add	get	notnull	select
and	global	nuint	set
alias	group	on	unmanaged (function
ascending	init	or	pointer calling convention)
args	into	orderby	unmanaged (generic type
async	join	partial (type)	constraint)
await	let	partial (method)	value
by	managed (function pointer	record	var
descending	calling convention)	remove	when (filter condition)
dynamic	nameof	required	where (generic type
equals	nint	scoped	constraint)
file	not		where (query clause)
from			with
			yield

7.4 Namensregeln und Namenskonventionen

Bei der Vergabe von eigenen Bezeichnern (z.B. Variablennamen, Parameternamen, Attributnamen und Methodennamen) gibt es verpflichtende Regeln und optionale Namenskonventionen.

Verpflichtende Regeln sind:

- Der Name darf nur Buchstaben (*), Zahlen und den Unterstrich enthalten.
- Der Name muss mit einem Buchstaben beginnen
- Die Groß- und Kleinschreibung ist relevant
- Es dürfen keine Namen von C#-Schlüsselwörtern verwendet werden (Theoretisch kann man C#-Schlüsselwörternamen mit vorangestelltem @ verwenden, also z.B. @class oder @if oder @for usw. Aber dies zu tun, ist nicht üblich und erschwert den Lesefluss von Programmcode!)

Hinweis: (*) Umlaute sind erlaubt, aber sollten dennoch besser vermieden werden: Nicht alle Werkzeuge und alle Menschen kommen damit gut klar!

Seit C# 11.0 gibt es zudem eine Compiler-Warnung (CS8981), wenn man Typnamen (für Klassen, Strukturen, Enumerationen, Record-Typen, Delegaten) verwendet, die nur aus Kleinbuchstaben bestehen. Dies geschieht vor dem Hintergrund, dass Microsoft zukünftig weitere neue Schlüsselwörter in die Programmiersprache C# einführen möchte (vgl. das in C# 11.0 neu eingeführte `required`), ohne dass es Konflikte mit bestehenden Typnamen der C#-Nutzer gibt.

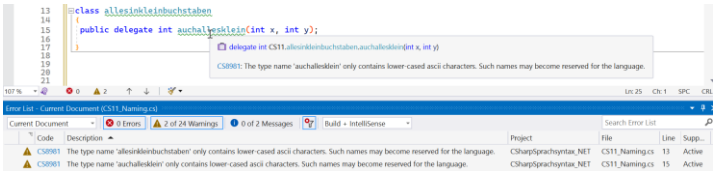


Abbildung: Warnung bei einem Klassennamen und einem Delegaten, die nur aus Kleinbuchstaben besteht.

Optionale Regeln hat Microsoft in den ".NET Framework Design Guidelines" [*learn.microsoft.com/en-us/dotnet/standard/design-guidelines*] definiert. Die wichtigsten Regeln dort sind:

- Für die Groß-/Kleinschreibung gilt grundsätzlich **PascalCasing**, d.h. ein Bezeichner beginnt grundsätzlich mit einem Großbuchstaben und jedes weitere Wort innerhalb des Bezeichners beginnt ebenfalls wieder mit einem Großbuchstaben.

  Beispiel: KundenPortalBenutzer

- Ausnahmen gibt es für Abkürzungen, die nur aus zwei Buchstaben bestehen. Diese dürfen komplett in Großbuchstaben geschrieben sein (z.B. UI und IO). Alle anderen Abkürzungen werden entgegen ihrer normalen Schreibweise in Groß-/Kleinschreibung geschrieben (z.B. Xml, Xsd und W3c).

  Beispiele: System.IO.File, System.Xml.XmlDocument

- Lokale Variablen, versteckte Attribute (private/protected) und Parameternamen sollen in **camelCasing** (Bezeichner beginnt mit einem Kleinbuchstaben, aber jedes weitere Wort innerhalb des Bezeichners beginnt mit einem Großbuchstaben) geschrieben werden.

  Beispiel: Login(KundenPortalBenutzer kundenPortalBenutzer)

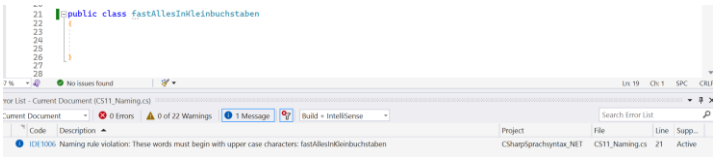


Abbildung: Hier gibt es keine Warnung, sondern nur eine Nachricht von der eingebauten Style-Polizei, weil der Klassenname zwar Großbuchstaben enthält, aber nicht mit einem solchen beginnt.

7.5 Blockbildung und Umbrüche

Blockbildung findet in C# im Stil der Programmiersprachen C und C++ statt, also mit geschweiften Klammern { }. Befehlstrenner ist das Semikolon (;).

Ein Zeilenumbruch kann zwischen den Elementen des Ausdrucks auftreten, ohne das besondere Vorkehrungen getroffen werden müssen. Zahlen können seit C# 7.0 mit einem Unterstrich gegliedert werden; aber man darf innerhalb von Zahlen keinen Zeilenumbruch haben.

```
// Formel ohne Umbrüche
double Ergebnis1 = (2 + 3) * ( 5 + 6) * (7 * 8) + 3.141_592_653_59;
```

```
// Formel mit Umbrüchen
double Ergebnis2 = (2 + 3) *
                   (5 + 6) *
                   (7 * 8)
                   + 3.141_592_653_59;
```

7.6 Hello World

Das folgende Listing zeigt das Hello World-Beispiel in C#, das man in jeder Programmiersprache zuerst schreibt.

```
using System;

namespace HalloWelt
{
 class Program
 {
  static void Main(string[] args)
  {
   Console.WriteLine("Hallo Welt!");
  }
 }
}
```

Marginal komplexer ist diese Variante, die – sofern vorhanden – den ersten übergebenen Kommandozeilenparameter als Name auffasst und die Person mit Namen grüßt.

```
namespace HalloWelt
{
 class Program
 {
  static void Main(string[] args)
  {
   if (args.Length > 0)
   {
    var name = args[0];
    // Ausgabe mit String Interpolation
    Console.WriteLine($"Hallo {name}!");
    Console.ReadLine();
   }
   else
   {
    Console.WriteLine("Hallo Welt!");
   }
  }
 }
}
```

7.7 Eingebaute Funktionen

Anders als in Visual Basic existieren in C# keine eingebauten Funktionen zur Typumwandlung (z.B. CBool(), CInt(), CLng(), CType()), Zeichenkettenverarbeitung (z.B. InStr(), Trim(), LCase()) und Ausgabe (z.B. MsgBox()). Auch die My-Klassenbibliothek ist nicht vorhanden.

Grundsätzlich ist es möglich, die in Visual Basic eingebauten Funktionen und die My-Bibliothek durch Referenzierung der Microsoft.VisualBasic.dll auch in C# zu nutzen. Dies sollte jedoch vermieden werden, um sprachunabhängig zu bleiben. Alle Visual Basic-Funktionen und -Objekte sind auch in der .NET-Klassenbibliothek enthalten, z.B. String.IndexOf() statt InStr() und Convert.ToInt32() statt CInt().

8 Der C#-Compiler

Es gibt zwei Varianten des C#-Compilers: eine alte, in C++ geschriebene, und neue, in C# geschriebene Implementierung.

8.1 Der ursprüngliche (alte) C#-Compiler

Der Kommandozeilencompiler für C# im .NET Framework Redistributable ist csc.exe. Er wird installiert im Verzeichnis C:\Windows\Microsoft.NET\Framework64\v4.0.30319. Alternativ kann er in der .NET Framework-Klassenbibliothek im sogenannten "CodeDOM" durch die Klasse Microsoft.CSharp.CSCodeProvider angesprochen werden.

Wenn Sie heute ein aktuelles Microsoft .NET Framework (z.B. 4.8.1) verwenden, so ist dort der ursprüngliche C#-Compiler immer noch in der Version 5.0 enthalten.

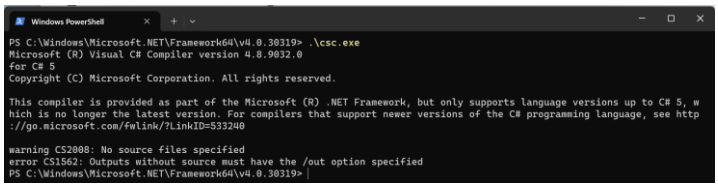


Abbildung: In .NET Framework 4.8.1 ist der C#-Compiler für C# 5.0 enthalten.

8.1.1 Kompilierung mit csc.exe

Der Befehl

```
csc.exe Dateiname1.cs Dateiname2.cs DateinameX.cs
```

oder

```
csc Dateiname1.cs Dateiname2.cs DateinameX.cs
```

übersetzt die angegebenen Dateien in eine Konsolenanwendung. Eine Datei, die als Konsolenanwendung oder Windows-Anwendung kompiliert wird, muss genau eine Klasse mit folgendem Einstiegspunkt besitzen: public static void Main().

Listing: »Hello World« in C#

```
class Hauptprogramm
{
        public static void Main()
        {
                System.Console.WriteLine("Hello World!");
        }
}
```

8.1.2 Kommandozeilenparameter

Der Kommandozeilencompiler bietet zahlreiche Optionen. Die wichtigsten davon sind:

- /target:winexe Der Compiler erzeugt eine Windows-Anwendung
- /target:library Der Compiler erzeugt eine DLL (kein Main() notwendig)
- /r:Dateiliste Die angegebenen Assemblys werden referenziert

- /out:Dateiname Name der Ausgabedatei
- /doc:Dateiname Der Compiler erzeugt zusätzlich eine XML-Dokumentationsdatei
- /help Anzeige der Hilfe zu den Compiler-Optionen
- Anders als beim Visual Basic .NET-Compiler vbc.exe müssen die Optionen /target und /out bei csc.exe vor den Namen der Quelldateien in der Parameterliste erscheinen.

Es folgt die komplette Liste der Kommandozeilenparameter des alten C#-Compilers

```
                      Visual C# Compiler Options

                      - OUTPUT FILES -
/out:<file>                   Specify output file name (default: base name of
file with main class or first file)
/target:exe                   Build a console executable (default) (Short form:
/t:exe)
/target:winexe                Build a Windows executable (Short form: /t:winexe)
/target:library               Build a library (Short form: /t:library)
/target:module                Build a module that can be added to another
assembly (Short form: /t:module)
/target:appcontainerexe       Build an Appcontainer executable (Short form:
/t:appcontainerexe)
/target:winmdobj              Build a Windows Runtime intermediate file that is
consumed by WinMDExp (Short form: /t:winmdobj)
/doc:<file>                   XML Documentation file to generate
/platform:<string>            Limit which platforms this code can run on: x86,
Itanium, x64, arm, anycpu32bitpreferred, or anycpu. The default is anycpu.

                      - INPUT FILES -
/recurse:<wildcard>           Include all files in the current directory and
subdirectories according to the wildcard specifications
/reference:<alias>=<file>     Reference metadata from the specified assembly
file using the given alias (Short form: /r)
/reference:<file list>        Reference metadata from the specified assembly
files (Short form: /r)
/addmodule:<file list>        Link the specified modules into this assembly
/link:<file list>             Embed metadata from the specified interop assembly
files (Short form: /l)

                      - RESOURCES -
/win32res:<file>              Specify a Win32 resource file (.res)
/win32icon:<file>             Use this icon for the output
/win32manifest:<file>         Specify a Win32 manifest file (.xml)
/nowin32manifest              Do not include the default Win32 manifest
/resource:<resinfo>           Embed the specified resource (Short form: /res)
/linkresource:<resinfo>       Link the specified resource to this assembly
(Short form: /linkres)
                              Where the resinfo format is <file>[,<string
name>[,public|private]]

                      - CODE GENERATION -
/debug[+|-]                   Emit debugging information
/debug:{full|pdbonly}         Specify debugging type ('full' is default, and
enables attaching a debugger to a running program)
/optimize[+|-]                Enable optimizations (Short form: /o)
```

```
                        - ERRORS AND WARNINGS -
/warnaserror[+|-]               Report all warnings as errors
/warnaserror[+|-]:<warn list>   Report specific warnings as errors
/warn:<n>                       Set warning level (0-4) (Short form: /w)
/nowarn:<warn list>             Disable specific warning messages

                        - LANGUAGE -
/checked[+|-]                   Generate overflow checks
/unsafe[+|-]                    Allow 'unsafe' code
/define:<symbol list>           Define conditional compilation symbol(s) (Short
form: /d)
/langversion:<string>           Specify language version mode: ISO-1, ISO-2, 3, 4,
5, or Default

                        - SECURITY -
/delaysign[+|-]                 Delay-sign the assembly using only the public
portion of the strong name key
/keyfile:<file>                 Specify a strong name key file
/keycontainer:<string>          Specify a strong name key container
/highentropyva[+|-]             Enable high-entropy ASLR

                        - MISCELLANEOUS -
@<file>                         Read response file for more options
/help                           Display this usage message (Short form: /?)
/nologo                         Suppress compiler copyright message
/noconfig                       Do not auto include CSC.RSP file

                        - ADVANCED -
/baseaddress:<address>          Base address for the library to be built
/bugreport:<file>               Create a 'Bug Report' file
/codepage:<n>                   Specify the codepage to use when opening source
files
/utf8output                     Output compiler messages in UTF-8 encoding
/main:<type>                    Specify the type that contains the entry point
(ignore all other possible entry points) (Short form: /m)
/fullpaths                      Compiler generates fully qualified paths
/filealign:<n>                  Specify the alignment used for output file
sections
/pdb:<file>                     Specify debug information file name (default:
output file name with .pdb extension)
/errorendlocation               Output line and column of the end location of each
error
/preferreduilang                Specify the preferred output language name.
/nostdlib[+|-]                  Do not reference standard library (mscorlib.dll)
/subsystemversion:<string>      Specify subsystem version of this assembly
/lib:<file list>                Specify additional directories to search in for
references
/errorreport:<string>           Specify how to handle internal compiler errors:
prompt, send, queue, or none. The default is queue.
/appconfig:<file>               Specify an application configuration file
containing assembly binding settings
/moduleassemblyname:<string>    Name of the assembly which this module will be a
part of
```

8.2 Der aktuelle (neue) C#-Compiler

Der im Projekt "Roslyn" neu implementierte C#-Compiler heißt auch csc.exe; er ist aber nicht mehr Teil des .NET Framework Redistributable. Er wird auf diesen Wegen verbreitet:

- Visual Studio bzw. Visual Studio Build Tools
- .NET SDK
  dotnet.microsoft.com/download/dotnet/6.0
- NuGet-Paket "Microsoft.Net.Compilers"
  www.nuget.org/packages/Microsoft.Net.Compilers

Visual Studio installiert den Compiler im Dateisystemverzeichnis C:\Program Files\Microsoft Visual Studio\2022\<Visual Studio-Edition>\MSBuild\Current\Bin\Roslyn z.B. *C:\Program Files\Microsoft Visual Studio\2022\Enterprise\MSBuild\Current\Bin\Roslyn.*

Das NuGet-Paket *www.nuget.org/packages/Microsoft.Net.Compilers* enthält den csc.exe im Ordner */Tools*. Im .NET Core SDK wird der C#-Compiler nicht als csc.exe mitgeliefert, sondern über die .NET CLI-Werkzeuge angesprochen (z.B. dotnet build).

Die folgende Abbildung zeigt die Installation des C#-Compilers per NuGet.exe mit dem Befehl:

```
nuget install Microsoft.Net.Compilers
```

Das Programm NuGet.exe bekommt man *www.nuget.org/downloads*

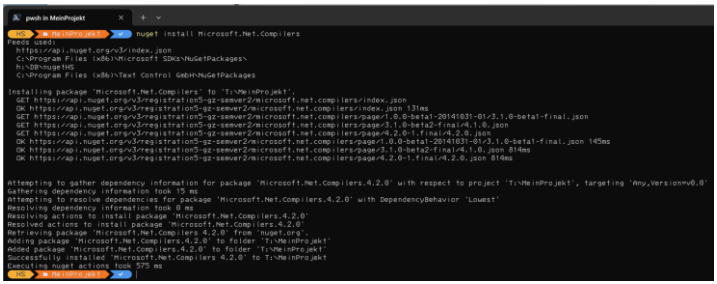


Abbildung: Installation des neuen C#-Compilers via NuGet

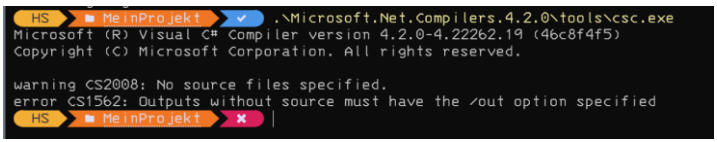


Abbildung: Start des neuen C#-Compiler aus der NuGet-Installation

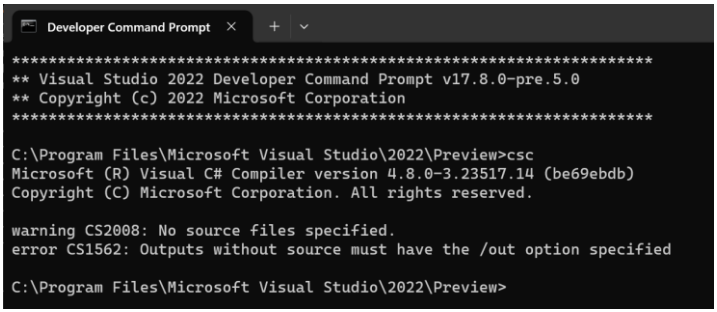


Abbildung: Start des neuen C#-Compilers aus der Visual Studio-Installation

Die Neufassung des CodeDOM-APIs mit dem neuen Compiler erhält man über das NuGet-Paket *www.nuget.org/packages/Microsoft.CodeDom.Providers.DotNetCompilerPlatform*.

8.2.1 Versionsnummern des Compilers

Die Versionsnummer des neuen C#-Compilers richtet sich nach dem Funktionsumfang des Compilers, nicht nach den Sprachfeatures (siehe folgende Abbildung).

- Versions `1.x` mean C# 6.0 and VB 14 (Visual Studio 2015 and updates). For instance, `1.3.2` corresponds to the most recent update (update 3) of Visual Studio 2015.
- Version `2.0` means C# 7.0 and VB 15 (Visual Studio 2017 version 15.0).
- Version `2.1` is still C# 7.0, but with a couple fixes (Visual Studio 2017 version 15.1).
- Version `2.2` is still C# 7.0, but with a couple more fixes (Visual Studio 2017 version 15.2). Language version "default" was updated to mean "7.0".
- Version `2.3` means C# 7.1 and VB 15.3 (Visual Studio 2017 version 15.3). For instance, `2.3.0-beta1` corresponds to Visual Studio 2017 version 15.3 (Preview 1).
- Version `2.4` is still C# 7.1 and VB 15.3, but with a couple fixes (Visual Studio 2017 version 15.4).
- Version `2.6` means C# 7.2 and VB 15.5 (Visual Studio 2017 version 15.5).
- Version `2.7` means C# 7.2 and VB 15.5, but with a number of fixes (Visual Studio 2017 version 15.6).
- Version `2.8` means C# 7.3 (Visual Studio 2017 version 15.7)
- Version `2.9` is still C# 7.3 and VB 15.5, but with more fixes (Visual Studio 2017 version 15.8)
- Version `2.10` is still C# 7.3 and VB 15.5, but a couple more fixes (Visual Studio 2017 version 15.9)
- Version `3.0` includes a preview of C# 8.0 (Visual Studio 2019 version 16.0), but `2.11` was used for preview1.
- Version `3.1` includes a preview of C# 8.0 (Visual Studio 2019 version 16.1)
- Version `3.2` includes a preview of C# 8.0 (Visual Studio 2019 version 16.2)
- Version `3.3` includes C# 8.0 (Visual Studio 2019 version 16.3, .NET Core 3.0)
- Version `3.4` includes C# 8.0 (Visual Studio 2019 version 16.4, .NET Core 3.1)
- Version `3.5` includes C# 8.0 (Visual Studio 2019 version 16.5, .NET Core 3.1)
- Version `3.6` includes C# 8.0 (Visual Studio 2019 version 16.6, .NET Core 3.1)
- Version `3.7` includes C# 8.0 (Visual Studio 2019 version 16.7, .NET Core 3.1)
- Version `3.8` includes C# 9.0 (Visual Studio 2019 version 16.8, .NET 5)
- Version `3.9` includes C# 9.0 (Visual Studio 2019 version 16.9, .NET 5)
- Version `3.10` includes C# 9.0 (Visual Studio 2019 version 16.10, .NET 5)
- Version `3.11` includes C# 9.0 (Visual Studio 2019 version 16.11, .NET 5)
- Version `4.0` includes C# 10.0 (Visual Studio 2022 version 17.0, .NET 6)
- Version `4.1` includes C# 10.0 (Visual Studio 2022 version 17.1, .NET 6)
- Version `4.2` includes C# 10.0 (Visual Studio 2022 version 17.2, .NET 6)
- Version `4.3.1` includes C# 10.0 (Visual Studio 2022 version 17.3, .NET 6)
- Version `4.4` includes C# 11.0 (Visual Studio 2022 version 17.4, .NET 7)
- Version `4.5` includes C# 11.0 (Visual Studio 2022 version 17.5, .NET 7)
- Version `4.6` includes C# 11.0 (Visual Studio 2022 version 17.6, .NET 7)
- Version '4.7' includes C# 11.0 (Visual Studio 2022 version 17.7, .NET 7)

Abbildung: Versionierung des neuen C#-Compilers
[github.com/dotnet/roslyn/blob/master/docs/wiki/NuGet-packages.md]

8.2.2 Kommandozeilenparameter

Es folgen die Kommandozeilenparameter des neuen C#-Compilers

```
                        Visual C# Compiler Options

                     - OUTPUT FILES -
/out:<file>                 Specify output file name (default: base name of
                            file with main class or first file)
/target:exe                 Build a console executable (default) (Short
                            form: /t:exe)
/target:winexe              Build a Windows executable (Short form:
                            /t:winexe)
/target:library             Build a library (Short form: /t:library)
/target:module              Build a module that can be added to another
                            assembly (Short form: /t:module)
/target:appcontainerexe     Build an Appcontainer executable (Short form:
```

```
                              /t:appcontainerexe)
 /target:winmdobj             Build a Windows Runtime intermediate file that
                              is consumed by WinMDExp (Short form: /t:winmdobj)
 /doc:<file>                  XML Documentation file to generate
 /refout:<file>               Reference assembly output to generate
 /platform:<string>           Limit which platforms this code can run on: x86,
                              Itanium, x64, arm, anycpu32bitpreferred, or
                              anycpu. The default is anycpu.

                        - INPUT FILES -
 /recurse:<wildcard>          Include all files in the current directory and
                              subdirectories according to the wildcard
                              specifications
 /reference:<alias>=<file>    Reference metadata from the specified assembly
                              file using the given alias (Short form: /r)
 /reference:<file list>       Reference metadata from the specified assembly
                              files (Short form: /r)
 /addmodule:<file list>       Link the specified modules into this assembly
 /link:<file list>            Embed metadata from the specified interop
                              assembly files (Short form: /l)
 /analyzer:<file list>        Run the analyzers from this assembly
                              (Short form: /a)
 /additionalfile:<file list>  Additional files that don't directly affect code
                              generation but may be used by analyzers for
producing
                              errors or warnings.
 /embed                       Embed all source files in the PDB.
 /embed:<file list>           Embed specific files in the PDB

                        - RESOURCES -
 /win32res:<file>             Specify a Win32 resource file (.res)
 /win32icon:<file>            Use this icon for the output
 /win32manifest:<file>        Specify a Win32 manifest file (.xml)
 /nowin32manifest             Do not include the default Win32 manifest
 /resource:<resinfo>          Embed the specified resource (Short form: /res)
 /linkresource:<resinfo>      Link the specified resource to this assembly
                              (Short form: /linkres) Where the resinfo format
                              is <file>[,<string name>[,public|private]]

                        - CODE GENERATION -
 /debug[+|-]                  Emit debugging information
 /debug:{full|pdbonly|portable|embedded}
                              Specify debugging type ('full' is default,
                              'portable' is a cross-platform format,
                              'embedded' is a cross-platform format embedded
into
                              the target .dll or .exe)
 /optimize[+|-]               Enable optimizations (Short form: /o)
 /deterministic               Produce a deterministic assembly
                              (including module version GUID and timestamp)
 /refonly                     Produce a reference assembly in place of the main
output
 /instrument:TestCoverage     Produce an assembly instrumented to collect
```

```
                                coverage information
 /sourcelink:<file>             Source link info to embed into PDB.

                        - ERRORS AND WARNINGS -
 /warnaserror[+|-]              Report all warnings as errors
 /warnaserror[+|-]:<warn list> Report specific warnings as errors
 /warn:<n>                      Set warning level (0-4) (Short form: /w)
 /nowarn:<warn list>            Disable specific warning messages
 /ruleset:<file>                Specify a ruleset file that disables specific
                                diagnostics.
 /errorlog:<file>               Specify a file to log all compiler and analyzer
                                diagnostics.
 /reportanalyzer                Report additional analyzer information, such as
                                execution time.

                        - LANGUAGE -
 /checked[+|-]                  Generate overflow checks
 /unsafe[+|-]                   Allow 'unsafe' code
 /define:<symbol list>          Define conditional compilation symbol(s) (Short
                                form: /d)
 /langversion:?                 Display the allowed values for language version
 /langversion:<string>          Specify language version such as
                                `default` (latest major version), or
                                `latest` (latest version, including minor
versions),
                                or specific versions like `6` or `7.1`

                        - SECURITY -
 /delaysign[+|-]                Delay-sign the assembly using only the public
                                portion of the strong name key
 /publicsign[+|-]               Public-sign the assembly using only the public
                                portion of the strong name key
 /keyfile:<file>                Specify a strong name key file
 /keycontainer:<string>         Specify a strong name key container
 /highentropyva[+|-]            Enable high-entropy ASLR

                        - MISCELLANEOUS -
 @<file>                        Read response file for more options
 /help                          Display this usage message (Short form: /?)
 /nologo                        Suppress compiler copyright message
 /noconfig                      Do not auto include CSC.RSP file
 /parallel[+|-]                 Concurrent build.
 /version                       Display the compiler version number and exit.

                        - ADVANCED -
 /baseaddress:<address>         Base address for the library to be built
 /checksumalgorithm:<alg>       Specify algorithm for calculating source file
                                checksum stored in PDB. Supported values are:
                                SHA1 (default) or SHA256.
 /codepage:<n>                  Specify the codepage to use when opening source
                                files
 /utf8output                    Output compiler messages in UTF-8 encoding
 /main:<type>                   Specify the type that contains the entry point
```

```
                              (ignore all other possible entry points) (Short
                              form: /m)
/fullpaths                    Compiler generates fully qualified paths
/filealign:<n>                Specify the alignment used for output file
                              sections
/pathmap:<K1>=<V1>,<K2>=<V2>,...
                              Specify a mapping for source path names output by
                              the compiler.
/pdb:<file>                   Specify debug information file name (default:
                              output file name with .pdb extension)
/errorendlocation             Output line and column of the end location of
                              each error
/preferreduilang              Specify the preferred output language name.
/nostdlib[+|-]                Do not reference standard library (mscorlib.dll)
/subsystemversion:<string>    Specify subsystem version of this assembly
/lib:<file list>              Specify additional directories to search in for
                              references
/errorreport:<string>         Specify how to handle internal compiler errors:
                              prompt, send, queue, or none. The default is
                              queue.
/appconfig:<file>             Specify an application configuration file
                              containing assembly binding settings
/moduleassemblyname:<string>  Name of the assembly which this module will be
                              a part of
/modulename:<string>          Specify the name of the source module
```

9 Erste C#-Schritte mit Visual Studio

Dieses Buch ist kein Handbuch für Visual Studio. Für Leser, die neu in Visual Studio sind, folgt jedoch hier eine kurze Einführung in das Anlegen und Übersetzen eines Projekts am Beispiel von Konsolenanwendungsprojekten für .NET Framework und .NET Core.

Achtung: Für C# 13.0 benötigen Sie Visual Studio 2022 ab Version 17.12. Für C# 12.0 benötigen Sie Visual Studio 2022 ab Version 17.8. Vorherige Versionen von Visual Studio 2022 ab 17.4 können C# 11.0. Die Version 17.0 bis 17.3 können nur C# 10.0. In Visual Studio 2019 kompiliert C# bis Version 9.0. Visual Studio 2017 kann nur C# 7.x.

Achten Sie auch darauf, ob Sie ein Konsolenprojekt für das klassische .NET Framework (Vorlagename: Console Application (.NET Framework) oder für das moderne .NET (Vorlagename: Console Application, Zusatz früher ".NET Core", heute kein Zusatztext mehr!) erstellen. Das klassische .NET Framework kann nur maximal C# 7.0 und einige Teile der moderneren C#-Versionen. Nur die jeweils modernsten .NET-Versionen können alle Sprachversionen.

9.1 Visual Studio versus Visual Studio Code

Visual Studio ist die primäre Entwicklungsumgebung für C#. Sie läuft allerdings auf Windows. *https://visualstudio.microsoft.com/downloads*

Falls Sie auf Linux oder macOS entwickeln wollen, sollten Sie Visual Studio Code (VSCode) verwenden: *https://code.visualstudio.com*

Hinweis: VSCode wird in diesem Buch nicht behandelt.

9.2 Visual Studio-Versionen

Visual Studio gibt es in drei Varianten und zwei Kanälen. Varianten sind:

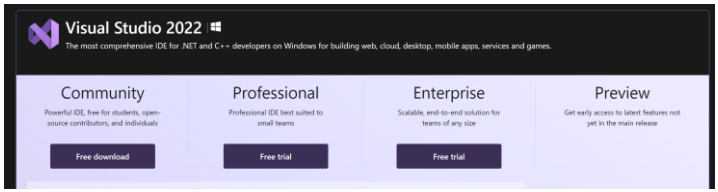
- Community
- Professional
- Enterprise

Die kostenfreie Community-Variante entspricht funktional der Professional-Variante. Allerdings darf die Community-Variante nur für Open Source-Projekte, von Studenten und kleineren Unternehmen eingesetzt werden.

Die Enterprise-Variante bietet zahlreiche zusätzliche Funktionen im Vergleich zu Professional und Community. Sie ist wesentlich teurer.

Hinweis: Für alle Inhalte in diesem Buch reichen die Varianten Professional bzw. Community.

Es gibt von jeder Visual Studio-Version stets zwei Kanäle: Den stabilen Kanal und den Preview-Kanal.



Sie können jeweils mehrere Visual Studio-Versionen und pro Kanal jeweils eine Unterversion parallel auf einem Windows-System installieren.

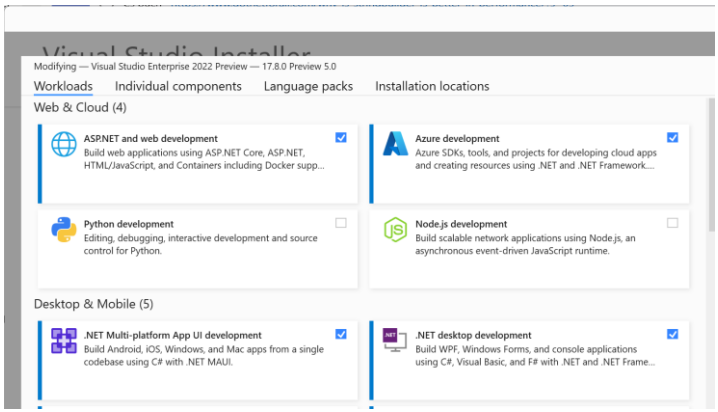
Beispiel: Auf einem Rechner ist möglich:

- Visual Studio 2019 Community Version 16.5
- Visual Studio 2022 Professional Version 17.7
- Visual Studio 2022 Enterprise Preview Version 17.12

Tipp 1: Installieren Sie die englische Version. In den deutschen Übersetzungen sind teilweise haarsträubende Übersetzungsfehler, die die Arbeit mit der Entwicklungsumgebung sehr erschweren.

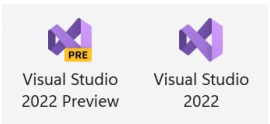
Tipp 2: Die Preview-Versionen sind immer kostenfrei. Ebenso gibt es kostenfrei Community-Versionen.

Wählen Sie bei der Installation von Visual Studio den Workload ".NET Desktop Development" aus. Die folgenden Screenshots zeigen Visual Studio 2022. Die Vorgehensweise in Visual Studio 2019 ist analog.



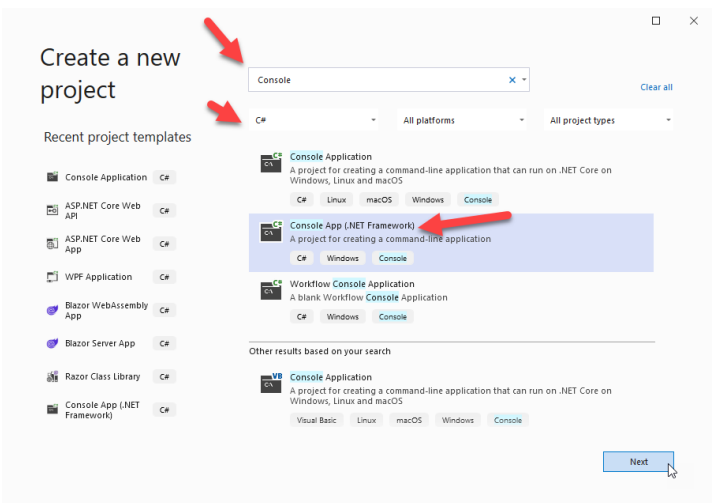
9.3 Hello World mit dem klassischen .NET Framework

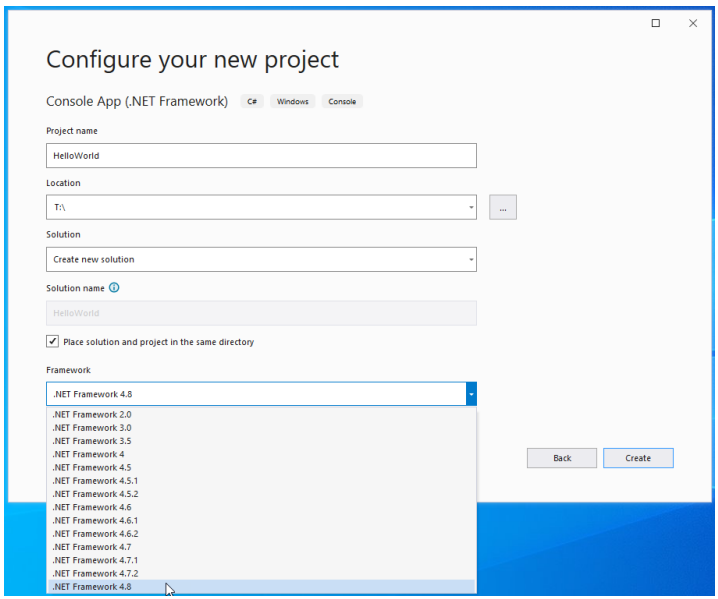
Starten Sie Visual Studio.



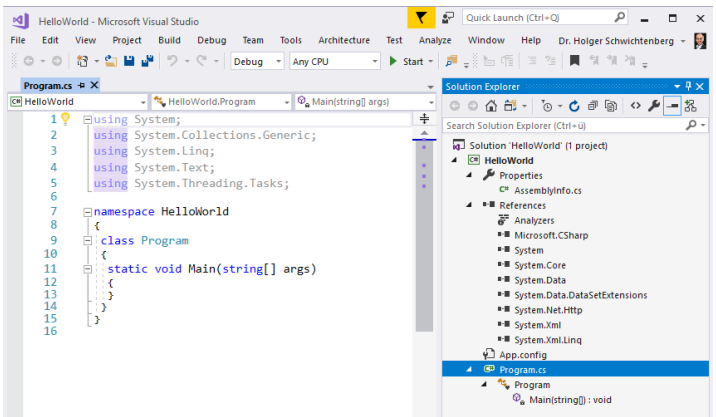
In Visual Studio wählen Sie File/New Project und dann in dem Dialog "Visual C#/Windows Classic Desktop/Console App".

Seit **Visual Studio 2019** wurde der Dialog komplett verändert und erscheint nur als Assistent mit zwei Seiten. Ein Konsolenprojekt findet man am leichtesten, wenn man in dem Suchfeld "Console" eingibt und den Filter auf "C#" stellt (siehe Screenshot). Die Auswahl der .NET Framework-Version kann man erst auf der zweiten Seite vornehmen.





Sie erhalten dann eine Projektmappe (.sln-Datei im Dateisystem) mit einem Projekt (.csproj-Datei). In dem Projekt gibt es eine Datei Program.cs mit der Grundstruktur der Konsolenanwendung.

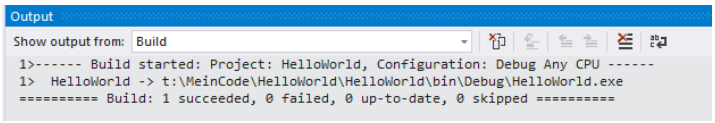


Ergänzen Sie in Main() den folgenden Programmcode:

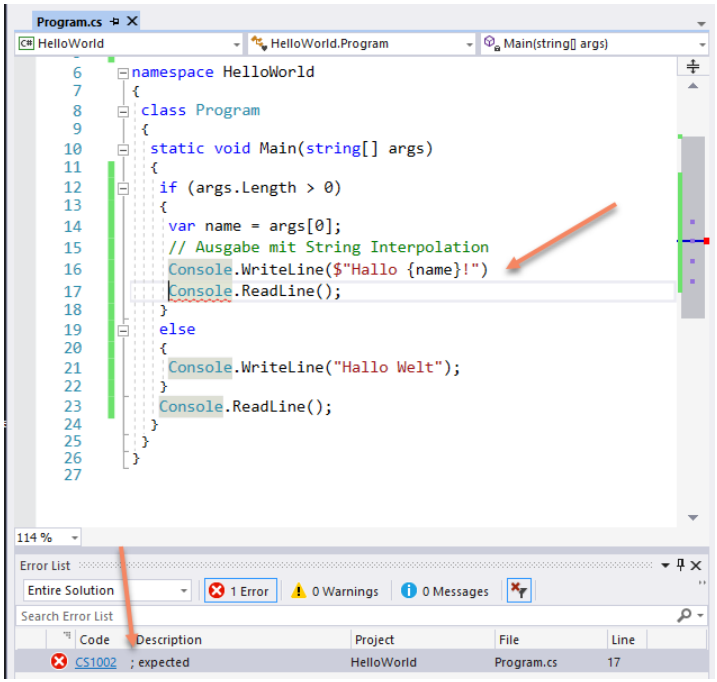
```
namespace HalloWelt
{
 class Program
 {
  static void Main(string[] args)
  {
   if (args.Length > 0)
   {
    var name = args[0];
    // Ausgabe mit String Interpolation
    Console.WriteLine($"Hallo {name}!");
    Console.ReadLine();
   }
   else
   {
    Console.WriteLine("Hallo Welt!");
   }
   Console.ReadLine();
  }
 }
}
```

Wählen Sie das Menü "Build/Build Solution" (Alternativ die Tastenkombination STRG+SHIFT+B), um den Programmcode zu übersetzen.

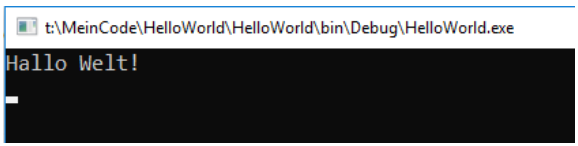
Sie sollten nun im Ausgabefenster (Einblenden über View/Output) dies sehen:



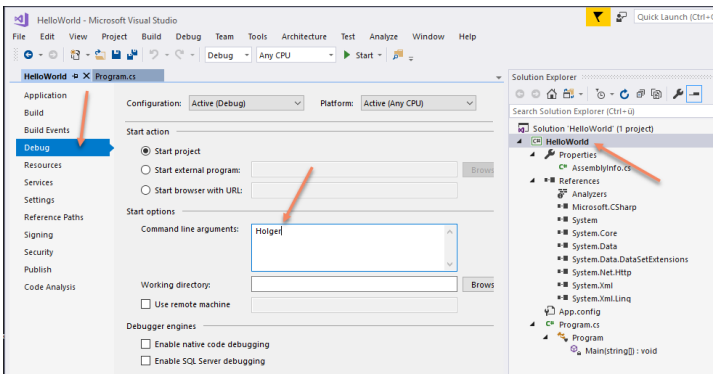
Falls Sie Eingabefehler gemacht haben, sehen Sie dies im Fenster "Error List".



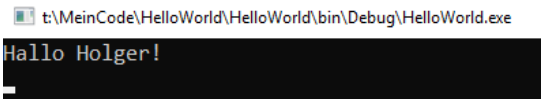
Wenn Ihr Programm erfolgreich übersetzt wurde, starten Sie es im Debugger mit Debug/Start Debugging oder der Taste F5.



Um dem Programm beim Start einen Kommandozeilenparameter zu übergeben, wählen Sie im Solution Explorer im Kontextmenü des Projekts (nicht der Projektmappe, wo "Solution" davor steht) den Eintrag "Properties" und tragen Sie in der Registerkarte "Debug" bei "Command Line Arguments" Ihren Namen ein.

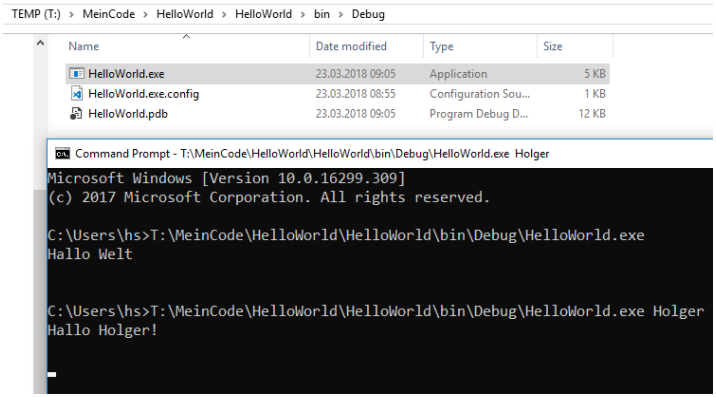


Drücken Sie wieder F5.



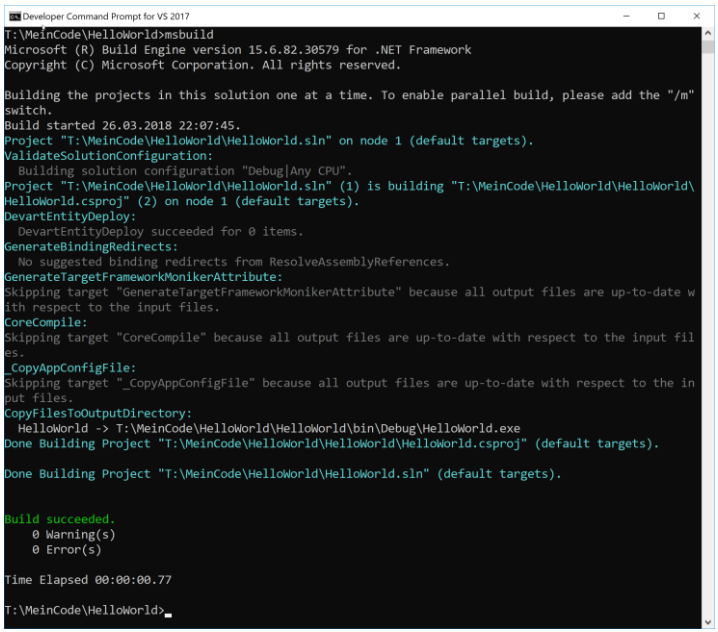
Schauen Sie sich das Projekt auf der Festplatte im Windows Explorer an. Sie erkennen ein Ausgabeverzeichnis bin/Debug in dem das kompilierte Programm als .EXE-Datei liegt, die man direkt starten kann.

Hinweis: Das Kompilat in .NET nennt man eine Assembly. Die Assembly ist in diesem Fall eine .EXE-Datei.



Sie können ein in Visual Studio erzeugtes .NET-Projekt auch an der Kommandozeile übersetzen. Theoretisch kann man dazu den C#-Compiler csc.exe direkt einsetzen, aber dann muss man alle Quellcodedateien sowie benötigte Referenzen auf andere Assemblies dort als Parameter angeben. Da diese Abhängigkeiten alle bereits in den Projektdateien definiert sind, bietet sich der Einsatz von msbuild.exe an, dass die .csproj-Dateien auswertet. Öffnen Sie dazu den "Developer Command Prompt", der mit Visual Studio installiert wird, gehen Sie in das Verzeichnis mit der .sln-Datei und rufen Sie msbuild.exe auf.

Hinweis: Andere .NET-Anwendungsarten (z.B. Webanwendungen mit ASP.NET, Desktop-Anwendungen mit Windows Forms oder Windows Presentation Foundation, Mobile Apps mit .NET MAUI) erstellen und übersetzen Sie mit den gleichen Funktionen und Werkzeugen. Sie müssen nur entsprechende Workloads im Setup von Visual Studio installieren und dann die entsprechende Projektvorlage wählen.



9.4 Hello World mit modernem .NET

Hier werden die Schritte beschrieben, die anders sind, wenn Sie das moderne .NET verwenden wollen statt .NET Framework. Dabei kommt Visual Studio 2022 zum Einsatz, denn die aktuellen .NET-Versionen ab 6.0 setzen diese Version voraus. Mit Visual Studio 2019 können Sie nur bis .NET 5.0 entwickeln.

Wichtig ist, dass Sie in Visual Studio den Workload "ASP.NET and Web Development" und/oder ".NET Desktop Development" wählen und zudem das .NET SDK in der aktuellen Version zusätzlich von [*dotnet.microsoft.com/download/dotnet*] installieren.

Hinweis: Es kann sein, dass Sie das aktuelle SDK schon durch Visual Studio installiert bekommen haben, da es aber häufig Updates des SDKs gibt, gehen Sie damit sicher, dass Sie die aktuelle Version haben.

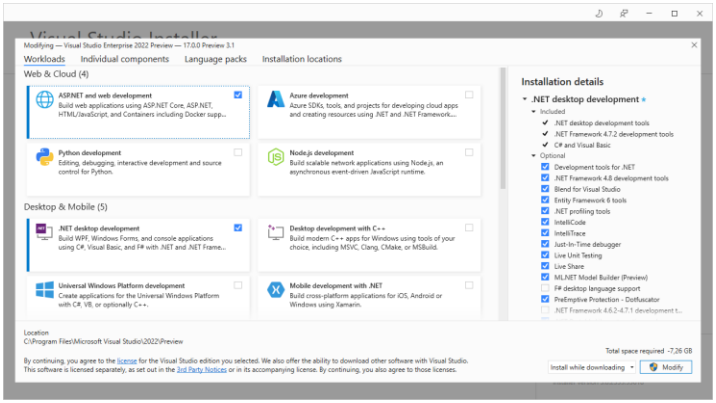


Abbildung: Installation von Workloads in Visual Studio 2022

Wählen Sie im Projektvorlagendialog (Menu File/New/Project oder Taste STRG+SHIFT+N) nun "Console Application" (ohne Zusatz).

Hinweis: In früheren Visual Studio-Versionen hatten die Vorlagennamen noch den Zusatz "(.NET Core)".

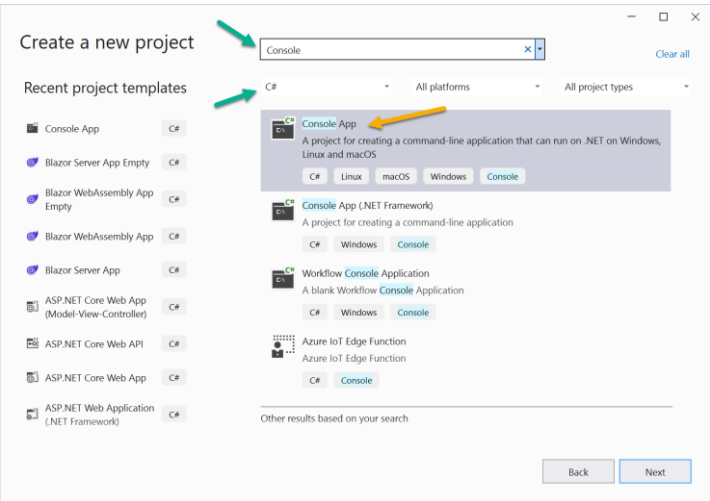


Abbildung: Konsolenprojekt anlegen in Visual Studio 2022

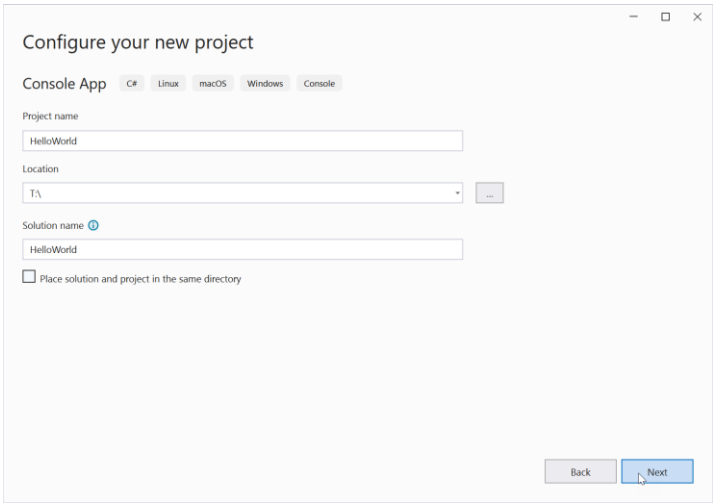


Abbildung: Optionen beim Anlegen eines Konsolenprojekt in Visual Studio 2022

Die zu verwendende .NET-Version kann man erst auf der dritten Seite wählen.

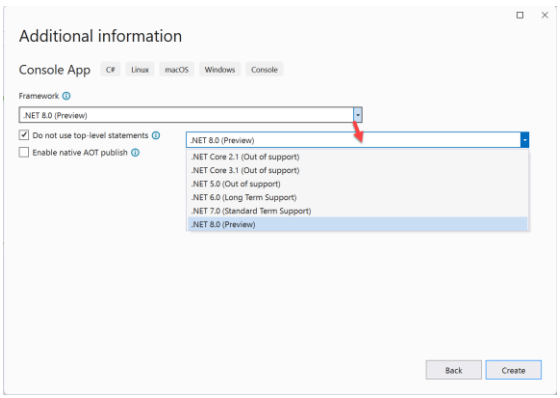


Abbildung: Weitere Optionen beim Anlegen eines Konsolenprojekt in Visual Studio 2022

Sie erhalten dann eine Projektmappe (.sln-Datei im Dateisystem) mit einem Projekt (.csproj-Datei) und einer Datei Program.cs. Der Projektaufbau eines modern .NET-Projekts ist etwas anders als

bei einem klassischen .NET-Projekt (z.B. Ast "Dependencies" statt "References"), die Bedienung bezüglich Übersetzung und Debugging sind aber gleich.

Bei der Struktur des erzeugten Codes gibt es zwei Möglichkeiten:

- Klassische Grundstruktur mit class Program und Methode Main() mit Parameter *args* für die übergebenen Kommandozeilenparameter
- Minimalcode mit Top-Level-Statements ohne Klasse und Methode. Auch in diesem minimalen C# 10-Konsolenprojekt kann man auf die Kommandozeilenparameter zugreifen: *args* ist jetzt eine "unsichtbare" deklarierte Variable.

Beides wird in den folgenden Abbildungen dargestellt.

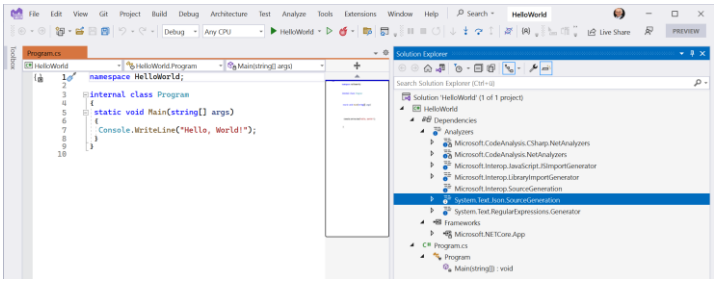


Abbildung: Klassische Grundstruktur einer Konsolenanwendung in Visual Studio 2019 bzw. Visual Studio 2022 seit Version 17.3 mit Option "Do not use top-level statements"

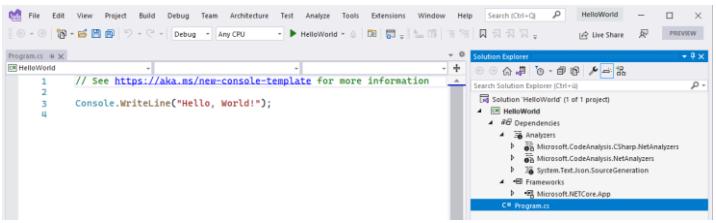


Abbildung: Minimal-Konsolenanwendung mit Top-Level-Statement in Visual Studio 2022

Welche Grundstruktur Sie erhalten, ist von der Version der eingesetzten Werkzeuge abhängig:

- Visual Studio 2019 und .NET SDK vor Version 6.0: Immer klassische Grundstruktur
- Visual Studio 2022 Versionen 17.1 und 17.2 sowie .NET SDK 6.0: Top-Level-Statements für viele (aber nicht alle) Projektarten, z.B. Konsolen- und Webanwendungen. WPF- und Windows Forms-Anwendungen werden weiterhin mit der klassischen Grundstruktur erstellt
- Visual Studio 2022 Versionen seit Version 17.3 sowie .NET SDK seit Version 7.0: Es gibt eine Option zur Abwahl der Top-Level-Statements (siehe oben "Do not use Top-Level-Statements").

In beiden Fällen gilt: Es gibt aber keine Namensraumimporte mehr: C# 10.0 bietet Implicit Namespace Imports für häufig genutzte Namensräume wie System, System.IO, System.Linq und System.Task.

Kommentar: Warum gibt es die reduzierten Vorlagen? Weil Microsoft Anfängern zeigen will, dass .NET sehr einfach ist – so einfach wie node.js. Ich bin **kein Fan** von diesem Minimalismus und der args-"Magie".

Es war mein Wunsch, dass es eine Auswahl der Entwickler zwischen klassischer Struktur und Minimal-Projekt gibt. Seit Visual Studio 2022 Version 17.3 ist mein Wunsch implementiert!

Egal wie die Struktur des Codes aussieht, der Start erfolgt gleich: Starten Sie die Anwendung im Debugger mit Debug/Start Debugging oder der Taste F5.

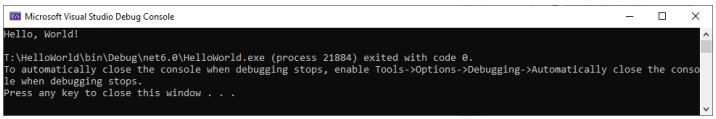


Abbildung: Start der Konsolenanwendung aus Visual Studio heraus

Hinweis: Bei modernen .NET-Projekten endet eine im Debugger gestartete Konsolenanwendung nicht automatisch, sondern wartet auf einen Tastendruck, siehe Screenshot. Wenn Sie die Konsolenanwendung aber außerhalb von Visual Studio starten, endet die Anwendung nach der Abarbeitung des Programmcodes sofort, außer wenn Sie mit Console.ReadLine() auf eine Eingabe warten.

Ein modernes .NET Core-/.NET-Projekt können Sie auch an der Kommandozeile mit *msbuild.exe* oder *dotnet build* übersetzen.

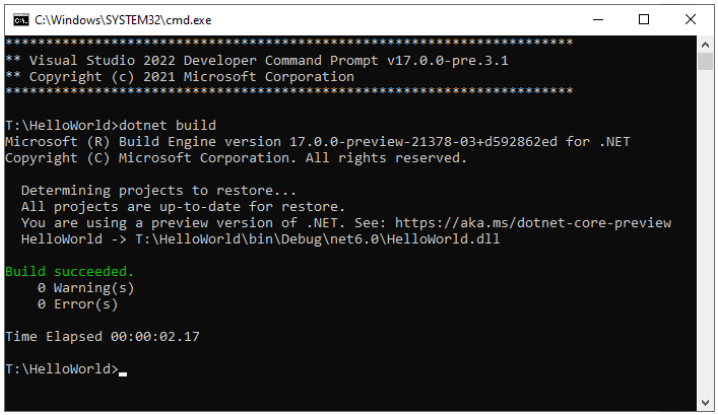


Abbildung: Übersetzung der Konsolenanwendung mit dotnet build

Hinweis: Andere moderne .NET-Anwendungsarten (z.B. Webanwendungen mit ASP.NET Core, Universal Windows Platform Apps) erstellen und übersetzen Sie mit den gleichen Funktionen und Werkzeugen. Sie müssen nur entsprechende Workloads im Setup von Visual Studio installieren und dann die entsprechende Projektvorlage wählen.

In älteren .NET Core-Versionen (vor .NET Core 3.0) sah man in der Titelzeile dotnet.exe, das universelle Kommandozeilenwerkzeug von .NET Core, dass auch zum Start einer .NET Core-Anwendung verwendet wurde. Während man beim .NET Framework im Ausgabeverzeichnis immer eine .EXE-Datei erhielt, bekam man bei .NET Core nur eine .DLL. Daher muss man dotnet.exe (oder abgekürzt dotnet) beim Start voranstellen. Die aktuelleren Versionen erzeugen aber wieder direkt Executables und brauchen dotnet.exe nicht mehr als Starthilfe. Auch in den aktuellen Versionen wird aber immer neben der EXE eine DLL erzeugt, die man über dotnet.exe starten kann (siehe folgende Abbildung).

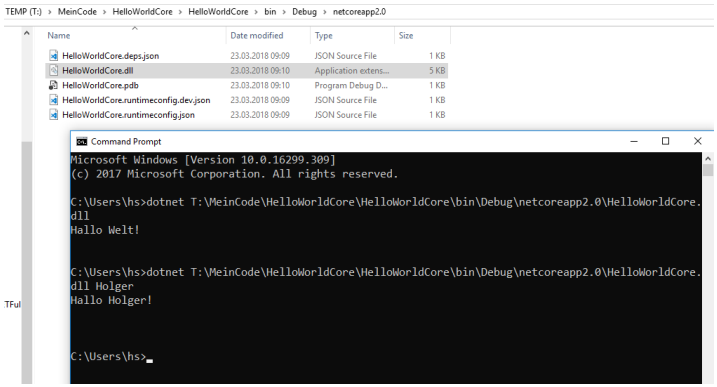


Abbildung: Start der Konsolenanwendungen in älteren .NET Core-Versionen

9.5 Programme ohne Main() (Top-Level Statements)

Seit C# 9.0 ist die Verwendung einer Einsprungmethode Main() nicht mehr verpflichtend. Man kann als Startcode der Anwendung auch direkt freien Programmcode in eine beliebige .cs-Datei schreiben, z.B.

```
using System;
Console.WriteLine("Hello Word");
```

oder sogar in eine Zeile

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

Intern erzeugt der C#-Compiler aus den Top-Level-Statements doch wieder eine class Program mit Main()-Methode.

Beispiel: Aus diesem Programm

```
string GetNETVersion()
{
 return System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription;
}

CUI.H1("C# Top-Level Statements (seit C# 9.0)");
Console.WriteLine(GetNETVersion());
Console.ReadLine();
```

erzeugt der Compiler

```
[CompilerGenerated]
internal class Program
{
    private static void <Main>$(string[] args)
    {
        CUI.H1("C# Top-Level Statements (seit C# 9.0)");
        Console.WriteLine(GetNETVersion());
        Console.ReadLine();
        static string GetNETVersion()
        {
            return RuntimeInformation.FrameworkDescription;
        }
    }
}
```

Abbildung: Decompilierung eines Top-Level-Statements mit ILSpy [https://github.com/icsharpcode/ILSpy]

Sofern es innerhalb des Top-Level-Codes ein await gibt

```
string GetNETVersion()
{
 return System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription;
}

CUI.H1("C# Top-Level Statements (seit C# 9.0)");
Console.WriteLine("Hole Daten...");
await System.Threading.Tasks.Task.Delay(1000);
Console.WriteLine(GetNETVersion());
Console.ReadLine();
```

erzeugt der C#-Compiler automatisch eine den Einsprungpunkt Main() mit dem Zusatz async:

```
[CompilerGenerated]
internal class Program
{
    private static async Task <Main>$(string[] args)
    {
        CUI.H1("C# Top-Level Statements (seit C# 9.0)");
        Console.WriteLine("Hole Daten...");
        await Task.Delay(1000);
        Console.WriteLine(GetNETVersion());
        Console.ReadLine();
        static string GetNETVersion()
        {
            return RuntimeInformation.FrameworkDescription;
        }
    }
}
```

Abbildung: Decompilierung eines Top-Level-Statements mit einem asynchronen Aufruf

Es darf natürlich in einem C#-Projekt nicht mehr als eine Datei geben, die solch freien Code enthält, sonst wäre der Einsprungpunkt der Anwendung nicht mehr eindeutig. Der Compiler beschwert sich dann "Error CS8802: Only one compilation unit can have top-level statements."

Zu beachten ist auch, dass in der Datei mit dem Top-Level-Statement keine Namensraumdeklaration mit File-Scoped Namespaces erfolgen kann (CS0116: A namespace cannot directly contain members such as fields, methods or statements) und die Programmstart-Befehle vor allen in der Datei ebenfalls noch möglichen Typdeklarationen stehen müssen (CS8803: Top-level statements must precede namespace and type declarations).

Falls es ein Top-Level Statement und ein void Main() gibt, wird void Main() ignoriert!

Praxishinweis: Der Einsatz dieses Sprachfeatures ist umstritten. Microsoft schreibt dazu: "One of the most common uses for this feature is creating teaching materials. Beginner C# developers can write the canonical “Hello World!” in one or two lines of code. None of the extra ceremony is needed."

Der Autor dieses Buchs sieht allerdings mit Top-Level-Statement die Gefahr, dass der Code unübersichtlicher wird. Der Einsprungpunkt einer Anwendung ist nicht mehr auf Anhieb zu finden und man schreibt leicht aus Versehen ein Top-Level-Statement! Der Autor dieses Buchs sieht in dem Weglassen von Main() allenfalls ein Einsatzgebiet und zwar im Einsatz von C# als Skriptsprache, wo das Skript nur aus einer Datei besteht.

In .NET 6.0 hatte Microsoft begonnen, in vielen modernen Projektvorlagen nur mit Top-Level-Statements zu arbeiten. Seit .NET 7.0 bzw. Visual Studio 2022 Version 17.3 hat der Entwickler wieder die Wahl.

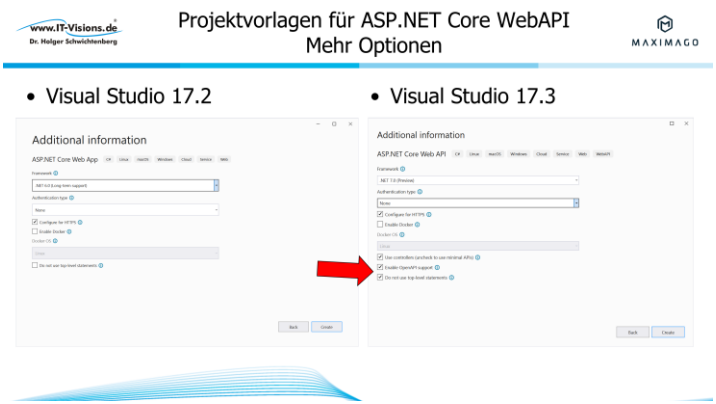


Abbildung: WebAPI-Projekte anlegen in Visual Studio 17.2 versus 17.3

Beim Kommandozeilenbefehl dotnet new gibt es ab .NET 7.0 dafür nun den Parameter --use-program-main

z.B.

```
dotnet new console --use-program-main
```

und

```
dotnet new webapi --use-program-main
```

9.6 Festlegung der Compilerversion

Während früher die verwendete Visual Studio-Version auch die verwendete Version des Sprachcompilers von C# festlegte, kann man seit Visual Studio 2017 die Sprachversion pro Projekt in den Projekteigenschaften (Build/Advanced) festlegen.

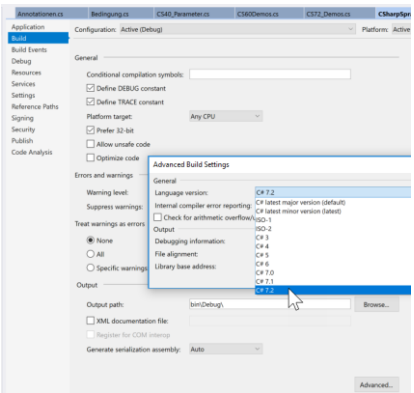
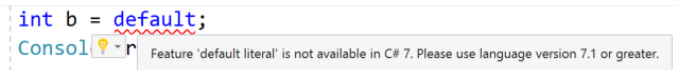


Abbildung: Einstellen der Sprachversion

Zudem warnt Visual Studio, wenn Sie ein Sprachfeature verwenden, welches es in der eingestellten Version noch nicht gibt.



Seit Visual Studio 2019 hat Microsoft diese freie Auswahl wieder abgeschafft. Nun legt die verwendete Framework-Version eine bestimmte Compiler-Version fest.

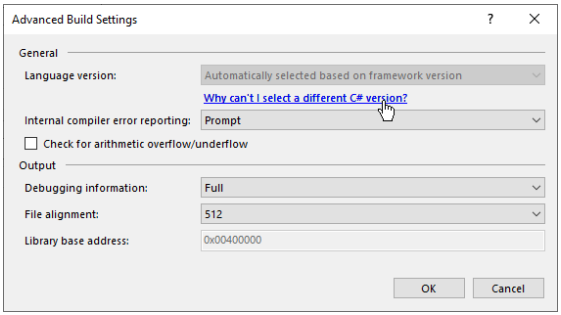


Abbildung: Auswahl der Sprachversion in Visual Studio 2019

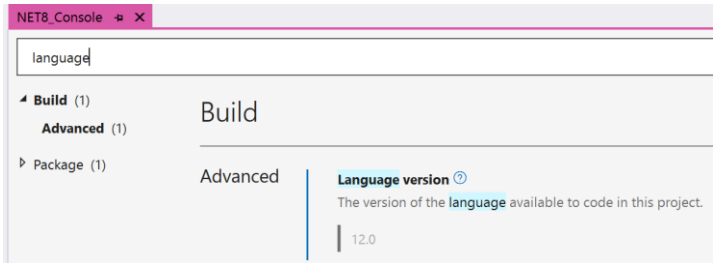


Abbildung: Informationen zur automatisch eingestellten Sprachversion-Version in Visual Studio 2022

Der Link führt zu [*learn.microsoft.com/de-de/dotnet/csharp/language-reference/configure-language-version*] und dort steht:

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

Abbildung: Mögliche Einstellungen für <LangVersion>
(Quelle: learn.microsoft.com/de-de/dotnet/csharp/language-reference/configure-language-version)

Um eine bestimmte Version der C#-Sprachsyntax zu erzwingen, kann man aber die Projektdatei manuell bearbeiten und dort mit dem Tag <LangVersion> eine bestimmte Version erzwingen. So ist es zum Beispiel möglich, in .NET Framework und .NET Standard auch Sprachsyntaxelemente aus C# 8.0 und höher zu verwenden.

Hinweis: In einem .NET 9.0-Projekt (Projekteinstellung: `<TargetFramework>net9.0</TargetFramework>`) ist C#-Sprachversion 13.0 der automatisch eingestellte Standard, auch ohne Tag `<LangVersion>`.

Die <LangVersion> legt man in den Projekteinstellungen (in der Datei .csproj) fest. Man kann auf diese Weise neuere aber auch ältere Sprachversionsnummern erzwingen.

Tipp: Bei modernen .NET-SDK-Projekten kann man die .csproj-Datei einfach bearbeiten, indem man einen Doppelklick auf der Projektdatei im Solution Explorer macht. Bei klassischen

.csproj-Dateien muss man erst "Unload Project" und dann "Edit Project File" wählen. Nach der Bearbeitung muss man "Reload Project" ausführen.

Listing: Setzen der <LangVersion> in modernen .NET-SDK-Projekten.
Hier: Upgrade von C# 12.0 auf C# 13.0 in einem .NET 8.0-Projekt

```
<PropertyGroup>
  <TargetFramework>.net8.0</TargetFramework>
  <LangVersion>13.0</LangVersion>
</PropertyGroup>
```

Listing: Setzen der <LangVersion> in klassischen .NET Framework-Projekten
Hier: Upgrade von C# 7.2 auf C# 13.0 in einem .NET Framework-Projekt

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
..
    <LangVersion>13.0</LangVersion>
</PropertyGroup>

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
..
    <LangVersion>13.0</LangVersion>
</PropertyGroup>
```

Wichtig: In den klassischen .csproj-Dateien, die .NET Framework verwendet, ist die <LangVersion> pro Compilerkonfiguration zu setzen.

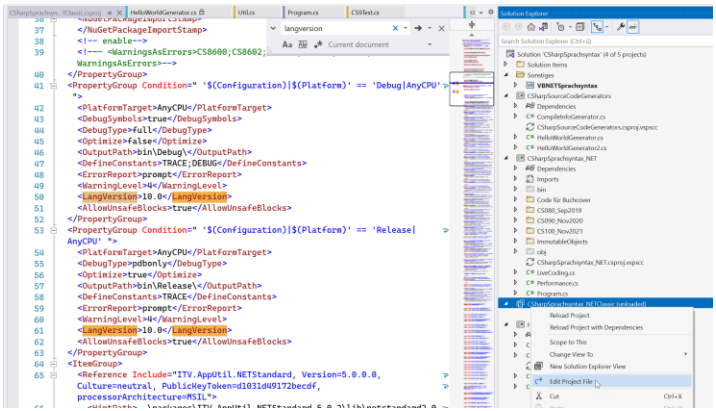


Abbildung: Bearbeitung der <LangVersion> in einem klassischen .NET Framework-Projekt nach dem "Unloading" des Projekts

Wenn ein Entwickler Sprachelemente verwendet, die gemäß aktuell gültiger Sprachversion nicht verfügbar sind, meckert der Compiler. Manchmal steht auch dabei, welche Sprachversion notwendig wird.

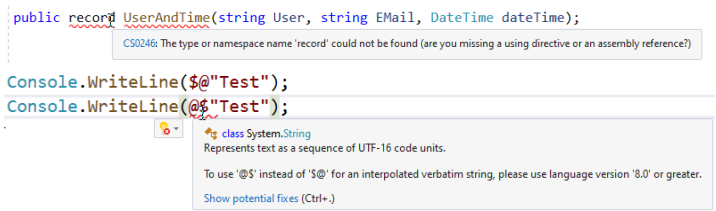


Abbildung: Fehlermeldung, wenn eine zu niedrige Sprachversion verwendet wird

Hinweis: Neben der Einstellung der <LangVersion> sind zum Teil weitere Tricks erforderlich, um neuere Sprachversionen auf älteren, von Microsoft nicht für die aktuellen C#-Sprachversionen unterstützten .NET-Versionen nutzen zu können. Sie finden darauf jeweils Hinweise in den einzelnen Kapiteln, siehe z.B. Kapitel "Init Only Setters in .NET Framework und .NET Standard".

9.7 Eingabeunterstützung in Visual Studio

Visual Studio unterstützt den Entwickler mit Hilfsfunktionen bei der Programmcodeeingabe.

9.7.1 IntelliSense

Die IntelliSense-Eingabeunterstützung, die kontextabhängige Vorschläge für Bezeichner und Klassenmitglieder macht, gibt es nicht erst seit der ersten Visual Studio-Version im Jahr 1997, sondern sie gab es auch schon in den Vorgängerprodukten (Visual C++, Visual Basic, Visual FoxPro etc). Seit dem Jahr 2018 gibt es mit IntelliCode [*hwww.heise.de/developer/meldung/Build-2018-IntelliCode-C-Eingabeunterstuetzung-mit-kuenstlicher-Intelligenz-4044483.html*] eine Zusatzfunktion, die aus dem Kontext heraus häufig verwendete Klassenmitglieder hervorhebt.

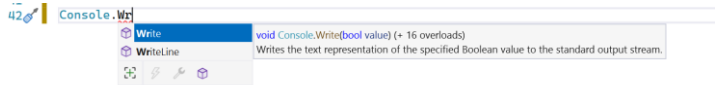


Abbildung: IntelliSense-Vorschläge in Visual Studio 2022

9.7.2 IntelliCode

In Visual Studio 2022 macht Microsoft erstmals nicht nur Vorschläge für einzelne Bezeichner, sondern aus dem aktuellen Kontext heraus für vollständige Programmzeilen (siehe Abbildungen). Microsoft nennt diese Funktion IntelliCode. Mit einem doppelten Drücken auf die Tabulator-Taste übernimmt der Entwickler den Vorschlag. Die Vorschläge basieren dabei auf dem KI-Training mit dem Quellcode einer halben Million Open-Source-Projekten auf GitHub. Details zu dieser erweiterten IntelliCode-Funktion findet man in einem Blogeintrag [*devblogs.microsoft.com/visualstudio/type-less-code-more-with-intellicode-completions*].

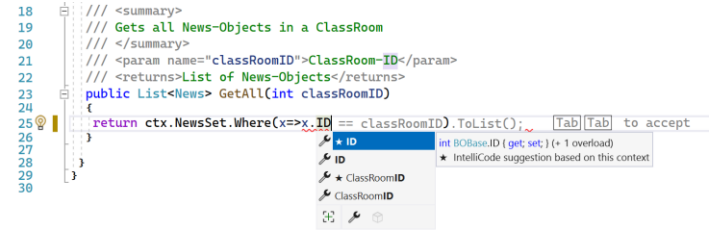


Abbildung: IntelliCode-Zeilenvorschläge in Visual Studio 2022

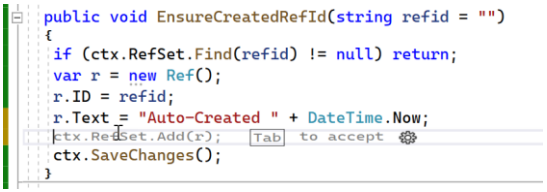


Abbildung: IntelliCode-Zeilenvorschläge in Visual Studio 2022

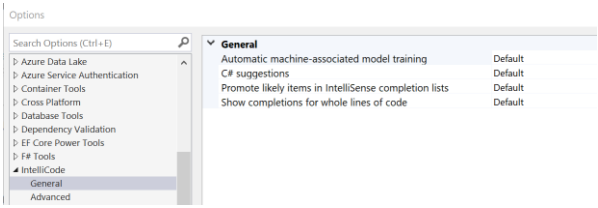


Abbildung: IntelliCode-Einstellungen in Visual Studio 2022

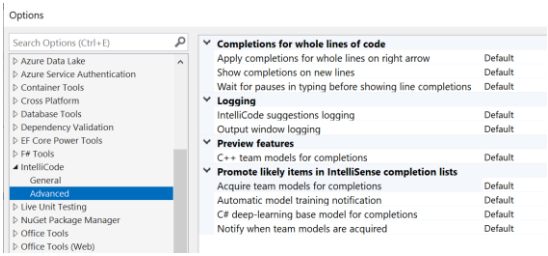


Abbildung: IntelliCode-Einstellungen in Visual Studio 2022

9.7.3 Copilot

Noch umfangreichere Vorschläge (mehrzeilige Codeblöcke / ganze Methoden auf Basis von Kommentaren) erhalten Sie mit GitHub Copilot, das wie ChatGPT auf den KI-Modellen von OpenAI basiert. Allerdings ist dazu ein kostenpflichtiges Copilot-Abo bei GitHub erforderlich (ab 10 Euro/Monat, Ausnahmen gelten für Studenten und Open Source-Projekte):

https://github.com/features/copilot

Für Visual Studio müssen Sie für Copilot eine Erweiterung installieren:

https://marketplace.visualstudio.com/items?itemName=GitHub.copilotvs

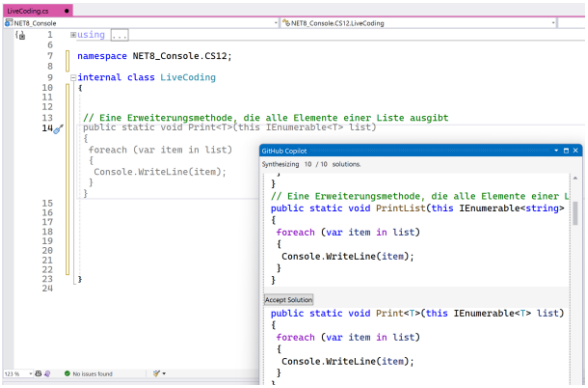


Abbildung: GitHub-Copilot schlägt auf Basis einer vom Entwickler erfassten Kommentarzeile eine ganze Methode vor. Den Vorschlag kann man mit Drücken der Tabulatortaste übernehmen. Im Copilot-Fenster sieht man weitere Alternativen, die man durch Klick auf "Accept Solution" übernehmen kann.

9.8 Refactoring in Visual Studio

Während man in den Anfangsjahren von .NET für das effiziente Refactoring (Umgestalten) von Code Zusatzsoftware wie ReSharper von JetBrains [*https://www.jetbrains.com/resharper*] oder CodeRush von Developer Express [*https://www.devexpress.com/products/coderush*] zwingend brauchte, bietet Visual Studio inzwischen zahlreiche integrierte Refactoring-Funktionen.

Die Refactoring-Funktionen findet man in der Glühbirne neben den Zeilennummern.

Tipp: Sofern die Glühbirne nicht automatisch erscheint, drücken Sie die Tasten STRG und . zusammen.

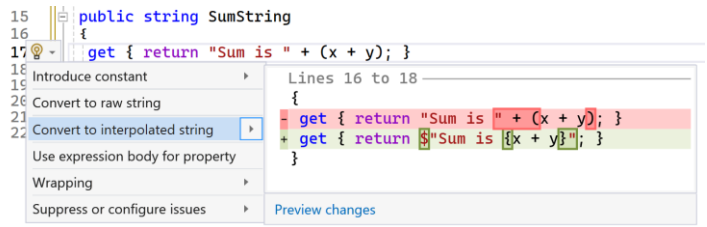


Abbildung: Vorschlag zur Umwandlung der Zeichenkette in eine interpolierte Zeichenkette

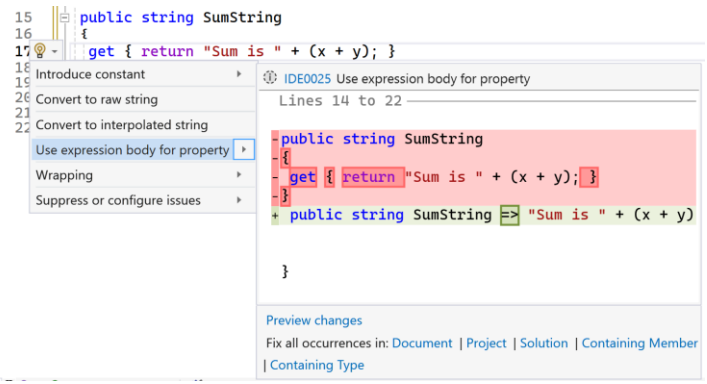


Abbildung: Vorschlag zur Umwandlung des Getters des Properties in einem Lambda-Ausdruck

9.9 .NET Fiddle

Eine Möglichkeit, C#-Code auf einfache Weise auszuprobieren, ist die Website .NET Fiddle [*https://dotnetfiddle.net*].

Hier kann man C#-Code eingeben und innerhalb der Webseite ausführen. Dabei kann man die .NET-Version wählen.

Ein Zugriff auf lokale Ressourcen auf dem PC des Entwicklers ist freilich wegen der Sandbox des Webbrowsers nicht möglich.

Angemeldete Benutzer können den Programmcode speichern.

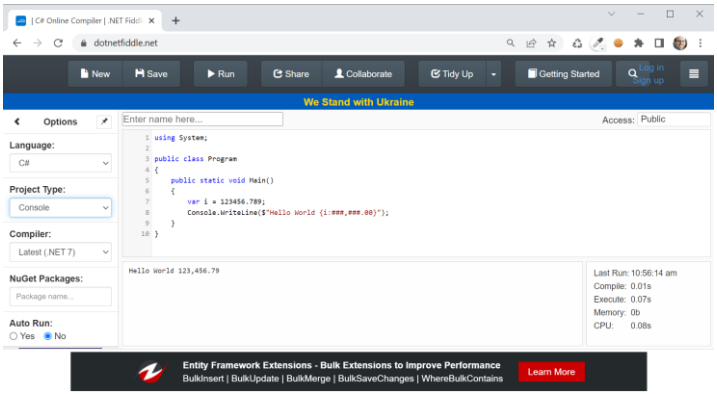


Abbildung: Test einer formatierten Ausgabe mit Console.WriteLine und interpolierter Zeichenkette in .NET Fiddle

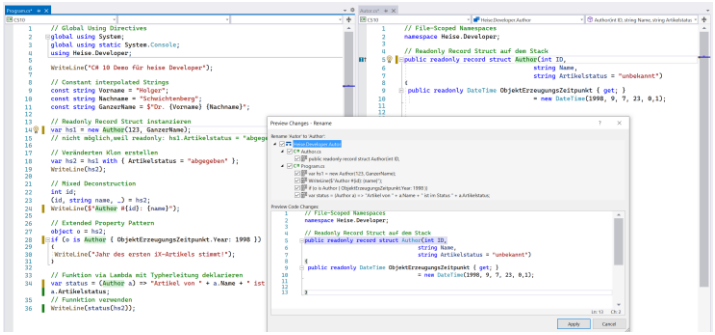


Abbildung: Umbenennen eines Typs (Refactoring "Rename")

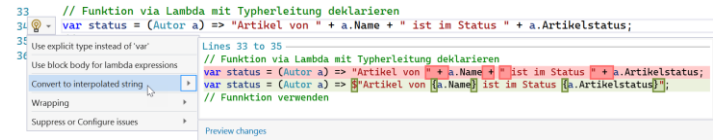


Abbildung: Umwandlung in eine interpolierte Zeichenkette (Refactoring "Convert to Interpolated String")

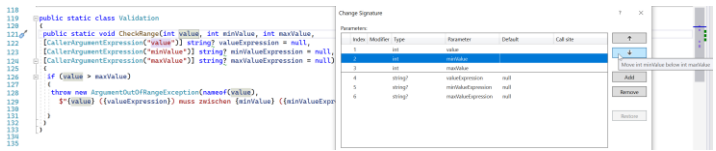


Abbildung: Ändern einer Methodensignatur (Refactoring "Change Signature")

10 Datentypen

Die Datentypen orientieren sich in allen .NET-Programmiersprachen basierend auf den in der .NET-Basisklassenbibliothek implementierten Datentypen. Innerhalb der Programmiersprache kann es für einen .NET-Basistyp einen Alias geben. Der Entwickler kann wahlweise entweder den Klassennamen oder den Alias verwenden, auch gemischt.

```
System.String vorname = "Holger"; // Variable typisiert mit Basisklasse
string nachname = "Schwichtenberg"; // Variable typisiert mit C#-Alias
System.String GanzerName = vorname + " " + nachname;
string GanzerNameUmgekehrt = nachname + ", " + vorname;
```

10.1 Überblick über die Datentypen

Die folgende Tabelle gibt einen Überblick über die wichtigsten Datentypen in C#.

Datentyp	.NET-Basisklasse	Alias in C#	Alias in Visual Basic .NET
Boolean	System.Boolean	bool	Boolean
Ganzzahl 1 Byte	System.Byte	byte	Byte
Ganzzahl 2 Bytes	System.Int16	short	Short
Ganzzahl 4 Bytes	System.Int32	int	Integer
Ganzzahl 8 Bytes	System.Int64	long	Long
Ganzzahl 16 Bytes	System.Int128 (seit .NET 7.0)	---	---
Gebrochene Zahlen 2 Bytes	System.Half (seit .NET 7.0)	---	---
Gebrochene Zahlen 4 Bytes	System.Single	float	Single
Gebrochene Zahlen 8 Bytes	System.Double	double	Double
Gebrochene Zahlen 12 Bytes	System.Decimal	decimal	Decimal
Zeichen 1 Byte oder 2 Bytes	System.Char	char	Char
Zeiger (Numeric Integer Pointer)	System.IntPtr	nint (seit C# 11.0)	---
Zeichenkette (UTF-16 codiert)	System.String	string	String
Datum / Uhrzeit	System.DateTime	DateTime	Date

Tabelle: Vergleich der wichtigsten Datentypen in .NET, C# und Visual Basic .NET

Hinweis: Für die Ganzzahltypen und die Zeiger gibt es jeweils auch eine Variante mit dem Vorbuchstaben "u" wie "Unsigned", also uint ist Alias für System.UInt32. Diese Zahlen haben einen Wertebereich von 0 beginnend.

Eine Bennungsausnahme ist System.UIntPtr: Hier heißt der Alias nicht unint, sondern nuint.

Eine weitere Ausnahme ist der Typ byte bzw. System.Byte: Der Wertebereich liegt von 0 bis 255. Wenn man negative Zahlen ausdrücken will mit einem Byte, gibt es den Typ System.SByte mit dem alias sbyte (Wertebreich -128 bis 127).

C#-Typ/Schlüsselwort	Bereich	Größe	.NET-Typ
`sbyte`	–128 bis 127	Ganze 8-Bit-Zahl mit Vorzeichen	System.SByte
`byte`	0 bis 255	8-Bit-Ganzzahl ohne Vorzeichen	System.Byte
`short`	–32.768 bis 32.767	Ganze 16-Bit-Zahl mit Vorzeichen	System.Int16
`ushort`	0 bis 65.535	16-Bit-Ganzzahl ohne Vorzeichen	System.UInt16
`int`	-2,147,483,648 bis 2,147,483,647	Eine 32-Bit-Ganzzahl mit Vorzeichen	System.Int32
`uint`	0 bis 4.294.967.295	32-Bit Ganzzahl ohne Vorzeichen	System.UInt32
`long`	-9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807	64-Bit-Ganzzahl mit Vorzeichen	System.Int64
`ulong`	0 bis 18.446.744.073.709.551.615	64-Bit-Ganzzahl ohne Vorzeichen	System.UInt64
`nint`	Hängt von der Plattform ab (berechnet zur Laufzeit)	32-Bit- oder 64-Bit-Integerwerte mit Vorzeichen	System.IntPtr
`nuint`	Hängt von der Plattform ab (berechnet zur Laufzeit)	32-Bit- oder 64-Bit-Integerwerte ohne Vorzeichen	System.UIntPtr

Abbildung: Wertebereich der Ganzzahl-Datentypen (Quelle: [https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/integral-numeric-types]).

C#-Typ/Schlüsselwort	Ungefährer Bereich	Genauigkeit	Größe	.NET-Typ
`float`	$\pm 1.5 \times 10^{-45}$ zu $\pm 3.4 \times 10^{38}$	~6–9 Stellen	4 Bytes	System.Single
`double`	$\pm 5,0 \times 10^{-324}$ bis $\pm 1,7 \times 10^{308}$	~15–17 Stellen	8 Bytes	System.Double
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 Stellen	16 Bytes	System.Decimal

Abbildung: Wertebereich der Fließkommazahl-Datentypen (Quelle: [https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types]).

10.2 Variablendeklarationen

In C# steht der Typ am Anfang jeder Deklaration. Mehrfachdeklarationen sind möglich durch Kommatrennung.

```
int a, b, c;
string x, y, z;
System.Guid g1, g2, g3;
```

10.3 Typinitialisierung

Ebenfalls sehr streng ist C# hinsichtlich der Initialisierung von Variablen. Während der Visual Basic .NET-Compiler in seiner Standardeinstellung folgende Anweisung immer durchgehen lässt,

```
Dim a As Integer
a = a + 1
```

weil a mit 0 vorinitialisiert wurde, erfordert der C#-Compiler die explizite Initialisierung bei allen lokalen (methodeninternen) Variablen (nicht aber bei Klassenmitgliedern).

```
int a = 0;
a = a + 1;
```

```
int a;
Console.WriteLine(a);
```

(local variable) int a

Use of unassigned local variable 'a'

Abbildung: Der C#-Compiler beschwert sich über die Verwendung einer nicht initialisierten Variable

Hinweis: Der C#-Compiler seit Version 2005 erzeugt Warnungen bei deklarierten, aber nicht verwendeten Variablen.

Mit dem Schlüsselwort default kann man eine Variable auf ihren Standardwert setzen. Dies ist 0 für alle Zahlen und null für Zeichenketten und Referenztypen. Für Datumswerte ist es der 01.01.0001 um 00:00:00 Uhr. Während in den bisherigen Versionen die Syntax vorsah, nach default in Klammern den Datentyp zu nennen

```
int x = default(Int32);
```

kann man diesen seit C# 7.1 weglassen (Default Literal Expressions):

```
int x = default;
```

Beispiele:

```
decimal zahl1 = default(decimal); // 0.0
decimal zahl2 = default; // 0.0
int ganzzahl1 = default(int); // 0
int ganzzahl2 = default; // 0
bool janein1 = default(bool); // false
bool janein2 = default; // false
string zeichenkette1 = default(string); // null
string zeichenkette2 = default; // null
Person person1 = default(Person); // null
Person person2 = default; // null
DateTime d1 = default(DateTime); // 01.01.0001 00:00:00
DateTime d2 = default;// 01.01.0001 00:00:00
```

10.4 Literale für Zeichen und Zeichenketten

Zeichenketten sind in doppelte Anführungszeichen zu setzen. Zeichenketten werden in .NET intern als Folge von Bytes in UTF-16-Codierung abgelegt.

Einzelne Zeichen, in einfache Anführungszeichen.

```
string Name = "Holger Schwichtenberg";
string Wichtigkeit1 = "A";
char Wichtigkeit2 = 'C';
```

Sonderzeichen in Zeichenketten werden – wie in C++ – durch einen Backslash (\) eingeleitet (z.B. steht \n für einen Zeilenumbruch). Man spricht von Escapesequenz (siehe Tabelle).

Escape-Sequenz	Bedeutung
\a	Ton
\b	Rücktaste
\f	Seitenvorschub
\n	Zeilenwechsel
\r	Wagenrücklauf
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Umgekehrter Schrägstrich
\?	Literales Fragezeichen
\xhh	ASCII-Zeichen in der Hexadezimalnotation
\xhhhh	Unicode-Zeichen in der Hexadezimalnotation

Beispiel 1:

```
string seineAussage = "Er sagte:\n\"Hallo Welt!\"";
Console.WriteLine(seineAussage);
```

```
Er sagte:
"Hallo Welt!"
```

Beispiel 2:

```
Console.WriteLine("\x48\x6f\x6c\x67\x65\x72 \x53\x63\x68\x77\x69\x63\x68\x74\x65\
x6e\x62\x65\x72\x67");
```

```
Holger Schwichtenberg
```

Da der Backslash in der Zeichenkette ein Sonderzeichen darstellt, müssen Pfadangaben besonders behandelt werden.

Richtig ist, entweder für jeden Backslash \ einen doppelten Backslash \\ zu verwenden oder aber die Zeichenkette mit einem @ einzuleiten. @ leitet eine "wortgetreue Zeichenkette" (Verbatim String) ein. Dadurch verlieren alle Escapesequenzen ihre Bedeutung und der Backslash ist wieder ein normales Zeichen. Synonym sind daher: "c:\\ordner\\datei.txt" und @"c:\ordner\datei.txt".

```
string PfadRichtig1 = "C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319";
string PfadRichtig2 = @"C:\Windows\Microsoft.NET\Framework64\v4.0.30319";
Console.WriteLine(PfadRichtig1 + ": "+ System.IO.Directory.Exists(PfadRichtig1));
```

```
Console.WriteLine(PfadRichtig1 + ": " +
System.IO.Directory.Exists(PfadRichtig1));
```

10.5 Konsolenausgabenformatierung mit ANSI-Codes

Mit den uralten VT100/ANSI-Codes (siehe *https://en.wikipedia.org/wiki/ANSI_escape_code*) kann man auch heute noch in Konsolenanwendungen zahlreiche Formatierungen auslösen, z.B. 24-Bit-Farben, Fettschrift, Unterstreichen, Durchstreichen, Blinken usw. Die VT100/ANSI-Codes werden durch das ESCAPE-Zeichen (ASCII-Zeichen 27, hexadezimal: 0x1b) eingeleitet.

Vor C# 13.0 konnte man dieses ESCAPE-ASCII-Zeichen 27 in .NET-Konsolenanwendungen bei Console.WriteLine() nur umständlich ausdrücken über \u001b, \U0000001b oder \x1b, wobei letzteres nicht empfohlen ist: "Wenn Sie die Escapesequenz \x verwenden, weniger als vier Hexadezimalziffern angeben und es sich bei den Zeichen, die der Escapesequenz unmittelbar folgen, um gültige Hexadezimalziffern handelt (z. B. 0–9, A–F und a–f), werden diese als Teil der Escapesequenz interpretiert. \xA1 erzeugt beispielsweise "¡" (entspricht dem Codepunkt U+00A1). Wenn das nächste Zeichen jedoch "A" oder "a" ist, wird die Escapesequenz stattdessen als \xA1A interpretiert und der Codepunkt "ਚ" erzeugt (entspricht dem Codepunkt U+0A1A). In solchen Fällen können Fehlinterpretationen vermieden werden, indem Sie alle vier Hexadezimalziffern (z. B. \x00A1) angeben." [*https://learn.microsoft.com/de-de/dotnet/csharp/programming-guide/strings/*].

Hinweis: ਚ ist ein Panjabi-Schriftzeichen. Panjabi ist eine in Pakistan und Indien gesprochene Sprache.

Typischerweise sahen Ausgaben mit VT100/ANSI-Escape-Codes dann aus wie im nächsten Listing.

Listing: Bisherige VT100/ANSI-Escape-Codes

```
Console.WriteLine("This is a regular text");
Console.WriteLine("\u001b[1mThis is a bold text\u001b[0m");
Console.WriteLine("\u001b[2mThis is a dimmed text\u001b[0m");
Console.WriteLine("\u001b[3mThis is an italic text\u001b[0m");
Console.WriteLine("\u001b[4mThis is an underlined text\u001b[0m");
Console.WriteLine("\u001b[5mThis is a blinking text\u001b[0m");
Console.WriteLine("\u001b[6mThis is a fast blinking text\u001b[0m");
Console.WriteLine("\u001b[7mThis is an inverted text\u001b[0m");
Console.WriteLine("\u001b[8mThis is a hidden text\u001b[0m");
Console.WriteLine("\u001b[9mThis is a crossed-out text\u001b[0m");
Console.WriteLine("\u001b[21mThis is a double-underlined text\u001b[0m");
Console.WriteLine("\u001b[38;2;255;0;0mThis is a red text\u001b[0m");
Console.WriteLine("\u001b[48;2;255;0;0mThis is a red background\u001b[0m");
Console.WriteLine("\u001b[38;2;0;0;255;48;2;255;255;0mThis is a blue text with a
yellow background\u001b[0m");
```

Seit C# 13.0 gibt es nun \e als Kurzform für das ESCAPE-Zeichen ASCII 27 ein, sodass die Zeichenfolgen deutlich kompakter und übersichtlicher werden (siehe nächstes Listings).

Listing: Etwas übersichtlichere VT100/ANSI-Escape-Codes mit der neuen Abkürzung \e in C# 13.0

```
Console.WriteLine("This is a regular text");
Console.WriteLine("\e[1mThis is a bold text\e[0m");
Console.WriteLine("\e[2mThis is a dimmed text\e[0m");
Console.WriteLine("\e[3mThis is an italic text\e[0m");
Console.WriteLine("\e[4mThis is an underlined text\e[0m");
```

```
Console.WriteLine("\e[5mThis is a blinking text\e[0m");
Console.WriteLine("\e[6mThis is a fast blinking text\e[0m");
Console.WriteLine("\e[7mThis is an inverted text\e[0m");
Console.WriteLine("\e[8mThis is a hidden text\e[0m");
Console.WriteLine("\e[9mThis is a crossed-out text\e[0m");
Console.WriteLine("\e[21mThis is a double-underlined text\e[0m");
Console.WriteLine("\e[38;2;255;0;0mThis is a red text\e[0m");
Console.WriteLine("\e[48;2;255;0;0mThis is a red background\e[0m");
Console.WriteLine("\e[38;2;0;0;255;48;2;255;255;0mThis is a blue text with a yell
ow background\e[0m");
```

Die Abbildung zeigt das Ergebnis, das sowohl beide Listings produziert.

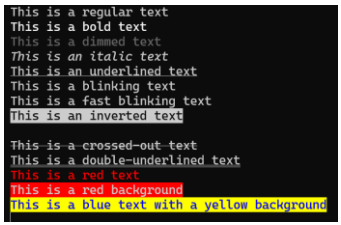


Abbildung: Die Ausgabe der beiden vorherigen Listings sieht gleich aus.

So gibt man ein Farbraster mit den neuen Escape-Codes aus (das war mit den alten Escape-Codes natürlich auch schon möglich, es ist jetzt nur prägnanter):

```
Console.WriteLine("\n\nFarbraster:");
for (int i = 0; i < 16; i++)
{
 for (int j = 0; j < 16; j++)
 {
  Console.Write("\e[48;5;" + (i * 16 + j) + "m" + (i * 16 + j).ToString().PadLeft
(4));
 }
 Console.WriteLine("\e[0m");
}
```

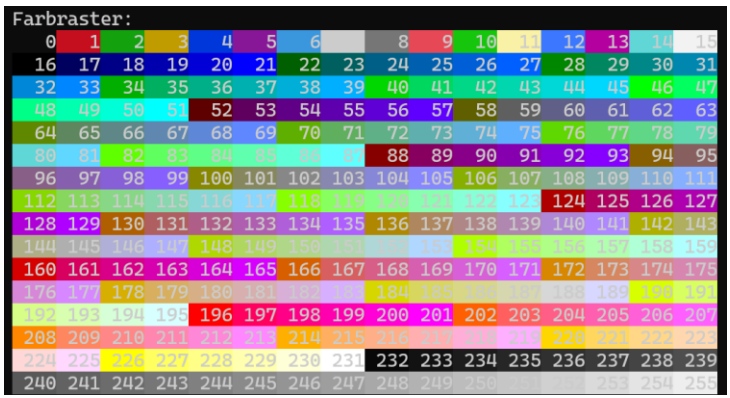


Abbildung: Farbraster in der Konsole mit ANSI-Codes

10.6 String Interpolation

Mit einer String Interpolation können Entwickler seit C# 6.0 die Zusammensetzung von Zeichenketten aus festen und variablen Bestandteilen übersichtlicher als bisher realisieren.

```
var ausgabeAlt1 = "Kunde #" + String.Format("{0:0000}", k.ID) +": " +
k.GanzerName + " ist in der Liste seit " + String.Format("{0:d}", k.ErzeugtAm) +
".";
var ausgabeAlt2 = String.Format("Kunde #{0:0000}: {1} ist in der Liste seit
{2:d}.", k.ID, k.GanzerName, k.ErzeugtAm);
var ausgabeNeu = $"Kunde #{k.ID:0000}: {k.GanzerName} ist in der Liste seit
{k.ErzeugtAm:d}.";
Console.WriteLine(ausgabeAlt1);
Console.WriteLine(ausgabeAlt2);
Console.WriteLine(ausgabeNeu);
```

```
Kunde #0123: Holger Schwichtenberg ist in der Liste seit 22.03.2018.
Kunde #0123: Holger Schwichtenberg ist in der Liste seit 22.03.2018.
Kunde #0123: Holger Schwichtenberg ist in der Liste seit 22.03.2018.
```

Abbildung: Ausgabe des obigen Beispiels

Der Einsatz des ternären Operators und Verschachtelungen von Interpolationsausdrücken sind möglich:

```
var ausgabeVerschachtelt = $"Kunde #{k.ID:0000}: {k.GanzerName} {(k.ErzeugtAm !=
null ? $"ist in der Liste seit {k.ErzeugtAm:d}" : "ist nicht in der Liste")}.";
```

Schon seit C# 6.0 lassen sich die einer Zeichenkette vorangestellten Operatoren $ und @ kombinieren, aber zunächst nur in der Reihenfolge

```
$@"{ID}: {Name} \\server\User{ID:000}";
```

Erst mit C# 8.0 wurde eingeführt, dass auch die andere Reihenfolge erlaubt ist:

```
@$"{ID}: {Name} \\server\User{ID:000}";
```

Erst seit C# 10.0 ist String Interpolation bei der Wertzuweisung an Konstanten möglich. Voraussetzung ist allerdings, dass die verwendeten Platzhalter auch alle mit Konstanten befüllt werden.

```
const string Vorname = "Holger";
const string Nachname = "Schwichtenberg";
// Constant Interpolated String
const string GanzerName = $"Dr. {Vorname} {Nachname}";
```

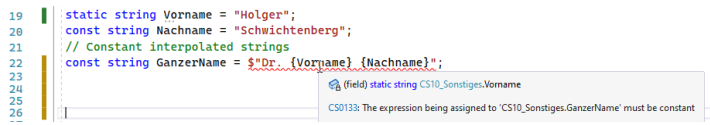


Abbildung: Der Compiler beschwert sich, dass im Interpolated String "GanzerName" eine Variable statt einer Konstanten verwendet wird.

Die String Interpolation ist seit C# 10 – sowohl mit Konstanten als auch Variablen – deutlich schneller als C# 6.0 bis 9.0, da Microsoft die Umsetzung durch den Compiler überarbeitet hat. Während vor C# 10.0 die Zeichenketten mit String.Format() und String.Concat() verbunden wurden, arbeitet im Untergrund nun eine Instanz der Klasse InterpolatedStringHandler, eine Variante der Klasse StringBuilder.

Aus dieser String Interpolation

```
string companyInfo = $"Company {ID:0000} {CompanyName} Postcode {Postcode:00000}
Founded {Founded:yyyy}";
```

macht der Compiler:

```
DefaultInterpolatedStringHandler defaultInterpolatedStringHandler = new DefaultInterpolatedStringHandler(28, 4);
defaultInterpolatedStringHandler.AppendLiteral("Company ");
defaultInterpolatedStringHandler.AppendFormatted(ID, "0000");
defaultInterpolatedStringHandler.AppendLiteral(" ");
defaultInterpolatedStringHandler.AppendFormatted(CompanyName);
defaultInterpolatedStringHandler.AppendLiteral(" Postcode ");
defaultInterpolatedStringHandler.AppendFormatted(Postcode, "00000");
defaultInterpolatedStringHandler.AppendLiteral(" Founded ");
defaultInterpolatedStringHandler.AppendFormatted(Founded, "yyyy");
string companyInfo = defaultInterpolatedStringHandler.ToStringAndClear();
```

Microsoft kommt zu diesen Performance-Ergebnissen, wobei "Old" C# 9.0 meint und "New" C# 10.0 [*devblogs.microsoft.com/dotnet/string-interpolation-in-c-10-and-net-6*]).

Method	Mean	Ratio	Allocated
Old	111.70 ns	1.00	192 B
New	66.75 ns	0.60	40 B

Mit dem folgenden Programmcode können Sie selbst nachmessen.

Listing: Vergleich von String Interpolation mit String.Format() und String.Concat()

```
using System.Diagnostics;

namespace CS10
{
 public class CS10_InterpolatedStringPerformance
```

```
{

  int loopCount = 1000000;

  public int ID { get; set; } = 123;
  public string CompanyName { get; set; } = "www.IT-Visions.de";
  public int Postcode { get; set; } = 45257;
  public DateTime Founded { get; set; } = new DateTime(1996, 1, 1);

  public void Run()
  {

   var sw1 = new Stopwatch();
   sw1.Start();
   for (int i = 0; i < loopCount; i++)
   {
    string name = "Company " + String.Format("{0:0000}", ID) + " " + CompanyName
+ " Postcode " + String.Format("{0:00000}", Postcode) + " Founded " +
String.Format("{0:yyyy}", Founded);
   }
   sw1.Stop();
   Console.WriteLine($"{loopCount} String Concat+String Format:
{sw1.ElapsedMilliseconds}ms");

   var sw2 = new Stopwatch();
   sw2.Start();
   for (int i = 0; i < loopCount; i++)
   {
    string name = String.Format("Company {0:0000} {1} Postcode {0:00000} Founded
{0:yyyy}", ID, CompanyName, Postcode, Founded);
   }
   sw2.Stop();
   Console.WriteLine($"{loopCount} String Format: {sw2.ElapsedMilliseconds}ms");

   var sw3 = new Stopwatch();
   sw3.Start();
   for (int i = 0; i < loopCount; i++)
   {
    string name = $"Company {ID:0000} {CompanyName} Postcode {Postcode:00000}
Founded {Founded:yyyy}";

   }
   sw3.Stop();
   Console.WriteLine($"{loopCount} String Interpolation:
{sw3.ElapsedMilliseconds}ms");
  }
 }
}
```

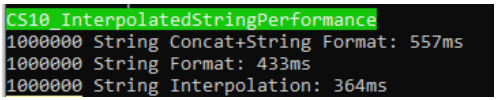


Abbildung: Ergebnisse des obigen Listings

Neu in C# 11.0 ist, dass Entwickler Zeilenumbrüche und Kommentare innerhalb von Zeichenketten-Interpolationsausdrücken (also innerhalb der geschweiften Klammern) erfassen können:

```
string ganzerName = "Dr. Holger Schwichtenberg";
var t = $"Vorname: {ganzerName // Aufteilen
                    .Split(" ")  // dann erstes Element
                    .ElementAt(1) }";
Console.WriteLine(t);
```

10.7 Raw Literal Strings (seit C# 11.0)

Seit C# 11.0 gibt es eine neue Syntaxform für Zeichenkettenliterale mit Umbrüchen. Bei einem "Raw Literal String" beginnt die Zeichenkette mit drei oder mehr Anführungszeichen (z.B. """) und endet mit der gleichen Anzahl von Anführungszeichen.

Die Motivation für dieses neue Sprachfeature war, eine Zeichenkettenrepräsentation zu verschaffen, in der keine Steuerzeichen (Escape-Sequenzen) notwendig werden, mit der sich aber dennoch einfach Umbrüche abbilden lassen und die Interpolation unterstützt.

In Raw Literal Strings gilt:

- Umbrüche landen in der Zeichenkette.
- Es gibt keine Steuerzeichen.
- Einrückungen bleiben erhalten, aber in jeder Zeile entfallen genau so viele Einrückungen wie es Einrückungen in der letzten Zeile vor dem Ende (z.B. """) gibt.
- Interpolationsausdrücke sind möglich mit zwei oder mehr Dollarzeichen vor den Anführungszeichen. Es sind dann in der Zeichenkette für den Interpolationsausdruck genauso viele geschweifte Klammern zu verwenden.

Ein erstes Beispiel mit einen Raw Literal String (hier ohne Einrückungen und ohne Interpolation) zeigt dieses Codefragment:

```
// Raw Literal String: 3 oder mehr Anführungszeichen zu Beginn
var rawLiteralString = """
.NET 7.0
ist am 8. November 2022 erschienen
mit Support für 18 Monate
""";
```

Die bisherigen Syntaxformen "Regular String" (Umbrüche mit \n) und "Verbatim String" (Beginn mit @) bleiben aber weiterhin erlaubt:

```
var regularString = "\n.NET 7.0\nist am 8. November 2022
erschienen\nmit Support für 18 Monate.\n";

var verbatimString = @"
.NET 7.0
ist am 8. November 2022 erschienen
```

```
mit Support für 18 Monate
";
```

Visual Studio 2022 seit Unterversion 17.2 bietet Refactoring-Funktionen, um zwischen den nun drei Zeichenkettenformen (Regular String, Verbatim String, Raw Literal String) umzuwandeln (siehe Abbildung).

Abbildung: Refactoring für Zeichenketten in Visual Studio 2022 seit Version 17.2

Die folgenden Beispiele zeigen Raw Literal String mit Einrückung und Interpolation:

```
  var name = "Dr. Holger Schwichtenberg";
  var website = "www.dotnet-doktor.de";

  var nameUndWebsite1 = $$"""
   Name: {{{name}}} Website: {{{website}}}
   """;
  Console.WriteLine(nameUndWebsite1); // Name: {Dr. Holger Schwichtenberg} Websit
e: {www.dotnet-doktor.de}


  var nameUndWebsite2 = $$$"""
   Name: {{{name}}} Website: {{{website}}}
  """;
  Console.WriteLine(nameUndWebsite2); // Name: Dr. Holger Schwichtenberg Website:
 www.dotnet-doktor.de
```

Praxisbeispiel

Das folgende Listing zeigt den Einsatz eines Raw Literal String für die Konstruktion einer JSON-Zeichenkette mit Einrückungen und Interpolation inklusive Kommentare in der Interpolation.

Listing: Ein Raw Literal String mit Interpolation, der JSON erzeugt

```
var name = "Dr. Holger Schwichtenberg";
var website = "www.dotnet-doktor.de";

var json = $$"""
           {
            "Person": {
              "Name": "{{name // Name der Person
                 }}",
              "Webseite": "{{website // Website in Kleinbuchstaben
                           .ToLower()}}"
            }
           }
           """;
```

Der Debugger Visualizer in Visual Studio zeigt bereits an, dass die Einrückungen per Leerzeichen erhalten bleiben (siehe nächste Abbildung).

Abbildung: Raw Literal String im Debugger-Visualizer

Die Ausgabe an der Konsole sieht daher so aus:

Abbildung: Ausgabe des Raw Literal String an der Konsole

Hinweis zu den Einrückungen in Raw Literal Strings

Die Einrückungen bei einem Raw Literal können gemischt aus Tabulatoren und Leerzeichen bestehen, z.B.

Abbildung: Gültige Mischung von Tabulatoren und Leerzeichen in einem Raw Literal String

Allerdings muss die Einrückung links von der Linie einheitlich in allen Zeilen sein. Die folgende Bildschirmabbildung zeigt eine ungültige Einrückung, weil in Zeile 51 und 53 jeweils zweimal einen Tabulator verwendet wird, in allen anderen Zeilen nur einmal. Visual Studio zeigt daher auch nicht die Linie an, die die Einrückung im Code von der Einrückung im String trennt.

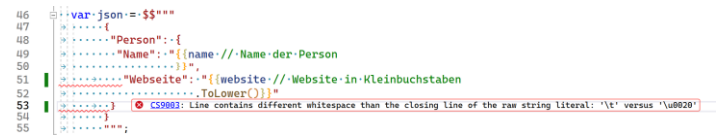


Abbildung 5: Ungültige Mischung von Tabulatoren und Leerzeichen in einem Raw Literal String

10.8 UTF-8-Zeichenkettenliterale (seit C# 11.0)

.NET arbeitet im Standard mit Zeichenketten in der Zeichencodierung UTF-16. In Webanwendungen wird in der Regel UTF-8 verwendet.

UTF-16 is the only web-encoding incompatible with ASCII and never gained popularity on the web, where it is declared by under 0.002% of web pages[(and many of these are actually UTF-8 because of "contradictory character encoding specifications" and/or "incorrect character encoding defined"). UTF-8, by comparison, accounts for 98% of all web pages. The Web Hypertext Application Technology Working Group (WHATWG) considers UTF-8 "the mandatory encoding for all [text]" and that for security reasons browser applications should not use UTF-16." [https://en.wikipedia.org/wiki/UTF-16]

Neu in C# 11.0 sind auch UTF-8-Zeichenkettenliterale mit denen Entwickler eine Zeichenkette angeben dürfen, aus der man eine Bytefolge von UTF-8-Codes in Form einer Instanz des .NET-Typs `ReadOnlySpan<byte>` erhält. Eine UTF-8-Zeichenkette benötigt den Nachsatz `u8` oder `U8` nach dem schließenden Anführungszeichen.

Die folgenden Beispiele zeigen "Hallo Holger!" in UTF-8-Zeichenkettenliteralen:

```
ReadOnlySpan<byte> s1 = "Hallo Holger!"u8;
var s2 = "Hallo Holger!"u8;
var s3 = "Hallo Holger!"U8;
byte[] s4 = "Hallo Holger!"u8.ToArray();
```

Alle diese Syntaxvarianten erzeugen in C# 11.0 die folgende Bytefolge:

```
0x48 0x61 0x6C 0x6C 0x6F 0x20 0x48 0x6F 0x6C 0x67 0x65 0x72 0x21
```

UTF8-Zeichenketten können jedoch nicht verwendet werden mit String Interpolation und in Standardwerten für Parameter!

10.9 Zahlenliterale

Für die gebrochenen Zahlen gibt es in C# besondere Kürzel, die in Literalen zu verwenden sind. Im Standard ist eine gebrochene Zahl vom Typ double. Der Suffix d ist also optional.

```
byte z1 = 123;
short z2 = 123;
int z3 = 123;
long z4 = 123;
float z5 = 123.45f;
double z6 = 123.45d;
decimal z7 = 123.45m;
```

Zahlenliterale kann der Entwickler seit C# 7.0 auch in Binärform hinterlegen. Der Unterstrich ist als Trennzeichen zur übersichtlicheren Darstellung bei Binär- und Dezimalsystemliteralen erlaubt und hat keinen Einfluss auf den Wert.

```
int AntwortAufAlleFragen = 0b001_01010; // 42
Console.WriteLine(AntwortAufAlleFragen);

decimal JahresGehalt = 123_456_789m;
Console.WriteLine(JahresGehalt);
```

10.10 Datumsliterale

Es gibt – anders als in Visual Basic .NET – keine eigene Syntax für Datumsliterale. Man kann ein Datum nur unter Verwendung des Konstruktors .NET-Klasse DateTime erzeugen.

```
   DateTime d1 = new DateTime(2018, 03, 23); // 23.3.2018 00:00:00 Uhr
   DateTime d2 = new DateTime(2018, 11, 11, 11,11,11); // 11.11.2018 11:11:11 Uhr
```

10.11 Lokale Typableitung (Local Variable Type Inference)

In C# 3.0 wurde die Typableitung neu eingeführt. Typableitung bedeutet, dass der Entwickler in seinem Programmcode keinen expliziten Typ vergibt, sondern der Compiler den Typ während der Übersetzung festlegt. Typableitung darf nicht mit Variant aus Visual Basic 5.0 / 6.0 verwechselt werden (auch wenn in C# 2008 das Schlüsselwort var heißt): Bei einem Variant konnte man jederzeit im Programmablauf den Typ ändern. Ein Variant war eine sehr speicherfressende Datenstruktur. Variablen, die mit Typableitung erzeugt wurden, erhalten hingegen zur Übersetzungszeit einen festen Typ, der im Programmablauf nicht mehr geändert werden darf und nicht mehr Speicher als bei einer expliziten Deklaration verbrauchen.

Typableitungen werden in C# durch das neue Schlüsselwort var anstelle des Datentyps, aber mit Initialisierung festgelegt.

Listing: Drei Typableitungen in C#

```
// Local Variable Type Inference für String
var heimatflughafen = "Essen/Mülheim";
Console.WriteLine(heimatflughafen.GetType().FullName);

// Local Variable Type Inference für Int32
var anzahl = vorstandsmitglieder.Count;
Console.WriteLine(anzahl.GetType().FullName);

// Local Variable Type Inference für die Klasse Vorstandsmitglied
var vorstandschef = vorstandsmitglieder[0];
Console.WriteLine(vorstandschef.GetType().FullName);
```

Hinweis: Die Typableitung heißt lokal, weil sie nur für lokale Variablen in Methoden möglich ist. Ein Einsatz als Attribut einer Klasse bzw. Parameter oder Rückgabewert einer Methode ist ausgeschlossen. Eine Typableitung muss immer mit einer Wertinitialisierung verbunden sein, da sonst keine Typableitung möglich ist. null bzw. nothing ist nicht erlaubt, da hier keine Typableitung möglich ist.

Man kann die Typableitung auch für Laufvariablen in Schleifen verwenden.

Wichtig: Bei vielen Entwicklern herrscht zunächst Skepsis über den Sinn der lokalen Typableitungen. Tatsächlich machen Typableitungen für sich isoliert betrachtet nur einen begrenzten Sinn. Typableitungen sind jedoch absolut notwendig im Zusammenhang mit anonymen Typen und LINQ-Projektionen. In beiden Szenarien entstehen Klassen, deren Namen der Entwickler nicht kennen kann.

Man darf Typableitung nicht mit dem Einsatz der allgemeinen Klasse System.Object verwechseln. Eine mit System.Object (alias object oder Object) deklarierte Variable kann tatsächlich im Programmablauf verschiedenartigste Inhalte aufnehmen. Eine mit lokaler Typableitung deklarierte Variable hingegen hat einen festen, unveränderbaren Typ.

Tipp: Gerade bei der Klasseninstanziierung in C# kann man durch die Typableitung die überflüssige Doppelnennung des Klassennamens vermeiden, denn man schreibt nun statt

```
Vorstandsmitglied v1 = new Vorstandsmitglied();
```

kürzer:

```
var v2 = new Vorstandsmitglied();
```

Die neue Schreibweise hat keinen Nachteil!

```
Vorstandsmitglied v1 = new Vorstandsmitglied();
Vorstandsmitglied v2 = new Vorstandsmitglied();
```

Abbildung:Beim Betrachten mit dem Decompiler .NET Reflector sieht man, dass der Compiler beide Zeilen gleich übersetzt hat

10.12 Gültigkeit von Variablen

Eine innerhalb eines Anweisungsblocks { … } deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unterroutine.

```
public void Aktion()
{
 int a = 1;
 {
  int b = 2;
  Console.WriteLine($"{a}+{b}={a}{b}");
 }
 // geht nicht, denn b ist hier nicht mehr gültig
 // Console.WriteLine($"{a}+{b}={a}{b}");
}
```

10.13 Typprüfungen

Mit GetType() ermittelt man von einer Variablen den Typ in Form einer Instanz der .NET-Klasse System.Type. Dies kann man mit dem Typ einer anderen Variablen vergleichen oder dem statischen Ausdruck typeof(Typ). Solch ein Vergleich macht nur Sinn für Variablen des Typs *object* oder *dynamic*. Wenn eine Variable typisiert ist (auch bei Einsatz des Schlüsselwortes var), wird die Prüfung immer nur für diesen Typ erfolgreich sein, selbst wenn eine Konvertierung in einen anderen Typ möglich wäre (hier am Beispiel: "5" ist eine Zeichenkette, keine Zahl). Eine solche Typkonvertierung muss man explizit implementieren (siehe nächstes Unterkapitel).

```
// Dieser Wert wurde eingegeben
object eingabe = "Holger";

if (eingabe.GetType() == typeof(string)) { Console.WriteLine("Eingabe ist ein
Text"); } // wahr

 eingabe = 1;
```

```
   if (eingabe.GetType() == typeof(int)) { Console.WriteLine("Eingabe ist eine
Zahl"); } // wahr

   dynamic eingabe2 = "Holger";

   if (eingabe2.GetType() == typeof(string)) { Console.WriteLine("Eingabe ist ein
Text"); } // wahr

   eingabe2 = 1;
   if (eingabe2.GetType() == typeof(int)) { Console.WriteLine("Eingabe ist eine
Zahl"); } // wahr

   var name = "Holger Schwichtenberg";
   if (name.GetType() == typeof(string)) { Console.WriteLine("name ist ein
Text"); } // wahr

    name = "5";
   if (name.GetType() == typeof(int)) { Console.WriteLine("name ist eine Zahl");
} // falsch
```

10.14 Typkonvertierung

Typkonvertierung (engl. Type Cast) bezeichnet die Umwandlung von einem Datentyp in einen anderen, z.B. Umwandeln einer Zahl in eine Zeichenkette oder Extrahieren einer Zahl aus einer Zeichenkette.

In C# kommt immer eine sehr strenge Typprüfung zum Einsatz, wohingegen sie in Visual Basic .NET explizit (mit Option Strict) eingeschaltet werden muss. Für

```
int zahl = 1;
```

sind folgende Konstrukte nicht gültig:

```
// falsch: string text = zahl;
// falsch: string text = ((string) zahl);
// falsch: string text = zahl as string;
```

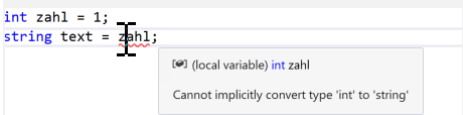


Abbildung: Der Compiler ist streng

Die Konvertierung von Zahl zu Text ist nur möglich über die ToString()-Methode oder über die .NET Basisklasse System.Convert.

```
string text1 = zahl.ToString();
string text2 = Convert.ToString(zahl);
```

Darüberhinaus bieten alle Klassen für Zahlen (System.Byte, System.Int16, System.Int32, etc.) sowie einige andere Typen wie System.Version und System.Guid die Möglichkeit, den Typ aus einer Zeichenkette zu extrahieren mithilfe der Methoden Parse() und TryParse().

```
decimal eingabezahlA;
if (System.Decimal.TryParse(eingabeA, out eingabezahlA))
   { Console.WriteLine("Eingabe ist die Zahl: " + eingabezahlA); }
else
```

```
   { Console.WriteLine("Eingabe war keine Zahl!"); }
```

Seit C# 7.0 kann man mit sogenannten "Inline-out-Variablen" die Syntax verkürzen:

```
string eingabeB = "123.45";
if (System.Decimal.TryParse(eingabeB, out decimal eingabezahlB))
   { Console.WriteLine("Eingabe ist die Zahl: " + eingabezahlB); }
else
   { Console.WriteLine("Eingabe war keine Zahl!"); }
```

Wenn es nur um die Typprüfung, aber nicht um die Konvertierung geht, dann kann man bei `out` die sogenannte Discard-Variable, die nur aus dem Unterstrich (_) besteht, einsetzen (ebenfalls seit C# 7.0).

```
string eingabeC = "123.45";
if (System.Decimal.TryParse(eingabeC, out _))
   { Console.WriteLine("Eingabe ist die Zahl!"); }
else
   { Console.WriteLine("Eingabe war keine Zahl!"); }
```

Zwischen polymorphen Klassen gibt es zwei Syntaxformen für die Typumwandlung:

1. Voranstellen des Zieltyps in runden Klammern

```
pass = ((Passagier)a[0]);
```

2. Verwendung des Operators `as`

```
pass = (a[0] as Passagier);
```

Der Unterschied zwischen der Schreibweise mit dem vorangestellten Typnamen und der Verwendung des as-Operators ist, dass in dem ersten Fall eine Ausnahme (InvalidCastException) erzeugt wird, wenn die Konvertierung nicht möglich ist, während der as-Operator in diesem Fall null zurückliefert.

Hinweis: Sie finden im Kapitel "Erweiterungsmethoden" Beispiele für einige sehr elegante Lösungen für die Typkonvertierung.

10.15 Dynamische Typisierung

Dynamische Typisierung bedeutet, dass die Einsprungstelle für einen Attributzugriff oder einen Methodenaufruf nicht zur Kompilierzeit feststeht (statische Typisierung), sondern erst zur Laufzeit ermittelt wird. Grundsätzlich ist statische Typisierung erstrebenswert, aber nicht immer ist dies möglich. Unmöglich ist die statische Typisierung zum Beispiel bei der Verwendung von COM-Bibliotheken, die als Datentypen Variant verwenden. Oder beim Zusammenspiel mit dynamischen Sprachen wie IronPython.

Achtung: Bei dynamischer Typisierung kann Visual Studio keine IntelliSense-Eingabeunterstützung bieten. Dynamische Typisierung birgt immer die Gefahr, dass die entsprechende Aktion nicht verfügbar ist, sei es durch einen Tippfehler oder weil ein anderes Objekt geliefert wird, als erwartet wurde. Wenn die Bindung nicht möglich ist, kommt es zum Laufzeitfehler (RuntimeBinderException).

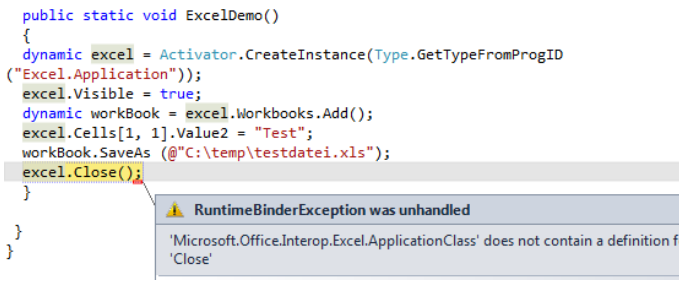


Abbildung: Laufzeitfehler, denn die Methode zum Schließen wäre Quit() statt Close() gewesen

In C# wurde die dynamische Typisierung erst in C# 4.0 auf einfache Weise ermöglicht. Vorher musste man sehr umständlich mit dem .NET-Reflection-Mechanismus arbeiten. C# bietet seit Version 4.0 dafür das Schlüsselwort dynamic.

Um dynamic in C# zu nutzen, muss man die Assembly Microsoft.CSharp.dll referenzieren. Es kommt sonst zum Fehler »Predefined type 'Microsoft.CSharp.RuntimeBinder.Binder' is not defined or imported.«

Listing: Verwendung der dynamischen Typisierung in C# [CS10_Dynamic.cs]

```
  /// <summary>
  /// Beispiel für dynamische Nutzung einer COM-
Bibliothek (hier: Microsoft Excel)
  /// </summary>
  public static void ExcelDemo()
  {
  dynamic excel = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Applicat
ion"));
  excel.Visible = true;
  dynamic workBook = excel.Workbooks.Add();
  excel.Cells[1, 1].Value2 = "Test";
  workBook.SaveAs (@"C:\temp\testdatei.xls");
  excel.Quit();
  }
```

10.16 Wertelose Wertetypen (Nullable Value Types)

Während Referenztypen bereits in .NET 1.x den Zustand null als Repräsentanz des Zustands nicht vorhanden / nicht gesetzt annehmen konnten, war dies für Wertetypen nicht vorgesehen. Ab .NET 2.0 existiert ein Hilfskonstrukt, um auch Wertetypen den Wert null zuweisen zu können.

In .NET (seit Version 2.0) ist ein auf null setzbarer Wertetyp eine generische Struktur (System.Nullable<T>), die aus dem eigentlichen Wert (Value) und einem Hilfs-Flag HasValue (Typ boolean) besteht, das anzeigt, ob der Wert des Typs null ist.

C# unterstützt Nullable Value Types bereits seit Version 2005 durch ein besonderes Sprachkonstrukt: Durch ein Fragezeichen als Suffix eines Wertetyps in einer Typdeklaration sorgt der C#-Compiler automatisch dafür, dass der Wertetyp in die generische System.Nullable-Struktur verpackt wird. Möglich ist auch eine explizite Deklaration mit System.Nullable.

```
// Wertetyp ohne null
int a = 1;
int b = 0;
// Wertetyp mit null erlaubt
int? x = 2;
System.Nullable<Int32> y = 6;
```

Die folgende Tabelle zeigt verschiedene Ergebnisse für Operationen mit den obigen Variablen.

Operation	Ergebnis, falls x den Wert 2 hat	Ergebnis, falls x null ist
string s1 = x.HasValue.ToString();	True	False
string s2 = x;	Kompilierungsfehler	Kompilierungsfehler
string s3 = x.Value.ToString();	2	Laufzeitfehler
string s4 = x.ToString();	2	Leere Zeichenkette
int? z = x + 10;	12	Null
int a1 = x;	2	Kompilierungsfehler
int a2 = (int)x;	2	Laufzeitfehler
int a3 = x ?? 0;	2	0

Tabelle: Verschiedene Operationen mit wertelosen Wertetypen in C# seit Version 2005

Bitte beachten Sie, dass man den Typ string (System.String) nicht als wertelosen Wertetyp verwenden kann, da String kein Wertetyp ist, sondern ein Referenztyp, der sich in einigen Punkten (z.B. Wertzuweisungen) verhält wie ein Wertetyp. Richtig ist also string i = null; statt string? i = null;

Listing: Verschiedene Beispiele mit Nullable Types

```
public void NullableTypes()
  {
   int a = 1;
   // Elegante Deklaration in C#
   int? b = 2;
   // a = null; // verboten!
   b = null; // Erlaubt
   // Explizite Deklaration
   System.Nullable<Int32> c = null;
   c = 100;
   Demo.Print(c.HasValue.ToString());
   Demo.Print(c.Value.ToString()); // Achtung: Geht nur, wenn c tatsächlich einen
Wert hat!
   // Besser: "Null" abfangen
   Demo.Print ("b = " + ( b.HasValue ? b.Value.ToString() : "null"));
  }
```

	C#	Visual Basic .NET
Deklaration eines normalen Wertetyps	int a;	Dim a As Integer
Zuweisung des nicht vorhandenen an einen normalen Wertetyp	Nicht möglich (Kompilierungsfehler)	a = nothing setzt den Wert auf die Zahl 0 bzw. anderen Startwert (z.B.z.B. DateTime.MinValue)
Deklaration eines wertelosen Wertetyps in Langform	System.Nullable<Int32> x = null	Dim x As System.Nullable(Of Integer) = Nothing
Deklaration eines wertelosen Wertetyps in Kurzform	int? x = null;	Integer? x = nothing;
Ausdruck x	Liefert Wert oder null	Visual Basic .NET 2005: Nicht möglich (Kompilierungsfehler) Ab Visual Basic .NET 2008: Liefert Wert oder null
Ausdruck x.Value	Liefert Wert oder Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«)	Liefert Wert oder Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«)
Ausdruck x.HasValue	Liefert true oder false	Liefert true oder false
Ausdruck x + 1	Liefert null, wenn x gleich null	Visual Basic .NET 2005: Nicht möglich (Kompilierungsfehler) Ab Visual Basic .NET 2008: Liefert null, wenn x gleich null
Zuweisung x = a	Erlaubt, liefert a	Erlaubt, liefert a
Zuweisung a = x	Kompilierungsfehler: Verbotene Typkonvertierung	Mit Option Strict: Verbotene Typkonvertierung Ohne Option Strict: Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn x gleich null
Zuweisung a = (int) x bzw. a = CType(x, Integer)	Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn x gleich null	Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn x gleich null

Konvertierung eines wertelosen Wertetyps in einen normalen Wertetypen mit der Semantik: liefert x, wenn x einen Wert hat oder Zahl 0, wenn x gleich null.	a = x ?? 0	If x.HasValue Then
		  a = x.Value
		Else
		  a = 0
		End If

Tabelle: Gegenüberstellung der Behandlung von wertelosen Wertetypen in C# und Visual Basic .NET

11 Operatoren

Es gibt einige wichtige Unterschiede zwischen den Operatoren in Visual Basic .NET und C#, die bei Portierungen von Code zu beachten sind:

- Das Gleichheitszeichen = ist in C# immer der Zuweisungsoperator. Zum Vergleichen müssen immer zwei Gleichheitszeichen = = verwendet werden.
- Das Ungleichheitszeichen ist != statt <>.
- Zeichenkettenverknüpfungen erfolgen immer mit dem Pluszeichen (+). Das kaufmännische Und (&) ist nicht erlaubt.
- Die logischen Operatoren Und (&&) und Oder (||) verwenden immer die Short-Circuit-Auswertung, d. h., die folgenden Teile eines Ausdrucks werden nicht mehr ausgewertet, sobald feststeht, dass der Ausdruck nicht mehr wahr werden kann. && entspricht also AndAlso und || also OrElse in Visual Basic .NET.
- Bei der Division ist es vom Typ der Operanden abhängig, ob die Division als Ganzzahldivision ausgeführt wird.

11.1 Überblick über die Operatoren

Die folgende Tabelle zeigt die Operatoren in C# im Vergleich zu anderen Programmiersprachen der .NET-Welt.

	Visual Basic	C#	Visual J#	C++	JScript
Mathematik					
Addition	+	+	+	+	+
Subtraktion	–	–	–	–	–
Multiplikation	*	*	*	*	*
Division	/	/	/	/	/
Ganzzahldivision	\	/			
Modulus	Mod	%	%	%	%
Potenz	^				
Negation	Not	~	~	~	~
Inkrement		++	++	++	++
Dekrement		- -	- -	- -	- -
Zuweisung					
Einfache Zuweisung	=	=	=	=	=
Addition	+ =	+ =	+ =	+ =	+=
Subtraktion	- =	- =	- =	- =	- =
Multiplikation	*=	*=	*=	*=	*=
Division	/=	/=	/=	/=	/=
Ganzzahl-Division	\=	/=			
Zeichenketten-verbindung	&=	+=	+=		+=
Modulus (Divisionsrest)		%=	%=	%=	%=
Bit-Verschiebung nach links	<< =	<< =	<< =	<< =	<< =
Bit-Verschiebung nach rechts	>> =	>> =	>> =	>> =	>> =
Bit-weises UND		&=	&=	&=	&=
Bit-weises XOR		^=	^=	^=	^=
Bit-weises OR		|=	|=	|=	|=
Vergleich					
Kleiner	<	<	<	<	<
Kleiner gleich	< =	< =	< =	< =	< =
Größer	>	>	>	>	>

	Visual Basic	C#	Visual J#	C++	JScript
Größer gleich	> =	> =	> =	> =	> =
Gleich	=	= =	= =	= =	= =
Nicht gleich	< >	!=	!=	!=	!=
Objektvergleich	Is	= =	= =		= =
Objektvergleich (negativ)	IsNot	!=	!=		!=
Objekttypvergleich	TypeOf x Is Class1	x is Class1	x instanceof Class1		Instanceof
Zeichenkettenvergleich	=	= =			= =
Zeichenkettenverbindung	&	+	+		+
Logische Operatoren					
UND	And	&&	&&	&&	&&
ODER	Or	||	||	||	||
NICHT	Not	!	!	!	!
Short-circuited UND	AndAlso	&&	&&	&&	&&
Short-circuited ODER	OrElse	||	||	||	||
Bit-Operatoren					
Bit-weises UND	And	&	&	&	&
Bit-weises XOR	Xor	^	^	^	^
Bit-weises OR	Or	|	|	|	|
Bit-Verschiebung nach links	<<	<<	<<	<<	<<
Bit-Verschiebung nach rechts	>>	>>	>>	>>	>>, >>>
Sonstiges					
Bedingt	IIF-Funktion und If-Operator	?:	?:	?:	?:
Bedingt (für Nullable Types)		?? :			

Tabelle: Vergleich der Operatoren in verschiedenen .NET-Sprachen

11.2 Überlaufprüfung

Standardmäßig ignoriert C# Überläufe in Ganzzahloperationen, was zu falschen Ergebnissen führen kann, ohne eine Ausnahme auszulösen.

Beispiel: Der maximale Wert eines int in C# ist 2.147.483.647. Wenn dieser Wert um 1 erhöht wird, führt dies zu einem Überlauf, wodurch der Wert negativ wird ("unterläuft"): -2.147.483.648.

```
int max = int.MaxValue;
int result = max + 1;  // Keine Exception, führt zu Überlauf und negativem Wert
Console.WriteLine(result);  // Ausgabe: -2147483648
```

Mit dem Einsatz von checked kann man diese Überläufe zur Laufzeit prüfen. Das Schlüsselwort checked kann man in einem einzelnen mathematischen Ausdruck wie eine Funktion verwenden oder als Blockoperator mit geschweiften Klammern wie z.B. using oder unsafe.

Listing: Einsatz von checked

```
try
{
 int result2 = checked(max + 1);
 Console.WriteLine(result2);
}
catch (Exception ex)
{
 CUI.Error(ex); // System.OverflowException: 'Arithmetic operation resulted in an
 overflow.'
}

try
{
 checked
 {
  int result3 = max + 1;
  Console.WriteLine(result3);
 }
}
catch (Exception ex)
{
 CUI.Error(ex); // System.OverflowException: 'Arithmetic operation resulted in an
 overflow.'
}
```

Zusätzlich gibt es auch das Gegenteil, das Schlüsselwort unchecked, das Überlaufüberprüfungen explizit ausschaltet in Fällen, in denen bereits der Compiler einen Überlauf erkennt, man dies aber zulassen will:

Listing: Einsatz von unchecked

```
unchecked
 {
  long e1 = long.MaxValue * 2;
  Console.WriteLine(e1); // erlaubter Überlauf -> -2

  ulong e2 = ulong.MaxValue * 2;
  Console.WriteLine(e2); // erlaubter Überlauf -> 18446744073709551614
 }

 long e3 = unchecked(long.MaxValue * 2);
```

```
 Console.WriteLine(e3); // erlaubter Überlauf -> 2

 ulong e4 = unchecked(ulong.MaxValue * 2);
 Console.WriteLine(e4); // erlaubter Überlauf -> 18446744073709551614
}
```

Hinweis: Das Ausschalten der Überlaufprüfung mit unchecked steigert die Performance, führt aber ggf. zu falschen Ergebnissen!

In C# unterstützen Fließkomma-Datentypen wie float, double und decimal weder checked noch unchecked. Das Verhalten ist aber bei diesen Typen verschieden:

- Bei einem Überlauf wirft der decimal-Typ immer eine OverflowException, unabhängig davon, ob checked oder unchecked verwendet wird.
- Für float und double ist das Verhalten bei Überläufen, Division durch Null und Ungenauigkeiten ist im IEEE 754-Standard festgelegt. Anstelle einer OverflowException geben diese Typen spezielle Werte wie Infinity, -Infinity und NaN (Not a Number) zurück.

11.3 Null Coalescing Operator ??

Ein C#-Operator, für den es keine Entsprechung in Visual Basic .NET gibt, ist das doppelte Fragezeichen (??). Der "Null Coalescing Operator" `??` liefert (seit C# 2.0) den Wert des vorangestellten Ausdrucks, wenn dieser nicht null ist. Wenn der Wert null ist, wird der Wert des nachfolgenden Ausdrucks übergeben. Somit kann man auf elegante Weise den null-Fall in einen anderen Wert umwandeln.

Listing: Einsatz des ??-Operators

```
// Umwandlung eines Nullable Int in einen Int
int? d = null;
int e = d ?? -1;
// Behandlung eines String
string s = null;
Demo.Print ("s = " + (s ?? "(kein Inhalt)"));
```

Leider ist der Operator nicht hilfreich, wenn man einen wertelosen Zahlenwert ausgeben möchte, weil beide Operanden den gleichen Typ besitzen müssen.

```
Demo.Print("d = " + (d ?? "null")); // geht leider nicht :-(
```

11.4 Null Coalescing Assignment ??=

Eine weitere Behandlung des null-Falls ist in C# 8.0 hinzugekommen in Form des Operators "Null Coalescing Assignment" mit `??=`. Mit diesem Zuweisungsoperator kann der C#-Softwareentwickler eine Zuweisung ausführen, wenn eine Variable den Wert null hat. Damit werden einige Einsatzgebiete des Null Assignment Operators nochmals verkürzt.

Statt

```
p = p ?? new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

oder

```
if (p == null) p = new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

kann man nun auch prägnanter schreiben:

```
p ??= new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

11.5 Null Conditional Operator ?.

Zu den sehr praktischen Neuerungen seit C# 6.0 gehört der Fragezeichen-Punkt-Operator (?.), der im Gegensatz zu dem einfachen Punkt-Operator keinen Laufzeitfehler auslöst, wenn der Ausdruck vor dem Punkt keinen Wert besitzt, also "null" (in C#) beziehungsweise "nothing" (in Visual Basic .NET) liefert. Microsoft nennt den Operator den Null Conditional Operator.

In der folgenden Zeile ist der Inhalt der Variablen name null, wenn entweder:

- Die Variable repository null ist
- Die Methode GetKontakt(123) null liefert
- Oder das String-Attribut Name im gelieferten Kontakt-Objekt null ist.

```
string name = repository?.GetKontakt(123)?.Name;
```

Hinweis: Auf den ersten Blick könnte man denken, dass hier die Ursache für einen Fehler nicht mehr erkennbar ist. In vielen Fällen geht es aber gar nicht darum, die Ursache für einen Fehler zu kennen, sondern primär erstmal darum, dass es gar keinen Fehler gibt. Hier hilft der Operator ?. sehr.

11.6 Operator nameof()

Der in C# 6.0 (und Visual Basic 14.0) neu eingeführte Operator nameof() liefert den Namen eines Bezeichners als Zeichenkette (bei mehrgliederigen Namen nur den letzten Teil). Dieser Operator erhöht die Robustheit und erleichtert das Refactoring in Situationen, in denen der Name einer Klasse oder eines Klassenmitglieds als Zeichenkette zu übergeben ist.

Listing: Einsatz des Operators nameof() für ArgumentNullException

```
public void SaveKontakt(Kontakt neuerKontakt)
  {
   if (neuerKontakt == null) throw new
ArgumentNullException(nameof(neuerKontakt));
   ...
  }
```

Listing: Einsatz des Operators nameof() für PropertyChangedEventArgs

```
public int KontaktAnzahl
  {
   get { return kontaktAnzahl; }
   set
   {
    PropertyChanged(this, new PropertyChangedEventArgs(nameof(KontaktAnzahl)));
    kontaktAnzahl = value;
   }
  }
```

Laut der Dokumentation ist der Operator nameof() auf Variablen, Typen und Mitglieder beschränkt.

nameof expression (C# reference)

07/12/2019 • 2 minutes to read •

A `nameof` expression produces the name of a variable, type, or member as the string constant:

Abbildung: learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof

Tatsächlich funktioniert nameof() aber auch mit Namensräumen.

Listing: Einsatzgebiete von nameof()

```
namespace CS60
{
 class CS60Demos
 {
  public int Property { get; set; }
  public static void DemoNameOf()
  {
   int Variable;
   Console.WriteLine("Namensraum: " + nameof(CS60));
   Console.WriteLine("Klasse: " + nameof(CS60Demos));
   Console.WriteLine("Methode: " + nameof(DemoNameOf));
   Console.WriteLine("Property: " + nameof(Property));
   Console.WriteLine("Variable: " + nameof(Variable));
  }
 }
}
```

Der Operator nameof() kann auch außerhalb der Klasse eingesetzt werden indem man den Klassennamen dem Mitgliedsnamen getrennt durch einen Punkt voranstellt:

```
nameof(Klasse.Klassenmitglied)
```

Hinweis: Dies ist aber nur für öffentliche Klassenmitglieder möglich [*github.com/dotnet/csharplang/issues/1990*]. Daher ist im nächsten Beispiel der Einsatz für "FieldPrivate" nicht möglich.

Listing: Einsatz des Operators nameof() für das Mapping eines Fields bei Entity Framework Core

```
public class DemoEntityClass
{
   public byte ID { get; set; }
    public int FieldPublic;
   private int FieldPrivate;
}


…
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
 modelBuilder.Entity<DemoEntityClass>().Property(nameof(DemoEntityClass.FieldPubl
ic));
 modelBuilder.Entity<DemoEntityClass>().Property("FieldPrivate");
}
```

11.6.1 Neuerungen für nameof() seit C# 11.0

Der Operator nameof() funktioniert seit C# 11.0 auch für Parameter von Methoden in Annotationen, die auf der Methode oder einen Parameter gesetzt sind.

```
[Description($"Diese Methode besitzt einen generischen Typparameter mit Namen {na
meof(T)} und erwartet eine Instanz dieses Types im Parameter {nameof(obj)}.")]
```

```
static void NameOfErweiterungen1<T>(T obj)
{

}

static void NameOfErweiterungen2<T>([Description($"Die Methode {nameof(NameOfErwe
iterungen2)} erwartet im ersten Parameter {nameof(obj)} ein Objekt vom Typ des ge
nerischen Parameters {nameof(T)}!")] T obj)
{

}
```

11.6.2 Neuerungen für nameof() seit C# 12.0

Der Operator nameof() funktionierte vor C# 12.0 in manchen Situationen nicht. Der Abruf des Namens von Instanzmitgliedern von Klassenmitglieder war nicht möglich in einigen Fällen (statische Mitglieder, Annotationen) vor C# 12.0. Microsoft hat den Einsatzbereich von C# in Version 12.0 auf diese Fälle erweitert.

Listing: nameof() funktioniert seit C# 12.0 auch in Annotationen und statischen Mitgliedern

```
[Description($"{nameof(StringLength)} liefert von {nameof(Name)} die Eigenschaft
{nameof(Name.Length)}")] // nameof(Name.Length) nicht möglich vor C# 12.0!
public struct Person
{
 public string Name;
 // bisher schon möglich:
 public string MemberName1() => nameof(Name);
 // bisher schon möglich:
 public string MemberName2() => nameof(Name.Length);
 // bisher schon möglich:
 public static string MemberName3() => nameof(Name);
 // bisher Fehler CS0120, da statisches Mitglieder versucht auf Mitglied von Mitg
lied zuzugreifen
 public static string MemberName4() => nameof(Name.Length);

 [Description($"{nameof(StringLength)} liefert von {nameof(Name)} die Eigenschaft
 {nameof(Name.Length)}")] // nameof(Name.Length) war nicht möglich vor C# 12.0!
 public int StringLength()
 {
  return Name.Length;
 }

 public void PrintMemberInfo()
 {
  Console.WriteLine($"Die Struktur {nameof(Person)} hat ein Mitglied {nameof(Name
)}, welches eine Eigenschaft {nameof(Name.Length)} besitzt!");
 }
}
```

Hier wäre vor C# 12.0 der Ausdruck nameof(Name.Length) in drei der sechs oben gezeigten Fälle nicht möglich gewesen und vom Compiler mit dem Kompilierungsfehler "error CS0120: An object reference is required for the non-static field, method, or property 'Name.Length'" quittiert worden.

Der folgende Screenshot zeigt mit roten Linien, was vor C# 12.0 nicht möglich war.

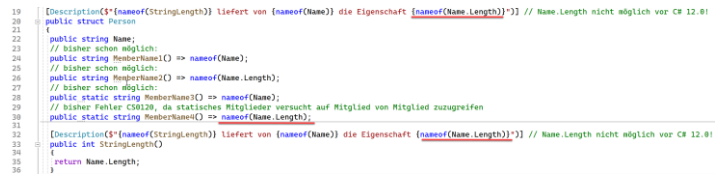


Abbildung: Unterstrichen sind vor C# 12.0 nicht mögliche Anwendungsfälle von nameof()

11.7 Index und Range (C# 8.0)

In C# 8.0 sind zwei neue Operatoren für die Auswahl von Teilmengen aus Mengen enthalten:

- Der Index-Operator (^), der eine Position relativ zum Ende einer Menge kennzeichnet. Der Compiler verwendet dafür die Klasse System.Index.
- Der Range-Operator (..), der einen Bereich mit Start und Ende aus einer Menge kennzeichnet. Der Compiler verwendet dafür die Klasse System.Range.

Hinweis: Range- und Index-Operator funktionieren nur in .NET Core seit Version 3.0, nicht aber im klassischen .NET Framework.

11.7.1 Index

Der Index-Operator (^) kennzeichnet eine Position relativ zum Ende einer Menge. Der Compiler verwendet dafür die Klasse System.Index.

```
string[] Namen = { "Leon", "Hannah", "Lukas", "Anna", "Leonie", "Marie",
"Niklas", "Sarah", "Jan", "Laura", "Julia", "Lisa", "Kevin" };
string i1 = Namen[^2]; // Index Operator: zweiter von hinten = "Lisa"  (^2 ==
Namen.Length-2)
string i2 = Namen[Namen.Length - 2]; // alte Schreibweise!

List<string> namensListen = Namen.ToList();
string l1 = namensListen[^2];

// ---- Andere Formulierungsweisen

Index i3 = ^2; // neue Klasse System.Index: zweiter von hinten
string n3 = Namen[i3]; // zweiter von hinten = "Lisa"

Index i4 = Index.FromEnd(2); // andere Schreibweise: zweiter von hinten
string n4 = Namen[i4]; // zweiter von hinten = "Lisa"
```

Achtung: Ein Zugriff auf Namen[^0] führt hingegen zum Laufzeitfehler "System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'", denn ^0 bedeutet Namen.Length-0, also Namen[Namen.Length], was ungültig ist, da die Zählung von 0 bis Namen.Length-1 läuft.

11.7.2 Range

Der Range-Operator (..) kennzeichnet einen Bereich mit Start und Ende aus einer Menge. Der Compiler verwendet dafür die Klasse System.Range.

Achtung: Bei Range ist der Start-Index inklusive (enthalten in der Zielmenge), aber der Ende-Index exklusiv (nicht enthalten in der Zielmenge). Die Zählung beginnt bei 0. Der Range 1..3 bedeutet also: das zweite und dritte Element. Das vierte Element der Menge ist NICHT dabei.

Dies empfinden einige Entwickler nicht als intuitiv. Microsoft hat sich nach einer Diskussion aber am 22.1.2018 bewusst so entschieden. Die Diskussion können Sie hier nachlesen: *https://github.com/dotnet/csharplang/blob/main/meetings/2018/LDM-2018-01-22.md*

Ranges können mit Indexen kombiniert werden.

```
// Ausschnitt .. von x bis vor!!! y (erstes ist INKLUSIV, zweites ist EXKLUSIV!)
string[] m1 = Namen[1..3]; // zweiter und dritter: "Hannah", "Lukas"
string[] m2 = Namen[6..^4]; // sechs von vorne und vier hinten abschneiden:
"Niklas", "Sarah", "Jan"
string[] m3 = Namen[11..]; // vom 12. Element bis Ende: "Lisa", "Kevin"
string[] m4 = Namen[0..^0]; // alle
string[] m5 = Namen[..]; // alle

// ---- Andere Formulierungsweisen
System.Range r1 = 1..3; // neue Klasse System.Range
string[] m6 = Namen[r1]; // zweiter und dritter: "Hannah", "Lukas"
```

Hinweise: Ranges waren ursprünglich schon für C# 7.3 geplant.

In der PowerShell gibt es schon seit Version 1.0 das Konzept der Ranges mit zwei Punkten als Operator. Der Ausdruck "0..20" generiert dabei die Menge aller Zahlen von 0 bis 20. Man kann Ranges in PowerShell auch Teilmengen adressieren, z.B. Menge[1..3] sind die Elemente 2, 3 und 4 der Menge. Bei der PowerShell ist also anders als in C# auch das Ende inklusive.

11.7.3 Weitere Beispiele

Das folgende Listing zeigt Beispiele für Range und Index mit einer Zeichenkette als Eingabemenge.

```
#region Ranges/Indexe mit Zeichenkette
string text = "0123456789";

//alt:
var teilstring1a = text.Substring(4, text.Length - 4); // --> ab dem 5. Zeichen
bis Ende "456789"
//neu
string teilstring1b = text[4..]; // --> ab dem 5. Zeichen bis Ende "456789"

string teilstring2 = text[^4..^0]; // 0 = Ende --> "6789"
string teilstring3 = text[2..4]; // "23"
string teilstring4 = text[0..^0]; // alle
string teilstring5 = text[..]; // .. == alle

// ---- Andere Formulierungsweisen
System.Range r7 = 1..3;// neue Klasse System.Range
string t7 = text[r7]; // "12"
#endregion
```

11.7.4 Einschränkungen

Während Indizes mit generischen Listen (Klassen, die IEnumerable<T> implementieren) funktionieren, ist dies mit Ranges nicht möglich.

```
List<string> namensListen = Namen.ToList();
string l1 = namensListen[^2];
Console.WriteLine(l1); // Lisa
//string l2 = namensListen[1..3]; // geht nicht ;-(
```

12 Schleifen

Sowohl Visual Basic .NET als auch C# unterstützen vier Typen von Schleifen:

- Kopfgeprüfte bedingte Schleifen *while (bedingung) { ... }*
- Fußgeprüfte bedingte Schleifen *do { ... } while (Bedingung)*
- Zählschleifen: Schleife mit einer bestimmten Anzahl von Durchläufen
  for ([Initialisierung];[Abbruchbedingung];[Iteration]) { ... }
- Mengenschleifen: Schleifen über alle Mitglieder eines Arrays oder eine anderer Objektmenge, welche die IEnumerable-Schnittstelle unterstützen (insbesondere die Klassen aus dem .NET-Basisklassen-Namensraum System.Collections): *foreach (x in y) { ... }*

Das Besondere an der for-Schleife ist, dass alle drei Bestandteile der runden Klammer optional sind. Das nachfolgende Beispiel enthält daher eine gültige for-Schleife, bei der Initialisierung, Abbruchbedingung und Iteration in eigenen Codezeilen enthalten sind. Eine innerhalb eines Anweisungsblocks einer Schleife deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unterroutine.

Normale For-Schleife	**For-Schleife ohne Inhalt in den runden Klammern**

for (int a = 0; a <= 10; a++)
{
…
}
	int b = 0;
	for (; ;)
	{
	b++;
	if (b > 10) break;
	…
	}

Tabelle: Beispiele für For-Schleifen in C#

Um eine aufzählbare Objektmengenklasse zu implementieren, leitet man diese von einer bestehenden aufzählbaren Klasse (aus dem Namensraum System.Collections) ab oder implementiert `IEnumerable` selbst unter Verwendung des Schlüsselworts `yield`, das mit C# 2.0 neu eingeführt wurde.

Listing: Beispiele für Schleifen

```
// 1. For-Schleife
   for (int a = 1; a <= 10; a++)
   {
    Console.WriteLine($"a={a}");
   }

// 2. Endlos-For-Schleife mit Abbruchbedingung
   int b = 0;
   for (;;)
   {
    b++;
    Console.WriteLine($"b={b}");
    if (b >= 10) break;
   }

// 3. while-Schleife
```

```
   int c = 0;
   while (c < 10)
   {
    c++;
    Console.WriteLine($"c={c}");
   }

// 4. do-while-Schleife
   int d = 0;
   do
   {
    d++;
    Console.WriteLine($"d={d}");
   } while (d < 10);

// 5. foreach-Schleife
   IEnumerable<int> zahlen = Enumerable.Range(1, 10);
   foreach (int e in zahlen)
   {
    Console.WriteLine($"e={e}"); ;
   }
```

13 Verzweigungen

Für die Verzweigung im Programmcode unterstützt C# die gleichen Konstrukte wie Visual Basic .NET: einfache Verzweigungen und Mehrfachverzweigungen.

- if (Bedingung) {…} else {…}
- switch (Bedingung) { case Wert:… default:… }

13.1 Einfache Verzweigungen mit if…else

Bei der if-Verzweigung und der if…else-Verzweigung sowie der if..else..else if-Verzweigung gelten folgende Regeln:

- Die Bedingungen müssen immer in runden Klammern stehen
- Die Befehlsblöcke müssen nur dann in geschweiften Klammern stehen, wenn mehr als eine Anweisung folgt. Wenn im Ausführungsblock nur eine Anweisung folgt, kann man die geschweifte Klammer weglassen. Die eine Anweisung kann direkt in derselben Zeile wie die Bedingung stehen oder eine Zeile danach. **Es ist Geschmackssache, ob man immer geschweifte Klammern setzen will. Viele Entwicklungsteam einigen sich hier auf Regeln im Team.**
- Eine innerhalb eines Anweisungsblocks { … } einer Bedingung deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unterroutine.

Listing: Fallunterscheidungen in C# mit if

```
   var note = 3; // Wert kommt irgendwo her

   // eine Ausführungszeile ohne Blockklammern ohne Umbruch
   if (note < 1 || note > 6) throw new ApplicationException("ungültige Note!");
   if (note <= 3) Console.WriteLine("akzeptable Leistung");
   else Console.WriteLine("zu schlecht");

   // eine Ausführungszeile ohne Blockklammern mit Umbruch
   if (note < 1 || note > 6)
    throw new ApplicationException("ungültige Note!");
   if (note <= 3)
    Console.WriteLine("akzeptable Leistung");
    // hier kann nicht noch eine Befehlszeile stehen , das bemängelt der
Compiler!: Console.WriteLine("Es geht aber noch besser!");
   else
    Console.WriteLine("zu schlecht");

   // mit Blockklammern mit beliebiger Zeilenanzahl und beliebigen Umbrüchen
   if (note < 1 || note > 6) { throw new ApplicationException("ungültige Note!");
}
   if (note <= 3)
    {
      Console.WriteLine("akzeptable Leistung");
      Console.WriteLine("Es geht aber noch besser!");
    }
   else { Console.WriteLine("zu schlecht"); }
```

13.2 Mehrfachverzweigungen mit switch

Bei der switch-Anweisung sind im Vergleich zu der Select-Anweisung in Visual Basic .NET folgende Punkte zu beachten:

- Jeder Fall muss mit einer break-Anweisung abgeschlossen werden
- Anders als in Visual Basic .NET kann man bei C# keine Wertebereiche nach case angeben

Listing: Fallunterscheidungen in C# mit switch

```
switch (note)
{
 case 1: e = "sehr gut"; break;
 case 2: e = "gut"; break;
 case 3: e = "befriedigend"; break;
 default: e = "zu schlecht"; break;
}
```

13.3 Switch Expressions (seit C# 8.0)

Die Mehrfachverzweigungen *switch { case: ... break; default: ... }* gibt es in C# seit der ersten Version. In C# 8.0 hat Microsoft eine deutlich prägnantere Variante dieses Sprachkonstrukts eingeführt.

Die neuen **Switch Expressions** sind so aufgebaut:

- Zuerst kommt der Wert (in der Regel in einer Variablen), anhand dessen unterschieden werden soll.
- Dann folgt das Schlüsselwort *switch*
- Dann folgt in geschweiften Klammern die Liste der Alternativen, jeweils gefolgt von einem Lambda-Pfeil => und dem resultierenden Wert. Break-Anweisungen sind dabei nicht notwendig.
- Anstelle des Schlüsselwortes default tritt die Discard-Variable (Unterstrich: _).

Hinweis: Da es sich bei einer Switch Expression dem Namen nach um einen Ausdruck handelt, muss ein Wert zurückgegeben und dieser verwertet werden, zum Beispiel für eine Zuweisung an eine Variable, als Rückgabewert oder Teil eines Ausdrucks.

Das folgende Beispiel zeigt eine Fallunterscheidung für eine textliche Aussage über einen Kunden anhand seiner Klassifizierung (A, B oder C). Die Fallunterscheidung wird zunächst mit dem klassischen Switch-Konstrukt realisiert, dann mit der in C# 8.0 neu eingeführten Switch Expressions.

Listing: Fallunterscheidung mit klassischen Switch-Konstrukt

```
string name = "Max Müller";
string status = "A";
string aussageUeberKunde = $"{name} ist ein ";

switch (status)
{
 case "A":
  aussageUeberKunde += "guter Kunde"; break;
 case "B":
  aussageUeberKunde += "durchschnittlicher Kunde"; break;
 case "C":
  aussageUeberKunde += "schlechter Kunde"; break;
```

```
 default:
  aussageUeberKunde += "sonstiger Kunde"; break;
}
Console.WriteLine(aussageUeberKunde);
```

Listing: Fallunterscheidung mit Switch-Expressions (seit C# 8.0)

```
string name2 = "Max Müller";
string status2 = "A";

var aussageUeberKunde2 = $"{name2} ist ein " + status2 switch
{
 // keine weiteren Statements hier erlaubt, z.B. string ausgabe = "{name} ist ein
";
 "A" => $"guter Kunde",
 "B" => $"durchschnittlicher Kunde",
 "C" => $"schlechter Kunde",
 _ => $"sonstiger Kunde"
};
Console.WriteLine(aussageUeberKunde2)
```

Es gibt auch die Option, dass eine ganze Methode nur aus einer Switch Expression bestehen kann. Dazu kombiniert man eine Switch Expression mit den Expression-bodied Members (seit C# 6.0 erlaubt. Hierzu gibt es ein eigenes Kapitel in diesem Buch.

Der erste Lambda-Ausdruck => für den Expression-bodied Member folgt nach der Parameterliste. Er legt die in der switch-Anweisung zu nutzende Variable fest. Danach folgt das Schlüsselwort switch. Die einzelnen Werte mit dem Folgeausdruck sind dann wieder jeweils durch den Lambda-Ausdruck => getrennt.

Die folgenden Listings zeigen drei Varianten des obigen Beispiels zur Kundenklassifizierung.

Listing: Klassisches Switch-Konstrukt in einer Methode

```
string GetKundenTypString_Classic(string name, string abc)
{
 //  weitere Statements hier erlaubt
 string kundenText = "";
 switch (abc)
 {
  case "A":
   kundenText = "guter Kunde"; break;
  case "B":
   kundenText = "durchschnittlicher Kunde"; break;
  case "C":
   kundenText = "schlechter Kunde"; break;
  default:
   kundenText = "sonstiger Kunde"; break;
 }
 return $"{name} ist ein {kundenText}.";
}
```

Seit C# 8.0 geht das prägnanter mit einer Switch Expression:

Listing: Switch Expression ohne Expression-bodied Member

```
string GetKundenTypString(string name, string abc)
{
 return abc switch
 {
```

```
  // keine weiteren Statements hier erlaubt, z.B. string ausgabe = "{name} ist ei
n";
  "A" => ${name} ist ein guter Kunde",
  "B" => ${name} ist ein durchschnittlicher Kunde",
  "C" => ${name} ist ein schlechter Kunde",
  _ => ${name} ist ein sonstiger Kunde"
 };
}
```

Noch prägnanter ist es, wenn man eine Switch Expression und Expression-bodied Member kombiniert:

Listing: Switch Expression und Expression-bodied Member

```
string GetKundenTypString2(string name, string abc) => abc switch
 {
  // keine weiteren Statements hier erlaubt, z.B. string ausgabe = "{name} ist ei
n";
  "A" => ${name} ist ein guter Kunde",
  "B" => ${name} ist ein durchschnittlicher Kunde",
  "C" => ${name} ist ein schlechter Kunde",
  _ => ${name} ist ein sonstiger Kunde"
 };
```

Der Wert, der zur Fallunterscheidung herangezogen wird, muss kein elementarer Datentyp sein. Das folgende Listing zeigt die Fallunterscheidung anhand eines Enumerationstyps.

Listing: Farbunterscheidung für einen in der klassischen Version

```
public ConsoleColor GetColor_Classic(LogLevel level)
  {
   switch (level)
   {
    case LogLevel.Information:
     return ConsoleColor.White;
    case LogLevel.Warning:
     return ConsoleColor.Yellow;
    case LogLevel.Error:
     return ConsoleColor.Red;
    default:
     throw new ArgumentException("Ungültiger Wert: " + level, nameof(level));
   };
  }
```

Listing: Farbunterscheidung für ein Property eines Objekts mit Switch Expression

```
public ConsoleColor GetColor(LogLevel level) => level switch
  {
   LogLevel.Information => ConsoleColor.White,
   LogLevel.Warning => ConsoleColor.Yellow,
   LogLevel.Error => ConsoleColor.Red,
   _ => throw new ArgumentException("Ungültiger Wert: " + level, nameof(level))
  };
```

Hinweis: Switch Expressions kann man auch verschachteln. Ein Beispiel dazu finden Sie im nächsten Kapitel "Pattern Matching".

13.4 Pattern Matching

Pattern Matching ist der Oberbegriff für eine Reihe von zusätzlichen Vergleichsoperationen, die Microsoft in C# 7.0 in die Sprache einbaut hat. Dieses Sprachfeature wurde von Microsoft in C# 9.0 und C# 10.0 sowie C# 11.0 weiter ausgebaut.

13.4.1 Pattern Matching in Bedingungen mit is und is not

Das Pattern Matching mit `is` wurde in C# 7.0 eingeführt. Seit C# 9.0 gibt es auch `is not`.

Mit `is` und `is not` sind Vergleiche nicht nur mit dem passenden Typ, sondern auch mit dem Basistyp `System.Object` möglich.

Beispiel:

Für diese Variable

```
object x = 42;
```

ist der folgende Vergleich nicht erlaubt, weil man ein Objekt vom Typ System.Object nicht mit einer Zahl vergleichen kann (Error CS0019 Operator '>' cannot be applied to operands of type 'object' and 'int'):

```
if (x > 0 && x <= 100) { Console.WriteLine($"x ist zwischen 0 und 100!"); }
```

Mit dem Pattern Matching-Operator ist der Vergleich jedoch möglich und funktioniert:

```
if (x is >= 0 and <= 100) { Console.WriteLine($"x ist zwischen 0 und 100!"); }
```

Weiteres Beispiel für:

```
object d = '1';
```

nicht erlaubt ist:

```
if ((d >= 'a' && d <= 'z') || (d >= 'A' && d <= 'Z')) { Console.WriteLine("Buchstabe!"); }
```

Möglich ist aber:

```
if (d is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z')) { Console.WriteLine("Buchstabe!"); }
if (d is not (>= 'a' and <= 'z') or (>= 'A' and <= 'Z')) { Console.WriteLine("Kein Buchstabe!"); }
```

In diesen beiden Pattern könnte man die inneren Klammern weglassen, da and wie üblich eine höhere Präzedenz als or besitzt:

```
if (d is (>= 'a' and <= 'z' or >= 'A' and <= 'Z')) { Console.WriteLine("Buchstabe!"); }
if (d is not (>= 'a' and <= 'z' or >= 'A' and <= 'Z')) { Console.WriteLine("Kein Buchstabe!"); }
```

Hinweis: Gewöhnungsbedürftig ist, dass man in Pattern Und mit and statt wie sonst in C# üblich mit && und Oder mit or statt wie sonst üblich || sowie Nicht mit not statt ! ausdrückt.

Typkonvertierungen von dem Typ System.Object in einen beliebigen anderen Typ können seit C# 7.0 in if- und switch-Bedingungen leichter realisiert werden mit Pattern Matching.

Mit dem Operator is kann der Entwickler ein Objekt in einer auf System.Object deklarierten Variablen auf einen beliebigen .NET-Typ prüfen und bei erfolgreicher Prüfung in eine zweite Variable, die nach dem Typ anzugeben ist, konvertieren lassen (siehe Listing). Anstelle der Bedingung *eingabe == null* kann der Entwickler in C# 7.0 auch *eingabe is null* schreiben.

Listing: Pattern Matching erlaubt Typprüfung und Typkonvertierung in einem Abwasch

```
// Beispiel: Ein Eingabewert, der aus einer Import-Datei kommt
object eingabe = 123;
// ...

if (eingabe is int zahl) { Console.WriteLine(zahl * 2); }
else { Console.WriteLine("Keine Zahl!"); }

if (eingabe is null) { Console.WriteLine("Leer"); }
```

Seit C# 9.0 ist auch ein Vergleich auf Ungleichheit mit *is not* möglich:

```
if (eingabe is not int zahl2) { Console.WriteLine("Keine Zahl!"); }
```

Die Routine PruefeEingabe() im folgenden Listing zeigt einige Beispiele für den Einsatz von is und is not.

Listing: Weitere Beispiele für Pattern Matching in Bedingungen

```
void PruefeEingabe(object eingabe)
{
 CUI.H3("Eingabe = " + eingabe);

 // Vergleiche mit null
 if (eingabe is null) { Console.WriteLine("Leer"); }
 if (eingabe is not null) { Console.WriteLine("Nicht leer"); }
 // Typprüfung
 if (eingabe is int zahl1) { Console.WriteLine("Zahl: " + zahl1); }
 if (eingabe is not int) { Console.WriteLine("Keine Zahl!"); }

 // Wertevergleiche mit is
 if (eingabe is >= 0 and <= 100) { Console.WriteLine($"Eingabe ist zwischen 0 und
 100!"); }
 if (eingabe is < 0 or > 100) { Console.WriteLine($"Eingabe ist nicht zwischen 0
und 100!"); }
 if (eingabe is 0 or 100) { Console.WriteLine($"Eingabe ist Extremwert!"); }
 if (eingabe is (>= 0 and <= 10) or (>= 90 and <= 100)) { Console.WriteLine($"Ein
gabe ist hoher oder niedriger Wert!"); }

 // Wertevergleiche mit is not
 if (eingabe is not (>= 0 and <= 100)) { Console.WriteLine("Eingabe ist nicht zwi
schen 0 und 100!"); }
 if (eingabe is not >= 0 or not <= 100) { Console.WriteLine("Eingabe ist nicht zw
ischen 0 und 100!"); }
}
```

13.4.2 Pattern Matching bei switch

Auch in Verbindung mit Schlüsselwort *switch* ist Pattern Matching möglich.

Listing: Pattern Matching bei switch mit Type Pattern

```
// Beispiel: Ein Eingabewert, der aus einer Import-Datei kommt
object eingabe = 123;
…
switch (eingabe)
   {
```

```
    case int z:
     Console.WriteLine("Das Doppelte ist: " + z * 2);
     break;
    case string s:
     Console.WriteLine(s);
     break;
    case bool b:
     if (b) Console.WriteLine("Die Aussage ist wahr!");
     break;
    case null:
     Console.WriteLine("Kein Wert");
     break;
    default:
     break;
   }
```

13.4.3 Pattern Matching für Typen

Seit C# 9.0 kann man auch Typvergleiche per Pattern Matching in Switch Expressions sehr prägnant definieren, wenn der konkrete Wert nicht interessiert. Microsoft nennt dies "Simplified Type Pattern".

Listing: Switch Expression mit Simplified Type Pattern

```
var ausgabe1 = eingabe switch
{
    int => "Eingabe ist eine Zahl!",
    string => "Eingabe ist eine Zeichenkette!",
    _ => "Eingabe ist etwas anderes"
};
```

13.4.4 Pattern Matching mit Größenvergleichen

Seit C# 9.0 sind auch Vergleiche mit den Operatoren >, >=, <= und < möglich. Microsoft nennt dies "Relational Pattern".

Listing: Switch Expression mit Relational Pattern

```
var ausgabe2 = eingabe switch
{
    < 0 => "Eingabe ist negative Zahl!",
    <= 100 => "Eingabe ist zwischen 0 und 100!",
    _ => "Eingabe ist größer als 100"
};
```

13.4.5 Pattern Matching mit logische Operatoren

Seit C# 9.0 sind auch logische Operatoren (und / oder) beim Pattern Matching möglich.

Aber Achtung: Abweichend von dem in C# sonst üblichen Standard drückt man diese nicht mit && und || aus, sondern mit den Wörtern `and` und `or`.

Listing: Switch Expression mit Relational Pattern und Logical Pattern

```
var ausgabe3 = eingabe switch
{
    < 0 => "Eingabe ist negative Zahl!",
    0 or 100 => "Alles oder nichts!",
```

```
    > 0 and < 100 => "Eingabe ist zwischen 0 und 100!",
    _ => "Eingabe ist größer als 100"
};
```

Praxisbeispiel

In dem folgenden Praxisbeispiel werden Type Pattern, Simplified Type Pattern, Relational Pattern und Logical Pattern kombiniert in einer verschachtelten Switch Expression.

Listing: Verschachtelte Switch Expression mit mehreren Pattern

```
public static void EingabeAuswerten()
  {
   // Beispiel: Ein Eingabewert, der aus einer Import-Datei kommt
   object eingabe = 98;

   string ausgabe = eingabe switch
   {
    int z => z switch
    {
     <0 => "Negative Zahl",
     0 => "Kein Ergebnis!",
     >= 1 and <=100 => "Zahl zwischen 1 und 100",
     _ => $"Sonstige Zahl: {z}"
    },
    DateTime => "Eingabe ist Datum!",
    _ => "Ungültige Eingabe!"
   };
   Console.WriteLine(ausgabe);
  }
```

13.4.6 Pattern Matching für Daten in einem Objekt (Property Pattern)

Bei einer Switch-Anweisung über ein komplexes Objekt kann man auch das sogenannte Property Pattern verwenden. Dabei wird bei den Fällen wieder das Objekt durch eine geschweifte Klammer repräsentiert und Bezug auf ein oder mehrere Properties genommen in der Form *{ Property1 : Wert1, Property2: Wert2, Property3: Wert3, usw }*.

Listing: Switch Expression mit Property Pattern über ein Property

```
string GetKundenTypStringFromKunde(Kunde k) => k switch
  {
   // keine weiteren Statements hier erlaubt, z.B. string ausgabe = "{name} ist
ein";
   { Status: 'A' } => $"{k.Name} ist ein gute Kunde",
   { Status: 'B' } => $"{k.Name} ist ein durchschnittlicher Kunde",
   _ => $"{k.Name} ist ein sonstiger Kunde"
  };
```

Listing: Switch Expression mit Property Pattern über zwei Properties

```
string GetKontaktTypString(Kontakt k) => k switch
  {
   // keine weiteren Statements hier erlaubt, z.B. string ausgabe = "{name} ist
ein";
   { Status: 'A', Art: KontaktArt.Kunde } => $"{k.Name} ist ein guter Kunde",
   { Status: 'A', Art: KontaktArt.Lieferant } => $"{k.Name} ist ein guter
Lieferant",
```

```
   { Status: 'B', Art: KontaktArt.Kunde } => $"{k.Name} ist ein
durchschnittlicher Kunde",
   { Status: 'B', Art: KontaktArt.Lieferant } => $"{k.Name} ist ein
durchschnittlicher Lieferant",
   _ => $"{k.Name} ist ein sonstiger Kontakt"
  };
```

Wenn man mehrere übergebene Parameter in die Fallunterscheidung einbeziehen will, kann man die Parameter in der Switch Expression zu einem Tupel zusammenfassen. Man spricht hier vom Tupel Pattern. Tupel gibt es in C# seit Version 7.0 (Hierüber gibt es ein eigenes Kapitel in diesem Buch). Dabei folgt nach dem Lambda-Pfeil in der Parameterliste die Erschaffung eines Tupels aus den gewünschten Parametern. Das Tupel wird dann in jeder Fallzeile der Switch Expression verwendet.

Listing: Switch Expression mit Tupel Pattern

```
public string GetAnrede(string Geschlecht, string Art)
    => (Geschlecht, Art) switch
    {
     ("w", "Kunde") => "Sehr geehrte Kundin",
     ("w", "Lieferant") => "Sehr geehrte Lieferantin",
     ("m", "Kunde") => "Sehr geehrter Kunde",
     ("m", "Lieferant") => "Sehr geehrter Lieferant",
     (_, _) => "Sehr geehrte Damen und Herren"
    };
```

Über das Property Pattern kann man auch Unterobjekte ansprechen. Wenn eine Klasse Person ein Unterobjekt Firma vom Typ Firma mit einem Property Firmennamen besitzt, kann man so prüfen, ob der Firmenname einen bestimmten Wert (hier: Leerstring) hat:

```
if (p is Person { Firma: { Firmenname: "" } })
  {
   Console.WriteLine("Firmenname fehlt!");
  }
```

Seit C# 10.0 geht das mit dem "Extended Property Pattern" auch eleganter mit der Punktnotation:

```
if (p is Person { Firma.Firmenname: "" })
  {
   Console.WriteLine("Firmenname fehlt!");
  }
```

13.4.7 Pattern Matching für Listen und Teilmengen (List Pattern und Slice Pattern)

Wie schon in den letzten C#-Versionen (seit Version 7.0) erweiterte Microsoft in C# 11.0 das Pattern Matching, dieses Mal um die Prüfung von Listen (List Pattern) und die Extraktion von Teilmengen (Slice Pattern).

Im Muster steht ein Unterstrich `_` für ein Element und der zweifache Punkt `..` für beliebig viele Elemente.

Die beiden Methoden CheckList() im folgenden Listing (in zwei Varianten mit Parameter vom Typ Integer-Array und List von Integer) prüfen, ob eine Zahlenmenge mit 1 und 2 oder nur mit 1 beginnt und liefert entsprechende Textaussagen zurück.

Listing: List Pattern

```
public string CheckList(int[] values)
=> values switch
```

```
{
 [1, 2, .., 10]
    => "Liste beginnt mit 1 und 2 sowie endet mit 10",
 [1, 2] => "Liste besteht aus 1 und 2",
 [1, _] => "Liste beginnt mit 1, es kommt danach noch genau ein Element",
 [1, ..] => "Liste beginnt mit 1, danach noch mehrere Elemente",
 [_] => "Liste aus einem Element, beginnt nicht mit 1",
 [..] => "Liste aus mehreren Elementen, beginnt nicht mit 1"
};

public string CheckList(List<int> values)
=> values switch
{
 [1, 2, .., 10]
    => "Liste beginnt mit 1 und 2 sowie endet mit 10",
 [1, 2] => "Liste besteht aus 1 und 2",
 [1, _] => "Liste beginnt mit 1, es kommt danach noch genau ein Element",
 [1, ..] => "Liste beginnt mit 1, danach noch mehrere Elemente",
 [_] => "Liste aus einem Element, beginnt nicht mit 1",
 [..] => "Liste aus mehreren Elementen, beginnt nicht mit 1"
};
```

Für die folgenden Beispielaufrufe bekommt der Aufrufer die jeweils als Kommentar dahinter genannten Rückgabewerte:

```
Console.WriteLine(CheckList(new[] { 1, 2, 10 }));          // "Liste beginnt mit
1 und 2 sowie endet mit 10"
Console.WriteLine(CheckList(new[] { 1, 2, 7, 3, 10 }));    // "Liste beginnt mit
1 und 2 sowie endet mit 10"
Console.WriteLine(CheckList(new[] { 1, 2 }));              // "Liste besteht aus
1 und 2"
Console.WriteLine(CheckList(new[] { 1, 3 }));              // "Liste beginnt mit
1, es kommt danach noch genau ein Element"
Console.WriteLine(CheckList(new[] { 1, 2, 5 }));           // "Liste beginnt mit
1, danach noch mehrere Elemente"
Console.WriteLine(CheckList(new[] { 3 }));                 // "Liste aus einem El
ement, beginnt nicht mit 1"
Console.WriteLine(CheckList(new[] { 3, 5, 6, 7 }));        // "Liste aus mehreren
 Elementen, beginnt nicht mit 1"
Console.WriteLine(CheckList(new[] { 3, 4 }));              // "Liste aus mehreren
 Elementen, beginnt nicht mit 1"

Console.WriteLine(CheckList(new List<int> { 1, 2, 10 }));          // "Liste begi
nnt mit 1 und 2 sowie endet mit 10"
Console.WriteLine(CheckList(new List<int> { 1, 2, 7, 3, 10 }));    // "Liste begi
nnt mit 1 und 2 sowie endet mit 10"
Console.WriteLine(CheckList(new List<int> { 1, 2 }));              // "Liste best
eht aus 1 und 2"
Console.WriteLine(CheckList(new List<int> { 1, 3 }));              // "Liste begi
nnt mit 1, es kommt danach noch genau ein Element"
Console.WriteLine(CheckList(new List<int> { 1, 2, 5 }));           // "Liste begi
nnt mit 1, danach noch mehrere Elemente"
Console.WriteLine(CheckList(new List<int> { 3 }));                 // "Liste aus
einem Element, beginnt nicht mit 1"
Console.WriteLine(CheckList(new List<int> { 3, 5, 6, 7 }));        // "Liste aus
mehreren Elementen, beginnt nicht mit 1"
```

```
Console.WriteLine(CheckList(new List<int> { 3, 4 }));              // "Liste aus
mehreren Elementen, beginnt nicht mit 1"
```

Man kann mit Variablennamen im Pattern auch einzelne Elemente einer Menge herausgreifen (Slice Pattern). ExtractValue() liefert eine Zeichenkette aus einer Menge von Zahlen:

Listing 8: Slice Pattern

```
/// <summary>
/// Slice Pattern
/// </summary>
public string ExtractValue(int[] values)
=> values switch
{
 [1, var middle, _] => $"Mittlere Zahl von 3 Zahlen (Beginn 1): {String.Join(", "
, middle)}",
 [_, var middle, _] => $"Mittlere Zahl von 3 Zahlen (Beginn beliebig): {String.Jo
in(", ", middle)}",
 [.. var all] => $"Alle Zahlen: {String.Join(", ", all)}"
};
```

Hier liefern die Aufrufe von `ExtractValue()` folgende Ergebnisse:

```
Console.WriteLine(ExtractValue(new[] { 1, 2, 6 }));        // "Mittlere Zahl von
3 Zahlen (Beginn 1): 2"
Console.WriteLine(ExtractValue(new[] { 3, 4, 5 }));        // "Mittlere Zahl von
3 Zahlen (Beginn beliebig): 4"
Console.WriteLine(ExtractValue(new[] { 2, 5, 6 }));        // "Mittlere Zahl von
3 Zahlen (Beginn beliebig): 5"
Console.WriteLine(ExtractValue(new[] { 1, 2, 5, 6 }));     // "Alle Zahlen: 1, 2,
 5, 6"
Console.WriteLine(ExtractValue(new[] { 2, 5, 6, 7 }));     // "Alle Zahlen: 2, 5,
 6, 7"
```

Durch Voranstellen von zwei Punkten vor der Variablen (`.. var middle`) entspricht die Teilmenge (Slice) mehreren Elementen. Hier eine Variante ExtractValues():

```
public string ExtractValues(int[] values)
     => values switch
     {
      [1, .. var middle, _] => $"Mittlere Zahlen: {String.Join(", ", middle)}",
      [.. var all] => $"Alle Zahlen: {String.Join(", ", all)}"
     };
}
```

Hier liefern die Aufrufe von `ExtractValues()` folgende Ergebnisse:

```
Console.WriteLine(ExtractValues(new[] { 1, 2, 5, 6 }));    // "Mittlere Zahlen (B
eginn 1): 2, 5"
Console.WriteLine(ExtractValues(new[] { 1, 2, 6 }));       // "Mittlere Zahlen (B
eginn 1): 2"
Console.WriteLine(ExtractValues(new[] { 2, 5, 6, 7 }));    // "Alle Zahlen: 2, 5,
 6, 7"
Console.WriteLine(ExtractValues(new[] { 2, 5, 6 }));       // "Alle Zahlen: 2, 5,
 6"
```

Hinweise: Das List-Pattern funktioniert mit allen Mengentypen, die eine Eigenschaft `Length` oder `Count` sowie einen Indexer (`name[x]`) besitzen. Beim Slice-Pattern muss der Indexer der Menge ein Range-Objekt als Eingabe unterstützen oder aber der Listentyp muss eine `Slice()`-Methode mit zwei Integer-Parametern besitzen. Diese Voraussetzungen sind für die auf der Schnittstelle `IEnumerable` basierenden Mengentypen noch nicht generell gegeben. Microsoft

ruft zum Feedback auf (siehe [*https://devblogs.microsoft.com/dotnet/early-peek-at-csharp-11-features/*]).

Die Struktur `Autor` im nächsten Listing bietet eine Methode ExtractTitleAndSurname() zur Extraktion von Namensbestandteilen. ExtractTitleAndSurname() liefert als Rückgabe eine Zeichenkette, die man per Split() bei den Leerzeichen auftrennt. Dann wird der Titel und der Nachname extrahiert. ToString() liefert Titel und Nachname als JSON-Zeichenkette.

Listing: Extraktion von Namensbestandteilen mit Slice Pattern

```
struct Autor : IAutor
{
 public required int ID;
 public string Name { get; set; }
 public Autor() { }

 private (string Titel, string Surname) ExtractTitleAndSurname(string fullname)
     => fullname.Split(" ") switch // Slice Pattern
     {
      ["Prof.", "Dr.", var nachname] => ("Professor Doktor", nachname),
      ["Dr.", var nachname] => ("Doktor", nachname),
      ["Prof.", var nachname] => ("Professor", nachname),
      ["Prof.", "Dr.", _, .. var all] => ("Professor Doktor", String.Join(" ", al
l)),
      ["Dr.", _, .. var all] => ("Doktor", String.Join(" ", all)),
      ["Prof.", _, .. var all] => ("Professor", String.Join(" ", all)),
      [_, var nachname] => ("", nachname),
      [var nachname] => ("", nachname),
      [_, .. var all] => ("", String.Join(" ", all)),
      _ => ("", "")
     };

 public override string ToString()
 {
  var json = $$"""
             {
              "Autor": {
               "ID": "{{ID}}",
               "Titel": "{{ExtractTitleAndSurname(Name)
                     .Titel}}",
               "Nachname": "{{ExtractTitleAndSurname(Name)
                        .Surname}}"
              }
             }
             """;
  return json;
 }
}
```

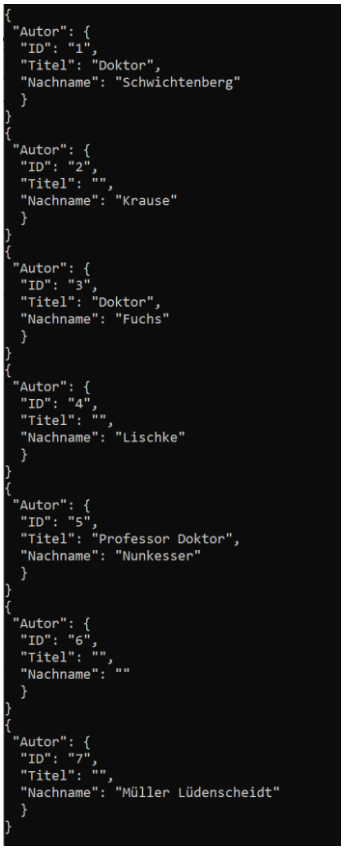
Hinweis: Es sind in dem Beispiel noch nicht alle möglichen Fälle abgedeckt. Es gibt in .NET und C# schon lange andere Optionen für solch eine Extraktion, z.B. reguläre Ausdrücke, die aber in so einem Fall unübersichtlicher sind.

Der Client im folgenden Listing zeigt, welche Fälle von ExtractTitleAndSurname() abgedeckt sind.

Listing: Nutzung des Slice Pattern aus dem vorherigen Listing

```
Autor hs = new() { ID = 1, Name = "Dr. Holger Schwichtenberg" };
Console.WriteLine(hs);

Autor mm = new() { ID = 2, Name = "Jörg Krause" };
Console.WriteLine(mm);

Autor jf = new() { ID = 3, Name = "Dr. Fuchs" };
Console.WriteLine(jf);

Autor ol = new() { ID = 4, Name = "Lischke" };
Console.WriteLine(ol);

Autor rn = new() { ID = 5, Name = "Prof. Dr. Robin Nunkesser" };
Console.WriteLine(rn);

Autor leer = new() { ID = 6, Name = "" };
Console.WriteLine(leer);

Autor mehrereNamen = new() { ID = 7, Name = "Max Müller Lüdenscheidt" };
Console.WriteLine(mehrereNamen);
```

Dies ist die zugehörige Ausgabe des Clients:



```
{
 "Autor": {
  "ID": "1",
  "Titel": "Doktor",
  "Nachname": "Schwichtenberg"
  }
}
{
 "Autor": {
  "ID": "2",
  "Titel": "",
  "Nachname": "Krause"
  }
}
{
 "Autor": {
  "ID": "3",
  "Titel": "Doktor",
  "Nachname": "Fuchs"
  }
}
{
 "Autor": {
  "ID": "4",
  "Titel": "",
  "Nachname": "Lischke"
  }
}
{
 "Autor": {
  "ID": "5",
  "Titel": "Professor Doktor",
  "Nachname": "Nunkesser"
  }
}
{
 "Autor": {
  "ID": "6",
  "Titel": "",
  "Nachname": ""
  }
}
{
 "Autor": {
  "ID": "7",
  "Titel": "",
  "Nachname": "Müller Lüdenscheidt"
  }
}
```


Abbildung: Ausgabe des obigen Listings

14 Klassendefinition

Klassen sind in .NET das zentrale Konzept zur Aufnahme von Daten und Programmcode. Eine Klassendefinition erstellt eine neue Klasse.

Klassen können folgende Elemente enthalten:

- Attribute in Form von Feldern oder Property-Routinen
- Methoden mit und ohne Rückgabewerte (Function/Sub)
- Ereignisse (Events)

Hinweis: Sowohl in C# als auch in Visual Basic .NET gilt: Anders als in Java darf eine Quellcodedatei beliebig viele Klassen enthalten und der Name der Quellcodedatei muss nicht dem in der Datei implementierten Klassennamen entsprechen. Die in Visual Studio integrierten Refactoring-Funktionen (Funktionen zur nachträglichen Umgestaltung von Programmcode) werden für C#-Klassen allerdings automatisch tätig, wenn eine Quellcodedatei umbenannt wird, die eine Klasse mit gleichem Namen enthält. In diesem Fall wird auch die Klasse umbenannt.

14.1 Klassendefinitionen

Klassen werden in C# durch das Schlüsselwort class und einen Block mit geschweiften Klammern gebildet.

Das Listing zeigt die Implementierung der Klasse Person mit zahlreichen Klassenmitgliedern, die in den folgenden Kapiteln näher erläutert werden.

Listing: Implementierung der Klasse Person in C#

```
#region Namensräume einbinden
using System;
using System.Collections.Generic;
using System.Text;
#endregion

namespace de.WWWings
{

 /// <summary>
 /// Basisklasse für Mitarbeiter und Passagiere
 /// </summary>
 [System.Serializable()]
 public class Person
 {
  #region Attribute (Fields)
  private long _ID;
  #endregion

  #region Attribute (Properties)
  public long ID
  {
   get { return _ID; }
   set { _ID = value; }
  }
```

```
public string Vorname { get; set; }
public string Nachname { get; set; }
public DateTime Geburtsdatum { get; set; }
#endregion

#region  Errechnete Attribute (Properties)

/// <summary>
/// Liefert Vorname und Nachname
/// </summary>
public string GanzerName
{
 get
 {
  return this.Vorname + " " + this.Nachname;
 }
}
#endregion

#region  Konstruktoren
// Parameterloser Konstruktor
public Person()
{
}
// Konstruktor, der an anderen Konstruktor delegiert
public Person(int id, string nachname, string vorname) : this(nachname,vorname)
{
 this.ID = id;
}

public Person(string Nachname, string Vorname)
{
 this.Vorname = Vorname;
 this.Nachname = Nachname;
}
#endregion

#region Methoden


/// <summary>
/// Überschreiben einer geerbten Methode
/// </summary>
public override string ToString()
{
 return "Person: " + this.GanzerName;
}

public virtual void Info()
{

 Console.WriteLine(this.ToString());
}
```

```
  #endregion
 }
}
```

14.2 Instanzierung mit dem Operator new

Eine Klasse wird mit dem Operator new instanziiert. Eine passende Objektvariable ist vorab zu deklarieren.

```
Person p;
…
p = new Person();
```

Wenn man eine Variablendeklaration und die Zuweisung in eine Zeile schreibt, ist im Standard der Klassenname zweimal zu verwenden:

```
Person p = new Person();
```

Mit dem Einsatz des Schlüsselwortes var (seit C# 3.0) bzw. dem Sprachfeature "Target-Typed New Expression" (seit C# 9.0) kann man dies verkürzen.

14.2.1 Angabe der Konstruktorparameter

In runden Klammern gibt der Nutzer der Klasse die Konstruktorparameter an.

```
Person p = new Person(123, "Schwichtenberg", "Holger");
```

Da es nur einen parameterlosen Konstruktor gibt, ist eine Instanziierung ohne Parameter mit new() möglich.

```
Person p = new Person();
```

Das folgende Listing zeigt Beispiele.

Listing: Verwendung des Operators new

```
// Person instanziieren mit parameterlosem Konstruktor (ohne
Konstruktorparameter)
Person p1 = new Person();
p1.Vorname = "Holger";
p1.Nachname = "Schwichtenberg";

Console.WriteLine(p1.GanzerName);
Console.WriteLine(p1.ToString());
Console.WriteLine(p1); // entspricht ToString()

// Person instanziieren mit Konstruktorparametern
Person p2 = new Person(123, "Schwichtenberg", "Holger");

Console.WriteLine(p2.GanzerName);
Console.WriteLine(p2.ToString());
Console.WriteLine(p2); // entspricht ToString()
```

14.2.2 Schlüsselwort var

Seit C# 3.0 gilt es durch die Verwendung des Schlüsselwortes var vor dem Variablennamen den Instanzierungsausdruck zu verkürzen:

```
var p2 = new Person(123, "Holger", "Schwichtenberg");
```

Wichtig: Das C#-Schlüsselwort var darf nicht mit dem Datentyp "Variant" in Visual Basic .NET verwechselt werden! "var" in C# ist kein eigener Datentyp, sondern bedeutet, dass der Compiler den Datentyp für die Variable aus dem Ergebnis der Zuweisung wählt. Für den

Compiler sind die Ausdrücke Person p1 = new Person(); und var p1 = new Person(); daher gleichbedeutend. Der Einsatz von var erspart dem Entwickler etwas Tipparbeit. Der Einsatz von var ist in vielen Entwicklungsteams umstritten.

14.2.3 Verwendung des Operators *new* ohne Typangabe (Target-Typed New Expression)

Seit C# 9.0 bietet Microsoft in der Sprachsyntax eine andere Verkürzung an, die das Potential hat, weniger umstritten zu sein. Man kann nun nach dem Operator new den Klassennamen weglassen, wenn man Deklaration und Initialisierung in eine Zeile schreibt und der Typ instanziiert wird, der durch die Deklaration vorgegeben wurde. Voraussetzung ist natürlich, man will die Klasse instanziieren, die der Deklaration entspricht und nicht etwa eine abgeleitete Klasse.

Der Entwickler kann also statt

```
Person p = new Person();
Person hs = new Person(123, "Holger", "Schwichtenberg");
```

oder

```
var p = new Person();
var hs = new Person(123, "Holger", "Schwichtenberg");
```

nun auch schreiben:

```
Person p = new();
Person hs = new(123, "Holger", "Schwichtenberg");
```

Auch mit generischen Typen ist dies möglich, also statt

```
List<Person> personList = new List<Person>();
```

oder

```
var personList = new List<Person>();
```

nun zu schreiben:

```
List<Person> personList = new();
```

Im Gegensatz zu var kann man Target-Typed New Expression auch in Klassenmitgliedern, z.B. bei der Initialisierung von Properties und Fields einsetzen:

```
class Person
{
   public int ID { get; init; }
   public string Firstname { get; set; }
   public string Surname { get; set; }
   public Adresse Adresse { get; set; } = new();
   …
}
```

Auch im Programmablauf kann man Datenmitglieder (Properties und Fields) seit C# 9.0 durch new() ohne Angabe des Klassennamens befüllen, da der Klassenname ja durch die Deklaration bereits feststeht:

```
Person p4 = new() { Vorname = "Holger", Nachname = "Schwichtenberg" };
p4.Adresse = new() { Ort = "Essen", Land = "DE" };
```

Selbst eine Übergabe als Methodenparameter ist ohne Klassennamen möglich, wenn sich dieser aus dem erwarteten Parameter ergibt:

```
public void Umziehen(Adresse adresse)
{
```

```
    this.Adresse = adresse;
}
…
p4.Umziehen(new() { Ort = "Essen", Land = "DE" });
```

Hinweis: Diese letzten hier gezeigten Anwendungsgebiete (Werte von Datenmitgliedern von außen setzen und Werte für Parameter) haben wieder das Potential zu Diskussionen in den Entwicklungsteams, denn man sieht dabei ja nicht auf den ersten Blick, welche Klasse hier instanziiert wird.

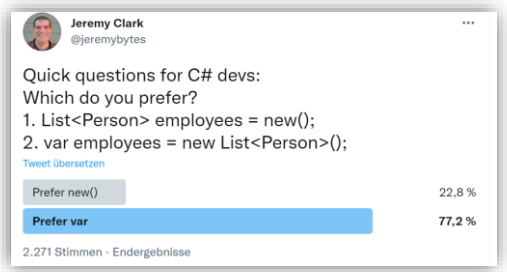


Abbildung: Abstimmungsergebnis auf Twitter zum Einsatz von new() ohne Typangabe [twitter.com/jeremybytes/status/1458105599623761929]

14.3 Objektinitialisierung

Ursprünglich konnte man Objekte nur prägnant und elegant bei der Instanziierung initialisieren, sofern die Klassen entsprechende Parameter im Konstruktor anboten.

Seit C# 3.0 und Visual Basic .NET 9.0 kann nun jedes öffentliche Attribut (egal ob Field oder Property) bei der Instanziierung initialisiert werden. C# bietet dazu eine Schreibweise mit geschweiften Klammern an, Visual Basic .NET das Schlüsselwort with (In Visual Basic .NET ist außerdem zu beachten, dass immer dem Attributnamen ein Punkt voranzustellen ist, in C# jedoch nicht!)

Hinweis: Man kann nur öffentliche und beschreibbare Attribute der Klasse von außen initialisieren. Man muss keineswegs alle Attribute initialisieren. Man darf aber jedes Attribut nur einmal initialisieren.

Listing: Initialisierung von Objekten bei der Instanziierung (C# seit Version 3.0)

```
Vorstandsmitglied MM = new Vorstandsmitglied() { Name = "Max Müller",
Aufgabengebiet = "Flugbetrieb", Alter = 33 };
Vorstandsmitglied HM = new Vorstandsmitglied() { Name = "Hans Meier",
Aufgabengebiet = "Personal", Alter = 42 };
Vorstandsmitglied HS = new Vorstandsmitglied() { Name = "Hubert Schmidt",
Aufgabengebiet = "Marketing",Alter = 35, Ort = "Essen" };
```

Hinweis: Man kann die Objektinitialisierung auch zusätzlich verwenden, wenn es einen parameterbehafteten Konstruktor gibt, z.B.

Vorstandsmitglied HS = new Vorstandsmitglied("Hubert Schmidt") { Aufgabengebiet = "Marketing", Alter = 35, Ort = "Essen" };

Man kann die Objektinitialisierung seit C# 9.0 auch mit dem Feature "Target Type New" verwenden, also den Klassennamen nach new weglassen:

Vorstandsmitglied HS = new("Hubert Schmidt") { Aufgabengebiet = "Marketing", Alter = 35, Ort = "Essen" };

14.4 Geschachtelte Klassen (eingebettete Klassen)

Klassendefinitionen können Klassendefinitionen (innere Klassen) enthalten.

```
class PersonMitAdresseClient
 {
  public static void Run()
  {
   var p = new PersonMitAdresse();
   p.Adresse = new PersonMitAdresse.AdressKlasse();
   p.Name = "Holger Schwichtenberg";
   p.Adresse.Ort = "Essen";

  }

 }
 /// <summary>
 /// Äußere Klasse
 /// </summary>
 class PersonMitAdresse
 {
  public class AdressKlasse
  {
   public string Strasse { get; set; }
   public string PLZ { get; set; }
   public string Ort { get; set; }
  }

  public int ID { get; set; }
  public string Name { get; set; }
  public AdressKlasse Adresse { get; set; }
 }
}
```

14.5 Sichtbarkeiten/ Zugriffsmodifizierer für Klassen und Klassenmitglieder

Die Zugriffsmöglichkeiten auf Klassen und Klassenmitglieder wird durch sogenannte Zugriffsmodifizierer gesteuert.

Für Klassen gilt,

- Eine Klasse ist im Standard `internal`, d.h. sie sind nur in ihrem Projekt sichtbar, aber nicht in Projekten, die das Projekt referenzieren.
- Wenn das Kompilat des Projekts in anderen Projekten referenziert wird und die Klasse verwendbar sein soll, muss der Modifizierer `public` vor die Klasse geschrieben werden: `public class Person { … }`

- Seit C# 11.0 gibt es auch den Modifizierer `file`, d.h. die Klasse kann nur innerhalb der Datei verwendet werden, in der sie sich befindet.

Für eine innere Klasse (eine in eine andere Klasse eingebettete Klasse) kann man auch anwenden:

- `private`: Die Klasse kann nur innerhalb der äußeren Klasse verwendet werden
- `internal`: Die Klasse kann innerhalb der gleichen Assembly verwendet werden.
- `public`: Die Klasse kann auch in referenzierenden Assemblies verwendet werden.

Klassenmitglieder können folgende Sichtbarkeiten besitzen:

- `private`: Das Mitglied kann nur innerhalb der Klasse genutzt werden
- `protected`: Das Mitglied kann innerhalb der Klasse und in abgeleiteten Klassen genutzt werden
- `private protected`: Seit C# 7.2 (wie in Visual Basic .NET seit Version 15.5) möglich für Klassenmitglieder in einer abgeleiteten Klasse in der gleichen Assembly verwendet zu werden, nicht aber in anderen Assemblies.
- `internal`: Das Mitglied kann in allen Klassen innerhalb der Assembly genutzt werden
- `public`: Das Mitglied kann in allen Klassen auch in referenzierenden Assemblys genutzt werden

Hinweis: Visual Basic .NET und C# unterscheiden sich bei den Klassendefinitionen außer bei `friend`/`internal` nur hinsichtlich der Groß-/Kleinschreibung der Schlüsselwörter. In C# müssen die Schlüsselwörter klein geschrieben werden. In Visual Basic .NET ist dies egal, der Editor schreibt die Wörter allerdings automatisch groß.

14.6 File-local Types (seit C# 11.0)

Als letztes neues Sprachfeature, kurz vor dem Erscheinen von C# 11.0, hat Microsoft einen neuen Zugriffsmodifizierer für Typen eingeführt, um deren Sichtbarkeit auf die Dateiebene zu beschränken.

Seit C# 11.0 gibt es auch die Sichtbarkeit (Scope) `file` (neben den bisher bekannten `public`, `private`, `protected`, `internal`, `protected internal` und `private protected`).

Mit `file` deklarierte Schnittstellen, Klassen, Strukturen, Enumerationen, Delegates und Records sind nur innerhalb der Datei sichtbar, in der sie deklariert werden. Eingebettete Typen können nicht mit `file` versehen werden.

Hinweis: Jetzt wird vielen Lesern als erster Gedanke kommen: In C# ist doch "Best Practice" pro Datei nur einen einzigen Typ zu deklarieren. Wenn man diesen einen Typ dann mit `file` deklariert, ist er ja nicht sinnvoll, weil er nirgendwo anders sichtbar ist. Da sieht man wieder einmal, wie es mit "Best Practices" ist: Sie gelten eben nicht immer und überall 😊

In der Praxis kann es aber durchaus Sinn machen, mehrere kleinere Typen in einer Datei zu deklarieren, z.B. weil eine Klasse eine eigene, persönliche, d.h. nur für die Klasse geltende Datenstrukturen in Form einer anderen Klasse oder eines Record-Typen erhält. Tatsächlich eingeführt hat Microsoft den Scope `file` für die Source Generatoren: Sie sollen Hilfsklassen erzeugen können ohne in Konflikt mit anderen Generatoren zu geraten.

In den Programmcodebeispielen zu diesem Buch macht der neue Scope `file` auch durchaus Sinn: Jede Datei behandelt ein Sprachfeature. Als Beispiel wird oft der Typname `Person`

verwendet, aber immer wieder anders implementiert. Die bisherige Trennung der verschiedenen `Person`-Implementierungen in verschiedene Namensräume kann nun entfallen.

Das folgende Listing zeigt den Inhalt einer Datei, die drei Typen deklariert:

- Schnittstelle `IPerson` mit Scope `public`
- Klasse `Person`, die `IPerson` implementiert, mit Scope `file`
- Klasse `PersonManager` mit Scope `public`

Das Listing zeigt: `PersonManager` kann durchaus eine Instanz von `Person` an die Außenwelt (Code in anderen Dateien) liefern, denn diese können die Instanz ja über Schnittstelle `IPerson` verwenden. Die Außenwelt kann aber keine Instanz von `Person` hineinreichen, weil sie diese Klasse nicht kennt.

Listing: C11_FileScope.cs

```
namespace NET7Console;

public interface IPerson
{
 public int ID { get; set; }
 public string? Name { get; set; }
 public string GetInfo();
}

file class Person : IPerson
{
 public int ID { get; set; }
 public string? Name { get; set; }
 public string GetInfo() => $"{this.GetType().FullName} {this.ID}: {this.Name}";
}

public class PersonManager
{
 public int ID { get; set; }
 public string? Name { get; set; }

 public string GetInfoFromTestPerson()
 {
  Person p = new();
  return p.GetInfo();
 }

 public IPerson CreatePerson()
 {
  return new Person();
 }

 // Nicht möglich: File-
local type 'Person' cannot be used in a member signature in non-file-
local type 'PersonManager'.
 //public int GetInfo(Person p)
 //{
 // return p.GetInfo();
 //}
```

```
}
```

Hinweise: Typen mit `file`-Scope bekommen einen vom Compiler vergebenen Namenszusatz, der sie eindeutig macht. Der Namensaufbau ist: *<Dateiname>HEX-ZAHL__Typname* z.B.

NET7Console.<C11_FileScope>FA5B2AEDF9084311D7828CE3F0191286CC8A2A06CFD7ACD9E4A15DDB99FB91671__Person

Warnung: Ein Typ mit Scope file kann einen anderen Typen, der übergeordnet sichtbar ist, verdecken. Beispiel: Wenn es neben einer Klasse Person in der Datei Person.cs, die internal oder public ist, noch einen Klasse file class Person in Test.cs gibt, ist innerhalb dieser Datei Test.cs der Typ Person aus Person.cs nicht sichtbar!

14.7 Statische Klassen

An die Stelle des Visual Basic .NET-Schlüsselworts Module tritt in C# seit Version 2005 das Konstrukt static class. Eine solche Klasse darf nur statische Mitglieder besitzen. Die Klasse kann nicht von einer anderen .NET-Klasse explizit erben; sie erbt automatisch von System.Object..

Listing:Beispiel für eine statische Klasse in C#

```
static class StatischeKlasse
 {
  public static void StatischesMitglied() { … }
  // Nicht erlaubt: Instanzmitglied
  // public void InstanzMitglied();
 }
```

Eine statische Klasse kann nicht instanziiert werden, weil der Konstruktor automatisch als private deklariert ist. Dies ist also nicht erlaubt:

```
StatischeKlasse obj = new StatischeKlasse ();
```

Die statische Klasse kann nur über die Klasse selbst verwendet werden:

```
StatischeKlasse.StatischesMitglied();
```

Eine häufig verwendete statische Klasse aus der .NET-Klassenbibliothek ist System.Environment.

```
Console.WriteLine(Environment.OSVersion);
Console.WriteLine(Environment.UserName);
foreach (string s in Environment.GetLogicalDrives())
 {
  Console.WriteLine(s);
 }
```

15 Datenmitglieder / Attribute (Fields und Properties)

Attribute sind in der objektorientierten Lehre Datenmitglieder (alias Merkmal, Kennzeichen, Informationsdetail) einer Klasse (vgl. *de.wikipedia.org/wiki/Attribut_(Objekt)*). Microsoft kennt in der Programmiersprache C# und anderen .NET-Sprachen zwei Arten von Attributen und spricht von

- Feldern (engl. Fields) und
- Eigenschaften (engl. Properties)

Praxishinweis: Sie sollten grundsätzlich Properties für öffentliche Attribute bevorzugen, da einige Bibliotheken (insbesondere GUI-Bibliotheken) Properties für die Datenbindung erfordern. Für private Klassenmitglieder können auch Fields in Frage kommen.

15.1 Abweichungen von der Lehre

Leider weicht Microsoft bei C# von den Begriffen von der objektorientierten Lehre erheblich ab:

- Attribute einer Klasse nennt Microsoft Felder und Properties
- Attribute sind bei Microsoft hingegen Metadaten (in anderen Sprachen besser "Annotationen" bezeichnet).

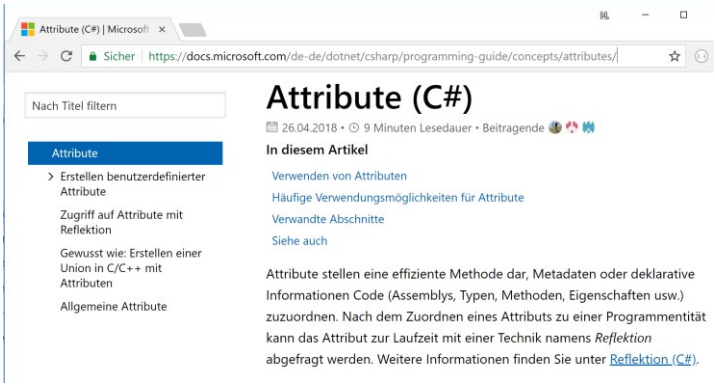


Abbildung: Microsofts Definition von "Attribute" [learn.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/attributes/]

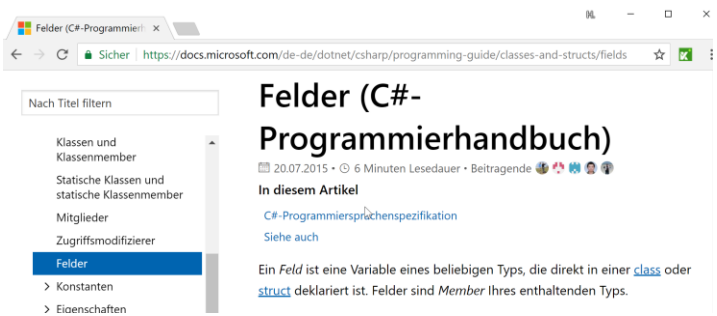


Abbildung: Microsofts Definition von "Feld" [learn.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/fields]

Hinweise: In diesem Buch werden – im Einklang mit der objektorientierten Lehre – die Datenmitglieder einer Klasse als "Attribute" bezeichnet. Was Microsoft "Attribut" nennt, finden Sie hier im Kapitel "Annotationen".

Felder sind in der Informatik eine Menge gleichartiger Daten (vgl. [*de.wikipedia.org/wiki/Feld_(Datentyp)*]). Hier bleibt das Buch aber bei der Verwendung des Begriffs

15.2 Felder (Field-Attribute)

Attribute (Daten) einer Klasse ohne Codehinterlegung werden – im Sprachjargon von Microsoft – durch "Felder" (engl. Fields) erzeugt.

15.2.1 Deklaration von Feldern

Felder können public (sichtbar für die Klasse und alle Nutzer), private (sichtbar nur für die Klasse) oder protected (sichtbar für die Klasse und geerbte Klassen) sein. In C# werden die Sichtbarkeitsmodifizierer vor den Field-Namen vorangestellt. Mehrere Fields gleichen Typs können durch ein Komma getrennt werden. Fields können bei der Deklaration explizit initialisiert werden durch eine Zuweisung. Wenn sie nicht explizit initialisiert werden, erhalten sie den Standardwert des Datentyps (z.B. 0 bei Zahlen, null bei Zeichenketten, false bei Boolean und den 1.1.0001 bei DateTime).

```
private string PersonalausweisNr;
public string Vorname, Nachname;
Protected System.DateTime Geburtstag;
Protected string Geburtsort = "unbekannt";
```

15.2.2 Felder mit readonly

Fields können auch mit dem Zusatz readonly deklariert werden. An ein readonly-Field kann man Werte nur in der Deklaration und letztmalig (!) im Konstruktor zuweisen. Danach sind sie unveränderlich.

Hinweis: Auch in einem Objekt-Initialisierer ist das Field dann nicht mehr änderbar! Dies ist ein Unterschied zu einem Property mit Init Only Setter. Ein Property mit Init Only Setter kann auch in einem Objekt-Initialisierer noch geändert werden!

```
public class Person
{
 #region Fields
  // Normales Fields ohne Initialiserung
  public DateTime ZuletztGeaendert;
  // Readonly Fields mit Initialiserung
  public readonly DateTime AngelegtAm = DateTime.Now;
 #endregion

 #region Konstruktoren
 public Person()
 {
  // letztmalige Änderungsmöglichkeit für das readonly field!
  this.AngelegtAm = DateTime.Now;
 }
…
}
```

15.3 Eigenschaften (Property-Attribute)

Ein Property dient dazu, ein Attribut (Datenmitglied) einer .NET-Klasse zu deklarieren, bei dem Programmcode sowohl beim Setzen des Werts als auch beim Lesen des Werts ausgeführt wird. Ein Property ist somit eine Mischung aus einem Attribut und einer Methode: Der Aufrufer sieht das Property als Attribut, die Klasse intern besitzt jedoch eine oder zwei Methoden: Die Get-Methode (alias **Getter**) zum Lesen und/oder die Set-Methode (alias **Setter**) zum Schreiben des Attributs. Getter und Setter können unterschiedliche Sichtbarkeiten besitzen (public, private, protected). Der Standard ist public.

In der deutschen Dokumentation verwendet Microsoft den Begriff "Eigenschaft" als Übersetzung für "Property", im Gegensatz zu den normalen (einfachen) Attributen, die Microsoft "Field" bzw. "Feld" nennt.

Was tatsächlich in Getter und Setter ausgeführt wird, ist dem Entwickler überlassen. Typische Beispiele für die Nutzung von Properties sind:

- Im Getter wird ein Wert berechnet, statt ihn aus dem Speicher zu lesen. Der Setter fehlt, weil es keinen Sinn macht, einen berechneten Wert zu speichern (z.B. Alter: Diese Property würde im Getter das Alter aus Geburtstag und aktuellem Datum errechnen. Einen Setter gäbe es nur für Geburtstag, aber nicht für Alter).
- Im Setter wird geprüft, ob der Wert Sinn macht (z.B. Geburtstag darf nicht in Zukunft liegen)
- Man darf einen Wert setzen, aber nicht wieder auslesen (z.B. Kennwort)

Architekturhinweis: Der Programmcode in Getter und Setter sollte nicht zu umfangreich werden. Er sollte nicht lang dauern und auch nichts unerwartetes tun, z.B. externe Ressourcen ansprechen. Aktionen, die länger dauern, sollten in Methoden implementiert werden.

Ursprünglich gab es nur explizite Properties. Seit C# 2008 gibt es auch automatische Properties.

15.3.1 Explizite Properties mit Field

Explizite Properties (ausformulierte Properties) sind der Grundtypus, bei dem man für Getter und Setter jeweils einen eigenen Programmcodeblock { … } schreibt. Dabei definiert man typischerweise ein zugehöriges privates Field als Datenspeicher.

> **Hinweis:** Das zu einem Property gehörige explizite Field kann der Entwickler beliebig benennen. Üblich ist aber, entweder den Namen des Properties (z.B. Flugstunden) mit kleinem Anfangsbuchstaben (flugstunden) zu verwenden oder aber dem Namen einen Unterstrich voranzustellen (_Flugstunden). Oft vereinbaren Entwicklungsteams dazu interne Konventionen.

Das folgende Property enthält noch keinen Programmcode außer dem Setzen und dem Lesen des privaten Fields.

Listing: Ein Property mit zugehörigem Field in expliziter Schreibweise

```
private long _Flugstunden;
public long Flugstunden
{
 get
 {
  return this._Flugstunden;
 }
 protected set
 {
  this._Flugstunden = value;
 }
}
```

Die folgende Variante enthält im Setter eine Validierung.

Listing: Ein Property mit zugehörigem Field in expliziter Schreibweise und Validierung im Setter

```
private long _Flugstunden;
public long Flugstunden
{
 get
 {
  return this._Flugstunden;
 }

 protected set
 {
  if (value < 0) throw new ApplicationException("Ungültiger Wert");
  this._Flugstunden = value;
 }
```

Seit C# 7.0 kann Properties auch verkürzt per Lambda-Expression implementieren.

Listing: Ein Property mit zugehörigem Field in expliziter Schreibweise per Lambda

```
private long _FlugStunden;
public long Flugstunden
{
 get => this._Flugstunden;
 protected set => this._Flugstunden = value;
}
```

Man kann die Lambda-Schreibweise für Getter und Setter getrennt wählen. So ist es z.B. möglich, in dem Setter noch Programmcode zu hinterlegen (z.B. zur Validierung), während der Getter prägnant per Lambda-Ausdruck nur den Wert des privaten Fields abruft.

```
private long _FlugStunden;
public long Flugstunden
{
 get => this._Flugstunden;
 protected set
 {
  if (value < 0) throw new ApplicationException("Ungültiger Wert");
  this._Flugstunden = value;
 }
}
```

Hinweis: Bei ausformulierten Properties kann man get weglassen, wenn der Lesezugriff nur über das korrespondiere Field erfolgen soll. Auch das Field ist optional; man kann den Wert auch woanders speichern. Theoretisch kann man ein Property auch wie eine Methode mit einem Parameter verwenden. Dies ist aber kein guter Programmierstil.

15.3.2 Automatische Properties

Die automatischen Eigenschaften (engl. Automatic Property) machen die Syntax prägnanter für solche Property-Attribute, die nichts anderes tun als ein privates Field-Attribut zu lesen und zu beschreiben. In diesem Fall kann man sich die explizite Definition des privaten Field-Attributs sparen und die Erzeugung dem Compiler überlassen. Damit verkürzt sich auch die Schreibweise von Getter und Setter radikal. Automatische Eigenschaften gibt es in C# seit Version 3.0 und in Visual Basic seit Version 2010. Seit C# 6.0 und Visual Basic 14 kann man automatische Properties auch direkt bei der Deklaration initialisieren.

Ein Property in C# mit zugehörigem Field als automatisches Property deklariert man so:

```
public long Flugstunden { get; set; }
```

Getter und Setter können unterschiedliche Sichtbarkeiten besitzen:

```
public long Flugstunden { get; protected set; }
```

Hinweis: Eine Validierung oder andere Logik ist bei automatischen Properties nicht möglich. Bei einer automatischen Property erzeugt der Compiler ein privates Field, dessen Namen der Entwickler nicht kennt und nicht sieht. Er kann es nicht ansprechen, d.h. alle Zugriffe laufen über das Property. Erst ab 7.3 kann man Annotationen für diese automatisch generierten privaten Fields setzen.

Wenn keine Validierung oder andere Logik notwendig ist, sollte man für öffentliche Klassenmitglieder dennoch immer ein automatisches Property realisieren und nicht der Versuchung verfallen, ein Field anzulegen. Einige Bibliotheken wie die Windows Presentation Foundation (WPF) erfordern Properties für die Datenbindung.

Seit C# 6.0 kann man automatische Properties, für die es keine explizite Felddeklaration gibt, direkt im Rahmen der Deklaration mit einem Wert initialisieren und auch automatische Properties schaffen, die nach ihrer Initialisierung unveränderbar sind, indem sie nur einen Getter besitzen. Lediglich im Konstruktor der Klasse kann der Entwickler solche Eigenschaften dann noch letztmalig ändern. Eine normale Methode oder der Nutzer des Objekts kann das Property nicht verändern.

Hinweis: Man darf den Setter bei automatischen Properties weglassen, aber es muss immer einen Getter geben!

Listing: Automatische Properties mit Initialisierung und optional auch ohne Setter

```
public class Kontakt
 {
  // Automatic Properties mit Initialisierung
  public string Land { get; set; } = "Deutschland";
  // Automatic Properties mit Initialisierung und ohne Setter
  public DateTime ErzeugtAm { get; } = DateTime.Now;

  public Kontakt(DateTime erzeugtAm)
  {
   // Getter Only Auto Property im Konstruktor setzen
   ErzeugtAm = DateTime.Now;
  }
}
```

15.3.3 Properties, die nach Initialisierung unveränderlich sind (Init Only Properties)

Seit C# 9.0 gibt es zusätzlich auch automatische Properties, deren Werte nur bei der Objektinitialisierung (Konstruktionsphase) gesetzt werden können und die danach unveränderlich sind. Man nennt diese Properties "**Init Only Property**" und sie werden mit dem "**Init Only Setter**" deklariert. Man kann Init Only Properties in Klassen, Strukturen und Record-Typen verwenden.

Ein solcher "Init Only Setter" verwendet das Schlüsselwort init anstelle von set:

```
class Person
{
   public int ID { get; init; }
…
}
```

Dies geht bei automatischen Properties ebenso wie bei ausformulierten Properties:

```
private int id;
public int ID
{
 get { return id; }
 init { id = value; }
}
```

und auch Properties in der Lambda-Schreibweise:

```
private int id;
public int ID
{
 get => id;
 init => id = value;
}
```

Hinweis: *init* und *set* dürfen nicht beide verwendet werden. Bei ausformulierten Properties kann man *get* weglassen; bei automatischen Properties nicht.

Das Init Only Property kann ein Softwareentwickler nur noch bei der Objektinitialisierung (Konstruktionsphase) eines Objekts setzen, also an zwei Stellen:

- Konstruktor der Klasse

```
public Person(int ID)
{
   this.ID = ID;
}
…
Person hs1 = new Person(123);
```

- Objekt-Initialisier direkt bei der Instanziierung

```
Person hs2 = new Person() { ID = 123 } ;
```

Bei der Instanziierung ist auch möglich, den Wert sowohl per Konstruktorparameter zu setzen als auch per Objekt-Initialisier einen (ggf. abweichenden) Wert zu setzen.

```
Person hs3 = new Person(123) { ID = 456 };
```

Das folgende Listing zeigt ein komplettes Beispiel für die Klasse mit zwei Properties mit Init Only Setters. Der Nutzer der Klasse darf nach der Instanziierung nur noch den Nachnamen (Surname) verändern, aber nicht den Vornamen und die ID der Person.

Hinweis: Auch innerhalb der Klasse können die Properties mit Init Only Setters nicht mehr verändert werden. Eine Methode ChangeID() ist nicht erlaubt!

```
public static void InitOnlysetters()
  {
   Person p1 = new Person(123, "Susanne", "Müller");
   Person p2 = new Person("Susanne", "Müller") { ID = 123 } ;
   Person p3 = new Person(123, "Susanne", "Müller") { ID = 456 };

   p3.Surname = "Schulze";
   // p3.Firstname = "Marianne";// verboten durch "Init Only Setter"!
   // p3.ID = 456; // verboten durch "Init Only Setter"!
  }

  class Person
  {
   public int ID { get; init; }
   public string Firstname { get; init; }
   public string Surname { get; set; }

   public Person()
   {   }
   public Person(int ID)

   {
    this.ID = ID;
   }

   public Person(string firstname, string surname) : this()
   {
    this.Firstname = firstname;
    this.Surname = surname;
   }

   public Person(int id, string firstname, string surname) : this(id)
   {
```

```
    this.Firstname = firstname;
    this.Surname = surname;
   }

   public override string ToString()
   {
    return this.Firstname.ToUpper() + " " + this.Surname.ToUpper();
   }

   // verboten durch "Init Only Setter"!
   //public void ChangeID(int newID)
   //{
   // this.ID = newID;
   //}
  }
```

15.3.4 Init Only Setters in .NET Framework und .NET Standard

Bei der Verwendung von Init Only Setters in Projekten, die auf .NET Framework oder .NET Standard basieren, kommt es zur Fehlermeldung "Error CS0518:Predefined type 'System.Runtime.CompilerServices.IsExternalInit' is not defined or imported".

```
class Person
{
 public int ID { get; init; }
```

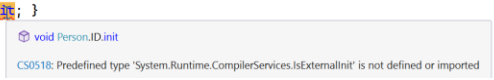


Abbildung: Fehlermeldung bei einem Init Only Setter

Das passiert selbst nach der Erhöhung der <LangVersion> auf eine Zahl >= 9, weil es die Klasse System.Runtime.CompilerServices.IsExternalInit im klassisches .NET Framework und .NET Standard nicht gibt.

Allerdings kann man dies mit einem Trick lösen: Diese Klasse ist eine Annotation (.NET-Attribute), das man selbst implementieren kann. Man fügt folgenden Programmcode einfach in jedes Projekt ein.

Listing: Hack für die Verwendung von C# >= 9 in .NET Standard und .NET Framework

```
using System.ComponentModel;

namespace System.Runtime.CompilerServices
{
 [EditorBrowsable(EditorBrowsableState.Never)]
 public class IsExternalInit { }
}
```

15.3.5 Zusammenfassung zu Properties

Das folgende Listing zeigt alle Spielarten von Properties im Vergleich.

```
public class Person
 {
…
  #region Properties
```

```
  // Automatisches Property mit öffentlichem Getter und Setter
  public string Name { get; set; }
  // Automatisches Property mit öffentlichem Getter und privatem Setter und Initi
alisierung
  public char Geschlecht { get; private set; } = '?';
  // Explizites Property mit privatem Field und öffentlichem Getter und Setter
  private string _Vorname;
  public string Vorname
  {
   get { return _Vorname; }
   set
   {
    if (String.IsNullOrEmpty(value)) throw new ApplicationException("Vorname darf
 nicht leer sein");
    if (value.Length>50) _Vorname = value.Substring(0,50);
    else _Vorname = value;
   }
  }
  // Explizites Property nur mit Getter (berechnetes Property)
  public string GanzerName
  {
   get { return $"{Vorname} {Name}"; }
  }
  // Explizites Property nur mit Getter (berechnetes Property),Lambda-Syntax
  public string GanzerNameMitGeschlecht => $"{Vorname} {Name} ({Geschlecht})";
  // Explizites Property nur mit Setter
  private string _KennwortHash;
  public string Kennwort
  {
   set { _KennwortHash = value.GetHashCode().ToString(); }
  }
  #endregion
 }
```

15.4 Pflichtmitglieder (Required Members)

Seit C# 11.0 gibt es ein neues Schlüsselwort required für Fields und Properties. Wenn ein Datenmitglied einer Klasse diesen Zusatz erhält, dann ist zwingend erforderlich, dass dieses Datenmitglied entweder im Konstruktor oder Objekt-Initialisierer vom Nutzer der Klasse gesetzt wird. Ein Konstruktor ist mit [SetsRequiredMembers] annotierbar, was dem Compiler anzeigt, dass er alle erforderlichen Mitglieder belegt.

Hinweis: Der Zusatz required ist erlaubt bei Datenmitgliedern in Klassen, Strukturen und Record-Typen, aber nicht in Schnittstellen.

Beispiel: Die Klasse im folgenden Listing deklariert ein Field und zwei Properties mit required sowie eine weitere Property ohne diesen Zusatz. Zudem gibt es neben dem parameterlosen Konstruktor zwei weitere Konstruktoren mit Parametern, die beide mit [SetsRequiredMembers] annotiert sind; allerdings setzt nur einer von beiden alle drei der erforderlichen Mitglieder auf belegt.

Achtung: Der Code kompiliert auch, wenn [SetsRequiredMembers] gar nicht alle erforderlichen Mitglieder setzt, siehe zweiter Konstruktor im folgenden Listing. Es gibt auch keine Warnung! Das heißt: Der Compiler verlässt sich auf die Angabe [SetsRequiredMembers]

des Entwicklers! Es gab den Plan, dass der Compiler das tatsächliche Setzen aller Pflichtmitglieder validiert; er wurde jedoch verworfen. Es gab beim C#-Entwicklungsteam den Plan, dass man einzelne Mitglieder ein- und ausschließen kann. Auch dies ist Stand C# 11.0 nicht möglich.

"An earlier version of this proposal had a larger metalanguage around initialization, allowing adding and removing individual required members from a constructor, as well as validation that the constructor was setting all required members. This was deemed too complex for the initial release, and removed. We can look at adding more complex contracts and modifications as a later feature." [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/required-members]

Listing: CS11_Required.cs

```
public class Consultant
{
 public Consultant() { }

 [SetsRequiredMembers]
 public Consultant(int id, string name, DateTime created) => (ID, Name, Created)
= (id, name, created);
 [SetsRequiredMembers]
 public Consultant(int id, string name) => (ID, Name) = (id, name);

 public required int ID; // Required Field
 public required string Name { get; init; } // Required Property
 public required DateTime Created { get; init; } = DateTime.Now; // Required Prop
erty

 public string? City { get; set; } // nicht "required"!
}
```

Diese Klasse ist nun wie folgt instanziierbar.

1. Aufruf des Konstruktors mit allen drei erforderlichen Angaben:

```
var p1 = new Consultant(1, "Dr. Holger Schwichtenberg", DateTime.Now);
```

2. Aufruf des Konstruktors mit nur zwei der drei Pflichtangaben:

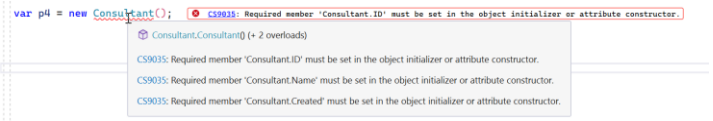
```
var p2 = new Consultant(2, "Dr. Joachim Fuchs");
```

3. Aufruf des parameterlosen Konstruktors und Initialisierung aller drei Angaben im Objekt-Initialisierer:

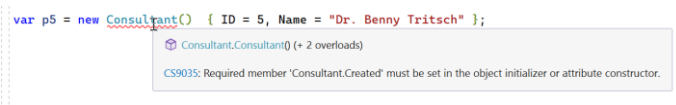
```
var p3 = new Consultant() { ID = 3, Name = "Dr.habil. Klaus Schmaranz",
                            Created = DateTime.Now  };
```

Nicht erlaubt ist hingegen:

- Parameterloser Konstruktor ohne Objekt-Initialisierer

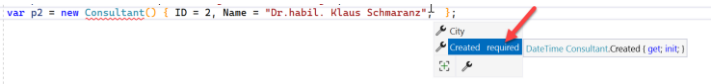


- Parameterloser Konstruktor mit unvollständigem Objekt-Initialisierer



Praxishinweis: Das Beispiel zeigt auch: Es reicht nicht, dass das Property Created eine Standardwertzuweisung in der Klasse besitzt. Der Aufrufer muss trotzdem Created belegen.

Visual Studio zeigt übrigens in den Tooltips deutlich an, wenn das Setzens eines Mitglieds erforderlich ist.



16 Methoden

Methoden sind Operationen in Klassen, die innerhalb der Klasse oder von Nutzern aufgerufen werden können. Methoden können einen Rückgabewert liefern. Parameter von Methoden können optional sein. Weggelassene Parameter werden durch Vorgabewerte ersetzt, die in der Methodendeklaration stehen müssen. Der Aufrufer gibt in der Regel die Parameter in der Deklaration vorgegebenen Reihenfolge an. Durch eine spezielle Syntax kann man aber die Parameter in einer beliebigen Reihenfolge angeben. Optionale Parameter dürfen Wertelose Wertetypen (Nullable Types) sein.

16.1 Methodendefinition und Rückgabewerte

In C# beginnt eine Methodendefinition mit der Sichtbarkeit. Danach folgt der Datentyp des Rückgabewerts. In C# gibt es kein direktes Schlüsselwortpendant zum Sub und Function aus Visual Basic .NET. Methoden ohne Rückgabewerte werden durch den Datentyp void signalisiert.

Der Rückgabewert wird in C# wie in Visual Basic .NET mit return festgelegt.

```
public class MethodenDemo
 {
  /// <summary>
  /// Methode ohne Rückgabewert
  /// </summary>
  public void DruckeUhrzeit()
  {
   Console.WriteLine("Aktuelle Uhrzeit: " + DateTime.Now.ToShortTimeString());
  }

  /// <summary>
  /// Methode mit Zeichenkette als Rückgabewert
  /// </summary>
  public string GetUhrzeit()
  {
  return (DateTime.Now.ToShortTimeString());
  }
}
```

Beim Methodenaufruf sind immer runde Klammern zu verwenden, auch wenn es keine Parameter gibt!

```
DruckeUhrzeit();
var Uhrzeit = GetUhrzeit();
Console.WriteLine(Uhrzeit);
```

16.2 Methodenparameter

Eine Methode kann eine Parameterliste besitzen, wobei der Typ – wie bei Variablendeklarationen – auch hier jeweils vor dem Parameternamen genannt wird.

```
  /// <summary>
  /// Methode mit Parametern
  /// </summary>
  public double Berechnen(int a, int b, double c)
  {
   return (a + b) / Math.Pow(c, 2);
```

```
}
```

16.3 Methodenüberladungen

Methoden können überladen sein, d.h. der gleiche Methodenname darf mehrfach mit verschiedenen Parameterlisten verwendet werden, sofern beim Aufruf die Zuordnung zu einer der Überladungen noch eindeutig ist. Für überladene Methoden gibt es kein Schlüsselwort in C#, während Visual Basic .NET dafür Overloads verwendet.

```
/// <summary>
/// Überladene Methode mit Parametern
/// </summary>
public double Berechnen(double a, double b, double c)
{
 return (a + b) / Math.Pow(c, 2);
}
```

Bei den folgenden Aufrufen geht der erste Aufruf an die erste Variante mit den zwei Int-Werten in den ersten Parametern, während der zweite Aufruf die Überladung mit den double-Werten aufrufen muss, da 2.8 nicht in int a passen würde. Dass der zweite Parameterwert hier kein double ist, stört nicht. Der Compiler konvertiert automatisch die 3 in 3.0.

```
Console.WriteLine(Berechnen(2,3, 4.456)); // Ruft die erste Überladung
Console.WriteLine(Berechnen(2.8, 3, 4.456)); // Ruft die zweite Überladung
```

Hinweis: Überladungen müssen sich hinsichtlich der Parameteranzahl und Parametertypen unterscheiden. Nicht gültig ist, wenn sich zwei Methodendeklarationen nur hinsichtlich des Rückgabetyps oder den Zusätzen out und ref zu den Parametern unterscheiden.

16.4 Prioritäten für Methodenüberladungen (ab C# 13.0)

Die in C# 13.0 und .NET 9.0 neu eingeführte Annotation [OverloadResolutionPriority] im Namensraum System.Runtime.CompilerServices bietet eine bedeutende Verbesserung für Überladungen von Methoden: Diese Annotation ermöglicht es, die Priorität von Überladungen explizit festzulegen, um die Entscheidung, welche Methodenüberladung der Compiler aufrufen soll, gezielt zu steuern.

Mit [OverloadResolutionPriority] können Entwicklerinnen und Entwickler festlegen, dass bestimmte Überladungen bei der Entscheidung, welche Überladung verwendet werden soll, eine höhere Priorität erhalten sollen. Dies hilft zum Beispiel, wenn mit [Obsolet] annotierte Überladungen einer Methode existieren. Bei der neuen Annotation [OverloadResolutionPriority] gibt man eine Integer-Zahl an: **Je höher die in der Annotation angegebene Zahl ist, je höher die Priorität.**

Das folgende Listing zeigt ein Beispiel: Der Aufruf von Print() mit einer Zeichenkette würde ohne [OverloadResolutionPriority] immer zur Implementierung von Print() mit einem String-Parameter gehen, auch wenn diese Überladung als [Obsolete] gekennzeichnet ist. Durch das Einfügen von [OverloadResolutionPriority] kann man den Compiler auf eine andere Implementierung umlenken. Würde man in dem Beispiel sowohl der Implementierung mit Parametertyp object als auch ReadOnlySpan<char> den gleichen Prioritätswert geben, wüsste der Compiler nicht, welche Konvertierung er machen soll und verweigert die Übersetzung:

The call is ambiguous between the following methods or properties:
'CS13_OverloadResolutionPriority.Print(object, ConsoleColor)' and
'CS13_OverloadResolutionPriority.Print(ReadOnlySpan<char>, ConsoleColor)'

Mit einem abweichenden Prioritätswert kann man den Compiler zu der einen oder der anderen Implementierung lenken, hier im Listing mit Wert 10 zu public void Print(ReadOnlySpan<char> text, ConsoleColor color).

Die Implementierung public void Print(object text, ConsoleColor color) kommt aber weiterhin zum Einsatz für alle anderen Datentypen, zum Beispiel Zahlen wie 42, denn diese kann der Compiler nicht automatisch in ReadOnlySpan<char> konvertieren.

Listing: Einsatz der neuen Annotation [OverloadResolutionPriority]

```
using System.Runtime.CompilerServices;

namespace NET9_Console.CS13;

public class CS13_OverloadResolutionPriority
{
 public void Run()
 {
  CUI.Demo(nameof(CS13_OverloadResolutionPriority));

  // verwendet Print(ReadOnlySpan<char> text)
  ReadOnlySpan<char> span = "www.IT-Visions.de".AsSpan();
  Print(span);

  // verwendet Print(ReadOnlySpan<char> text) wegen OverloadResolutionPriority(10
)
  Print("Dr. Holger Schwichtenberg");

  // verwendet public void Print(object obj)
  Print(42);

 }

 [Obsolete]
 //[OverloadResolutionPriority(10)]
 public void Print(string text)
 {
  // Set the console color
  Console.ForegroundColor = ConsoleColor.Red;

  // Print the text
  Console.WriteLine("string: " + text);

  // Reset the console color
  Console.ResetColor();
 }

 [OverloadResolutionPriority(1)]
 public void Print(object obj)
 {
  // Set the console color
```

```
 Console.ForegroundColor = ConsoleColor.Yellow;

 // Print the text
 Console.WriteLine("Object: " + obj.ToString());

 // Reset the console color
 Console.ResetColor();
}

[OverloadResolutionPriority(10)]
public void Print(ReadOnlySpan<char> text)
{
 // Set the console color
 Console.ForegroundColor = ConsoleColor.Green;

 // Print the text
 Console.WriteLine("ReadOnlySpan<char>: " + text.ToString());

 // Reset the console color
 Console.ResetColor();
}

}
```

```
CS13_OverloadResolutionPriority
ReadOnlySpan<char>: www.IT-Visions.de
ReadOnlySpan<char>: Dr. Holger Schwichtenberg
Object: 42
```

Abbildung: Ausgabe des Listings

Würde man bei public void Print(string text, ConsoleColor color) auch eine Overload Resolution Priority von mindestens 10 setzen

```
 [Obsolete]
[OverloadResolutionPriority(10)]
public void Print(string text, ConsoleColor color)
{
 // Set the console color
 Console.ForegroundColor = color;

 // Print the text
 Console.WriteLine("string: " + text);

 // Reset the console color
 Console.ResetColor();
}
```

dann wird bei

```
Print("Dr. Holger Schwichtenberg", ConsoleColor.Yellow);
```

die Überladung mit string-Parameter genommen, auch wenn diese mit [Obsolete] markiert ist.

16.5 Optionale und benannte Parameter

Seit C# 4.0 gibt es optionale und benannte Parameter. Zuvor musste man optionale Parameter durch Methodenüberladung nachbilden. Optionale Parameter werden in C# durch einen Vorgabewert in dem Methodenkopf angezeigt. Optionale Parameter dürfen nur am Ende der Parameterliste erscheinen.

Listing: Methode mit zwei optionalen Parametern

```
/// <summary>
/// Methode mit zwei optionalen Parametern
/// </summary>
public void Print(string text, ConsoleColor Farbe = ConsoleColor.Gray, bool Datum
 = false)
  {
   if (Datum) text = DateTime.Now.ToShortTimeString() + ": " + text;
   ConsoleColor bisherigeFarbe = Console.ForegroundColor;
   Console.ForegroundColor = Farbe;
   Console.WriteLine(text);
   Console.ForegroundColor = bisherigeFarbe;
  }
```

Die obige Methode kann man wie folgt aufrufen:

```
CS10_Parameter obj = new CS10_Parameter();
obj.Print("Ausgabe ohne spezielle Farbe und ohne Datum.");
obj.Print("Ausgabe in grün und ohne Datum.", ConsoleColor.Green);
obj.Print("Ausgabe in grün und mit Datum.", ConsoleColor.Green, true);
```

Benannte Parameter erlauben die Angabe der Parameter in beliebiger Reihenfolge unabhängig von der Reihenfolge in der Deklaration. Ein benannter Parameter ist allein Sache des Aufrufers, d.h. hierzu sind keine Änderungen in der Deklaration notwendig. Der Aufrufer gibt durch Parametername und Doppelpunkt an, welchen Parameter er übergeben will.

```
obj.Print(text: "Ausgabe ohne spezielle Farbe und mit Datum.", Datum: true);
```

Von C# 4.0 bis C# 7.1 konnte man zwar benannte Parameterwerte und unbenannte Parameterwerte mischen in einem Aufruf, aber es galt die Regel, dass unbenannte Parameterwerte nur am Anfang vor dem ersten benannten Parameterwert verwendet werden dürfen. Dies wurde erst in C# 7.2 aufgehoben ("Non-trailing named arguments").

```
// Aufruf gemischt mit unbenannten und benannten Parametern
obj.Print("Ausgabe ohne spezielle Farbe und mit Datum.", Datum: true);
obj.Print("Ausgabe ohne spezielle Farbe und mit Datum.", ConsoleColor.Green, Datu
m: true);
obj.Print("Ausgabe ohne spezielle Farbe und mit Datum.", Farbe: ConsoleColor.Gree
n, Datum: true);

// erst ab C# 7.2 möglich: Benannte und unbenannte Parameter an beliebiger Stelle
obj.Print(text: "Ausgabe ohne spezielle Farbe und mit Datum.", ConsoleColor.Green
, true);
```

Achtung: Wenn man das Kompilat eines optionalen Parameteraufrufs mit einem Decompiler betrachtet, wird man überrascht: Die Aufrufe erfolgen gar nicht mit weniger Parametern, vielmehr werden die Vorgabewerte mit in den Aufruf hineinkompiliert. Das gilt sowohl für C# als auch Visual Basic.

```
CS40_Parameter obj = new CS40_Parameter();
obj.Print("Ausgabe ohne spezielle Farbe und ohne Datum.", ConsoleColor.Gray, false);
obj.Print("Ausgabe in grün und ohne Datum.", ConsoleColor.Green, false);
obj.Print("Ausgabe in grün und mit Datum.", ConsoleColor.Green, true);
```

Abbildung: Dekompilat mit ILSpy [github.com/icsharpcode/ILSpy]

In der Verwendung optionaler Parameter besteht also eine Gefahr: Wenn die optionale Methode in einer anderen Assembly als der Aufrufer ist und diese beiden Assemblys unabhängig voneinander kompiliert werden (also nicht in einer Projektmappe sind), dann kann es zu Inkonsistenzen kommen. Nach einer Änderung der Vorgabewerte würden nicht erneut kompilierte Aufrufer weiterhin die alten Werte verwenden.

16.6 Parametermodifizierer in, ref und out

Mit dem Zusatz in bei einem Parameter deklariert eine Methode, dass sie den übergebenen Parameter nur lesen, aber nicht verändern wird.

Für die Übergaberichtung der Parameter vom Aufrufer an eine Methode gibt es in C# für den Call-by-Value-Fall (Übergabe als Wert) kein Schlüsselwort und für den Call-by-Reference-Fall (Übergabe eines Zeigers) zwei Wörter:

- Der Zusatz ref vor einem Parameter (entspricht ByRef in Visual Basic .NET) bedeutet, dass der Wert bzw. das Objekt von außen hereingegeben wird und innerhalb der Methode verändert werden darf. Seit C# 12.0 gibt es auch ref readonly. Mit diesem Zusatz darf die Methode den empfangenen Wert bzw. die empfangene Objektreferenz nicht ändern. Bei der Übergabe von Referenztypen per ref readonly kann die Methode aber weiterhin die einzelnen Objektinhalte ändern.
- Der Zusatz out vor einem Parameter bedeutet, dass der Aufrufer nur leeren (nicht initialisierten) Speicherplatz hereingibt. Der Wert muss zwangläufig von der Methode selbst gesetzt werden und wird dann dem Aufrufer geliefert.

Hinweise: Wichtig ist, dass man nicht nur in der Methodensignatur selbst out und ref verwenden muss, sondern auch beim Aufruf der Methode.

Zudem ist zu beachten, dass keine Properties als Zeiger übergeben werden können!

Referenztypen werden immer als Zeiger übergeben! Wenn ein Referenztyp übergeben wird, kann die aufgerufene Methode immer die Daten im Objekt ändern. Die Modifizierer verhindern dann ggf. nur, dass ein anderes Objekt zugewiesen wird!

Wichtig für das Verhalten ist, ob als Parameter ein Wertetyp oder ein Referenztyp übergeben wird, siehe Tabelle.

	Parameter ist Wertetyp	**Parameter ist Referenztyp**	
	Methode kann Wert ändern	Methode kann einzelne Werte im übergebenen Objekt ändern	Methode kann neues Objekt zuweisen
Übergabe ohne Modifizierer	Ja, aber Aufrufer bekommt den neuen Wert nicht	Ja	Ja, aber Aufrufer bekommt das neue Objekt nicht
Übergabe mit in	Nein	Ja	Nein

	Parameter ist Wertetyp	Parameter ist Referenztyp	
	Methode kann Wert ändern	Methode kann einzelne Werte im übergebenen Objekt ändern	Methode kann neues Objekt zuweisen
Übergabe mit ref	Ja	Ja	Ja
Übergabe mit ref readonly (seit C# 12.0)	Nein	Ja	Nein
Übergabe mit out	Ja	Ja	Ja

Tabelle: Unterschiedliche Auswirkungen der Parametermodifizierer bei Übergabe von Wertetypen und Referenztypen

Die folgenden drei Listings zeigen dazu Beispiele inklusive eines Screenshots der jeweiligen Bildschirmausgaben.

Listing: Wirkung der Parametermodifizierer, wenn Parameter Wertetyp ist

```
/// <summary>
/// Wertetypen an Methode übergeben
/// </summary>
public void ParameterValueTypes()
 {
  CUI.H2(nameof(ParameterValueTypes));
  #region
  int a = 10;
  int b = 20;
  int c = 30;
  int d = 40;
  int e = 50;
  CUI.H3("Der Aufrufer hat vorher folgende Werte:");
  Console.WriteLine(a + ";" + b + ";" + c + ";" + d + ";" + e);
  string r = ParameterDemoValueTypes(a, b, ref c, ref d, out e);
  CUI.H3("Die Methode hat folgende Werte:");
  Console.WriteLine(r); // 11;20;31;40
  CUI.H3("Der Aufrufer hat nachher folgende Werte:");
  Console.WriteLine(a + ";" + b + ";" + c + ";" + d + ";" + e);
  #endregion
 }

public string ParameterDemoValueTypes(int WertValue, in int WertIn, ref int
WertRef, ref readonly int WertRefRO, out int WertOut)
{
  WertValue++;
  // nicht erlaubt, da in-Wert: WertIn++;
  WertRef++;
  // WertRefRO++; // nicht erlaubt, da readonly
  // nicht erlaubt, da noch nicht initialisiert: WertOut++;
  WertOut = 41;
  return WertValue.ToString() + ";" + WertIn.ToString() + ";" +
WertRef.ToString() + ";" + WertOut.ToString();
```

```
}
```

```
ParameterValueTypes
Der Aufrufer hat vorher folgende Werte:
10;20;30;40;50
Die Methode hat folgende Werte:
11;20;31;41
Der Aufrufer hat nachher folgende Werte:
10;20;31;40;41
```

Abbildung: Ausgabe des vorherigen Listings

Listing: Parameter ist Referenztyp (class Counter). Methode ändert Wert im Objekt

```
class Counter
{
 public string Name { get; set; }
 public int Value { get; set; }
 public override string ToString() => Name + "=" + Value;
}

/// <summary>
/// Referenztypen an Methode übergeben, die Wert in dem Objekt ändert
/// </summary>
public void ParameterReferenceType1()
{
  CUI.H2(nameof(ParameterReferenceType1));
  Counter a = new Counter() { Name = "a", Value = 10 };
  Counter b = new Counter() { Name = "b", Value = 20 };
  Counter c = new Counter() { Name = "c", Value = 30 };
  Counter d = new Counter() { Name = "d", Value = 40 };
  Counter e = new Counter() { Name = "e", Value = 50 };
  CUI.H3("Der Aufrufer hat vorher folgende Werte:");
  Console.WriteLine(a);
  Console.WriteLine(b);
  Console.WriteLine(c);
  Console.WriteLine(d);
  Console.WriteLine(e);
  string r = ParameterDemoRef1(a, b, ref c, ref d, out e);
  CUI.H3("Die Methode hat folgende Werte:");
  Console.WriteLine(r);
  CUI.H3("Der Aufrufer hat nachher folgende Werte:");
  Console.WriteLine(a);
  Console.WriteLine(b);
  Console.WriteLine(c);
  Console.WriteLine(d);
  Console.WriteLine(e);
}

public string ParameterDemoRef1(Counter WertValue, in Counter WertIn, ref Counter
WertRef, ref readonly Counter WertRefRO, out Counter WertOut)
 {
  WertValue.Value++;
```

```
  WertIn.Value++;
  WertRef.Value++;
  WertRefRO.Value++;
  WertOut = new Counter { Name = "d", Value = 41 };
  return WertValue.ToString() + ";" + WertRef.ToString() + ";" +
WertIn.ToString() + ";" + WertOut.ToString();
 }
```

```
ParameterReferenceType1
Der Aufrufer hat vorher folgende Werte:
a=10
b=20
c=30
d=40
e=50
Die Methode hat folgende Werte:
a=11;c=31;b=21;d=41
Der Aufrufer hat nachher folgende Werte:
a=11
b=21
c=31
d=41
d=41
```

Abbildung: Ausgabe des vorherigen Listings

Listing: Parameter ist Referenztyp (class Counter). Methode ändert Objektreferenz

```
class Counter
{
 public string Name { get; set; }
 public int Value { get; set; }
 public override string ToString() => Name + "=" + Value;
}

/// <summary>
/// Referenztypen an Methode übergeben, die neues Objekt zuweist ändert
/// </summary>
public void ParameterReferenceType2()
 {
  CUI.H2(nameof(ParameterReferenceType2));
  Counter a = new Counter() { Name = "a", Value = 10 };
  Counter b = new Counter() { Name = "b", Value = 20 };
  Counter c = new Counter() { Name = "c", Value = 30 };
  Counter d = new Counter() { Name = "d", Value = 40 };
  Counter e = new Counter() { Name = "e", Value = 50 };
  CUI.H3("Der Aufrufer hat vorher folgende Werte:");
  Console.WriteLine(a);
  Console.WriteLine(b);
  Console.WriteLine(c);
```

```
  Console.WriteLine(d);
  Console.WriteLine(e);
  string r = ParameterDemoRef2(a, b, ref c, ref d, out e);
  CUI.H3("Die Methode hat folgende Werte:");
  Console.WriteLine(r);
  CUI.H3("Der Aufrufer hat nachher folgende Werte:");
  Console.WriteLine(a);
  Console.WriteLine(b);
  Console.WriteLine(c);
  Console.WriteLine(d);
  Console.WriteLine(e);
}

public string ParameterDemoRef2(Counter WertValue, in Counter WertIn, ref Counter
WertRef, ref readonly Counter WertRefRO, out Counter WertOut)
{
  WertValue = new Counter { Name = "a*", Value = 101 };
  // WertIn = new Counter { Name = "b*", Value = 100 }; // nicht erlaubt
  WertRef = new Counter { Name = "c*", Value = 102 };
  // WertRefRO = new Counter { Name = "c*", Value = 103 }; // nicht erlaubt, da
readonly
  WertOut = new Counter { Name = "d*", Value = 104 };
  return WertValue.ToString() + ";" + WertRef.ToString() + ";" +
WertIn.ToString() + ";" + WertOut.ToString();
 }
```

```
ParameterReferenceType2
Der Aufrufer hat vorher folgende Werte:
a=10
b=20
c=30
d=40
e=50
Die Methode hat folgende Werte:
a*=101;c*=102;b=20;d*=104
Der Aufrufer hat nachher folgende Werte:
a=10
b=20
c*=102
d=40
d*=104
```

Abbildung: Ausgabe des vorherigen Listings

Für die Deklaration von out-Variablen gibt es seit C# 7.0 eine verkürzte Syntax, bei der die Deklaration der Variablen im Aufruf selbst erfolgt (siehe folgendes Listing).

Auch neu in C# 7.0 ist das Konstrukt out _. Der Unterstrich ist die Discard-Variable und bedeutet, dass das Ergebnis verworfen wird.

Listing: In den Aufruf eingebettete Deklaration von out-Variablen

```
// alt
int zahl;
string eingabe = "123";
if (int.TryParse(eingabe, out zahl))
    Console.WriteLine("Zahl=" + zahl);
else
    Console.WriteLine("Fehler!");

// neu
string eingabe2 = "123";
if (int.TryParse(eingabe2, out int zahl2))
    Console.WriteLine("Zahl=" + zahl2);
else
    Console.WriteLine("Fehler!");

// neu: _ = Wert ignorieren
string eingabe3 = "123";
if (int.TryParse(eingabe3, out _))
    Console.WriteLine("Ist eine Zahl!");
```

16.7 Parameterlisten

Seit der ersten Version von C# gibt es Parameter-Arrays für sogenannte variadische Parameter (vgl. *https://de.wikipedia.org/wiki/Variadische_Funktion*), mit denen eine Methode eine beliebig lange Liste von Parametern eines Typs empfangen kann, wenn dies mit dem Schlüsselwort params eingeleitet wird.

Beispiel:

```
public void MethodeMitBeliebigVielenParametern_Alt(string text, params int[] args
)
{
 CUI.H2(nameof(MethodeMitBeliebigVielenParametern_Alt));
 CUI.Print(text + ": " + args.Length);
 foreach (var item in args)
 {
  CUI.LI(item);
 }
}
```

Diese Methode kann man beispielsweise so aufrufen:

```
MethodeMitbeliebigVielenParametern_Alt("Anzahl Zahlen", 1, 2, 3);
MethodeMitbeliebigVielenParametern_Alt("Number of numbers", 1, 2, 3, 4);
```

Neu seit C# 13.0 ist, dass statt eines Arrays bei den Parametern auch generische Mengentypen verwendet werden dürfen, z.B. List<T>:

```
public void MethodeMitBeliebigVielenParametern_Neu(string text, params List<int>
args)
{
 CUI.H2(nameof(MethodeMitBeliebigVielenParametern_Neu));
 CUI.Print(text + ": " + args.Count);  // statt args.Length
 foreach (var item in args)
 {
  CUI.LI(item);
```

```
 }
}
```

Analog ist der Aufruf dann genauso flexibel möglich wie beim Parameter-Array:

```
MethodeMitBeliebigVielenParametern_Neu("Anzahl Zahlen", 1, 2, 3);
MethodeMitBeliebigVielenParametern_Neu("Number of numbers", 1, 2, 3, 4);
```

Dann sind diese generischen Mengentypen bei params in C# 13.0 erlaubt:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IReadOnlyCollection<T>
- System.Collections.Generic.IReadOnlyList<T>
- System.Collections.Generic.ICollection<T>
- System.Collections.Generic.IList<T>
- Alle Klassen, die System.Collections.Generic.IEnumerable<T> implementieren
- System.Span<T>
- System.ReadOnlySpan<T>

16.8 Statische Methoden als globale Funktionen

In C# 6.0 hat Microsoft eingeführt, was in Visual Basic .NET schon seit der ersten Version möglich ist: statische Klassen mit using so einzubinden, dass man auf die einzelnen Klassenmitglieder nun ohne Verwendung des Klassennamens zugreifen darf:

```
// bisherige Schreibweise
 Console.WriteLine(Environment.UserDomainName + @"\" + Environment.UserName);

 // neu seit C# 6.0
using static System.Console;
using static System.Environment;
…
 WriteLine(UserDomainName + @"\" + UserName);
```

Dieses Sprachfeature ist jedoch umstritten, weil hier die Lesbarkeit des Programmcodes zugunsten einer ersparten Tipparbeit geopfert wird.

16.9 Lokale Funktion (seit C# 7.0)

C# 7.0 unterstützt lokale Funktionen, die in andere Methoden eingebettet und nur dort sichtbar sind. Lokale Funktionen können über mehrere Ebenen geschachtelt sein und die Variablen der äußeren Ebenen (der umgebende Klasse und Funktion) verwenden (siehe folgendes Listing). Solch ein Einbetten ist auch in Getter- und Setter-Routinen erlaubt.

Hinweis: Seit C# 9.0 können lokale Funktionen auch Annotation mit .NET-Attributen besitzen, z.B. [Obsolete].

Listing: Eingebettete Funktionen haben Zugriff auf die Variablen der äußeren Funktionen.

```
  public static void LocalFunctionDemo()
  {
   var count = 0;
   CUI.Headline(nameof(LocalFunctionDemo));

   PrintWithTime("Rom");
```

```
   PrintWithTime("Paris");
   PrintWithTime("Essen");

   // Funktion ist Teil der Funktion, möglich in Methoden, Getter und Setter
   void PrintWithTime(string s)
   {
    void Print(string s2)
    {
     // innere Funktion kann Variablen der äußeren nutzen
     count++;
     Console.WriteLine(count + ": " + s2);
    }
    Print($"{DateTime.Now.ToShortTimeString()}: {s}");
    }
  }
```

16.10 Statische lokale Funktionen (seit C# 8.0)

Die in C# 8.0 neu eingeführten statischen lokalen Funktionen können im Gegensatz zu den in C# 7.0 eingeführten nicht-statischen lokalen Funktionen NICHT auf Variablen der äußeren Ebenen (der umgebende Klasse und Funktion) zugreifen.

Listing: Eingebettete statische Funktionen haben keinen Zugriff auf die Variablen der äußeren Funktionen.

```
using System;

namespace CS80
{
 class StaticLocalFunctionsDemo
 {
  int field = 42;
  public int prop { get; set; } = 42;
  public void Run()
  {
   int x = 42;

   NonStaticLocalFunc(x);
   StaticLocalFunc(x);

   // seit C# 7.0: Nicht-
statische lokale Funktion kann umgebende Variablen nutzen!
   int NonStaticLocalFunc(int p)
   {
    int y = 42;
    int x = 43;                // verdeckt x aus Run()
    Console.WriteLine(x);      // OK
    Console.WriteLine(prop);   // OK
    Console.WriteLine(field);  // OK
    Console.WriteLine(y);      // OK
    return p;
   }

   // ----> seit C# 8.0: Kann umgebende Variablen NICHT sehen!
```

```
   static int StaticLocalFunc(int p)
   {
    int y = 42;
    int x = 43;                  // verdeckt x aus Run()
    Console.WriteLine(x);        // lokales x
    //Console.WriteLine(field);  // nicht erlaubt, weil static
    //Console.WriteLine(prop);   // nicht erlaubt, weil static
    Console.WriteLine(y);        // Ok, weil lokal
    return p;
   } // Ende der statischen lokalen Funktion
  } // Ende der Methode Run()
 } // Ende der Klasse
}
```

16.11 Caller-Info-Annotationen

Seit Version C# 5.0 (auch in Visual Basic .NET seit Version 11.0) bieten die Compiler sogenannte Caller-Info.Annotationen

- [CallerFilePath]
- [CallerLineNumber]
- [CallerMemberName]

mit denen man Methodenparameter annotieren kann. Dadurch erhält die gerufene Methode Informationen über den Aufrufer (vgl. __FILE__ und __LINE__ in C++).

Listing: Nutzung der Caller-Info-Annotationen

```
public void Run()
{
 var Ergebnis = Berechnen(10);
 Console.WriteLine("Berechnungsergebnis: " + Ergebnis);
 Run2();
}

public int Berechnen(int Wert,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string filePath = "",
      [CallerLineNumber] int lineNumber = 0)
{
 // Ausgabe hier zu Anschauungszwecken an der Konsole
 Console.ForegroundColor = ConsoleColor.Yellow;
 Console.WriteLine("Routine Berechnung() wurde aufgerufen!");
 Console.WriteLine("Aus diesem Quellcodepfad: " + filePath);
 Console.WriteLine("Von diesem Mitglied: " + memberName);
 Console.WriteLine("In dieser Zeilennummer: " + lineNumber);
 Console.ForegroundColor = ConsoleColor.Gray;
 // Eigentlicher Inhalt der Berechnung
 Console.WriteLine("Hier tue ich was...");
 return 10 * Wert;
}
```

```
CallerInfoDemo
Routine Berechnung() wurde aufgerufen!
Aus diesem Quellcodepfad: H:\TFS\Demos\NET\CSharpSprachsyntax\CSharpSprachsyntax\CS50_NET45_2012\Cal
lerInfo.cs
Von diesem Mitglied: Run
In dieser Zeilennummer: 12
Hier tue ich was...
Berechnungsergebnis: 100
```

Abbildung: Ausgabe des obigen Listings

Insbesondere [CallerMemberName] ist sehr hilfreich, um die Schnittstelle INotifyPropertyChanged zu realisieren, die einige GUI-Frameworks (z.B. Windows Forms, WPF) in .NET für Datenbindungsmechanismen erfordern. Ohne [CallerMemberName] müsste man beim Aufruf NotifyPropertyChanged() den Namen des Properties manuell als Zeichenkette übergeben: NotifyPropertyChanged("Wert"), was fehleranfällig ist. Erst seit C# 7.0 kann man auch schreiben: NotifyPropertyChanged(nameof(Wert)), was aber immer noch mehr Tipparbeit ist als der Einsatz von [CallerMemberName].

Listing: Elegante Realisierung von INotifyPropertyChanged mit [CallerMemberName]

```
class DatenobjektDemo
 {
  public static void Run()
  {
   CUI.Headline(nameof(DatenobjektDemo));
   var d = new Datenobjekt();
   d.PropertyChanged += (x, args) =>
   {
    Console.WriteLine("DatenobjektDemo: Property " + args.PropertyName + " hat
sich geändert!");
   };
   d.Wert = 123;
  }
 }

 class Datenobjekt : System.ComponentModel.INotifyPropertyChanged
 {
  public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

  /// <summary>
  /// Realisierung mit expliziter Übergabe des Property-Namens
  /// </summary>
  /// <param name="propertyName"></param>
  private void NotifyPropertyChangedAlt(String propertyName = "")
  {
   Console.WriteLine("Datenobjekt: Property " + propertyName + " hat sich
geändert!");
   if (PropertyChanged != null)
   {
    PropertyChanged(this, new
System.ComponentModel.PropertyChangedEventArgs(propertyName));
   }
  }

  /// <summary>
  /// Realisierung ohne dass der Aufrufer den Property-Namen übergeben muss
  /// </summary>
  /// <param name="propertyName"></param>
  private void NotifyPropertyChanged([CallerMemberName] String propertyName = "")
```

```
  {
   Console.WriteLine("Datenobjekt: Property " + propertyName + " hat sich
geändert!");
   if (PropertyChanged != null)
   {
    PropertyChanged(this, new
System.ComponentModel.PropertyChangedEventArgs(propertyName));
   }
  }

  private int wert;

  public int Wert
  {
   get { return wert; }
   set { wert = value; NotifyPropertyChanged(); }
  }
 }
```

16.12 Caller Argument Expressions

Zusätzlich zu schon in C# 5.0 eingeführten Caller-Info-Annotationen [CallerFilePath], [CallerLineNumber] und [CallerMemberName] gibt es seit C# 10.0 nun auch Caller Argument Expressions, mit denen eine Methode die Information erhält, welche Ausdrücke (Variablennamen bzw. Formeln) hinter den vom Aufrufer übergebenen Werten stehen.

Dafür kann der Entwickler in der Parameterliste die Annotation System.Runtime.CompilerServices. CallerArgumentExpressionAttribute einsetzen, die es seit .NET Core 3.0 gibt. Die folgende Methode besitzt sechs Parameter: drei "echte Parameter" und drei Caller Argument Expressions für die ersten drei Parameter. Die Caller Argument Expressions beziehen sich auf den Namen der Parameter.

Hinweis: Leider muss man die Parameter als Zeichenkette angeben: der Operator nameof() funktioniert hier nicht.

Listing: Einsatz von Caller Argument Expressions

```
public static class Validation
{
 public static void CheckRange(int value, int minValue, int maxValue,
 [CallerArgumentExpression("value")] string? valueExpression = null,
 [CallerArgumentExpression("minValue")] string? minValueExpression = null,
 [CallerArgumentExpression("maxValue")] string? maxValueExpression = null)
 {
  if (value > maxValue)
  {
   throw new ArgumentOutOfRangeException(nameof(value),
     $"{value} ({valueExpression}) muss zwischen {minValue}
({minValueExpression}) und {maxValue} ({maxValueExpression}) liegen!");

  }
 }
}
```

Beim Aufruf der Methode werden nur die ersten drei Parameter erwartet:

```
var a = 5;
var max = Convert.ToInt32(Math.Floor(Math.PI));
Validation.CheckRange(a * 2, 0, max);
```

Die übrigen drei füllt der Compiler automatisch. Der folgende Screenshot zeigt, dass die Methode für den ersten Parameter die Information erhält, dass sich 10 aus a * 2 zusammensetzt. Der zweite Parameter war ein Zahlenliteral (0), der dritte Parameter eine Variable.

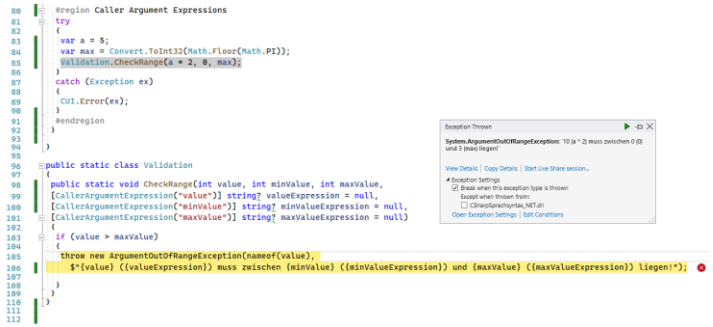
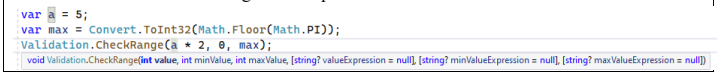


Abbildung: Wirkung der Caller Argument Expressions

Hinweis: Die Caller Argument Expressions können bei der Protokollierung und in Fehlermeldungen helfen. Sie bergen aber auch die Gefahr, dass fremder Programmcode an Informationen (Variablenname oder Formeln) kommt, die er nicht bekommen sollte. Daher sieht der Aufrufer die Caller Argument Expressions im Editor.



17 Konstruktoren und Destruktoren (Finalizer)

Ein Konstruktor ist eine Methode, die beim Instanziieren einer .NET-Klasse aufgerufen wird. In ihm kann man das Objekt initialisieren. Ein Desktruktor wird bei der Vernichtung eines Objekts aufgerufen.

Konstruktoren besitzen den Namen der Klasse und haben keinen Rückgabetyp (auch nicht void). Der Bezeichner für den Finalizer besteht aus ~, gefolgt vom Klassennamen. Es kann nur höchstens einen Finalizer geben, aber beliebig viele überladene Konstruktoren. Diese können sich gegenseitig mit dem : this() aufrufen (ggf. unter Angabe der Parameter). Das : this() muss vor der öffnenden geschweiften Klammer stehen.

Echte Destruktoren, die beim Löschen eines Objekts aufgerufen werden, kennt .NET hingegen nicht. Der Aufruf des Destruktors ist in .NET nicht deterministisch, weil er erst bei einer Speicherbereinigung (Garbage Collection) erfolgt oder ggf. ganz ausbleibt, wenn das Programm vorher endet. Daher spricht man oft auch von **Finalizern** statt von Destruktoren.

17.1 Klasse mit Konstruktoren und Finalizer

Die Klasse im folgenden Listing besitzt drei überladene Konstruktoren und einen Finalizer. Die Konstruktoren rufen sich gegenseitig auf. Im parameterlosen Konstruktor wird das private statische Attribut Count hochgezählt, sodass jede Instanz innerhalb eines Programmlaufs eine eindeutige ID erhält.

Listing: Klasse mit Konstruktoren und Finalizer

```
/// <summary>
/// Klasse mit Konstruktoren und Finalizer
/// </summary>
class Dozent
 {
  private static int Count = 0;

  // Konstruktor mit einem Parameter
  public Dozent(string Name) : this(Name, null)
  {
  }

  // Weiterer Konstruktor mit zwei Parametern
  public Dozent(string name, string themen) : this()
  {
   this.Name = name;
   this.Themen = themen;
  }

  // Konstruktor ohne Parameter
  public Dozent()
  {
   Count++;
   this.ID = Dozent.Count;
   CUI.Print("Dozent #" + this.ID + " wurde instanziiert!", ConsoleColor.Cyan);
  }

  // Finalizer
  ~Dozent()
  {
   CUI.Print("Dozent #" + this.ID + " wurde vernichtet!", ConsoleColor.Cyan);
  }
```

```
  // Automatisches Property
  public int ID { get; set; }
  // Automatisches Property
  public string Themen { get; set; }

  // Property mit explizitem Field
  string name;
  public string Name
  {
   get { return name; }
   set { name = value; }
  }
 }
```

Achtung: Ein parameterloser Konstruktor, der nichts tut, scheint auf den ersten Blick überflüssig zu sein. Sofern kein parameterbehafteter Konstruktor vorhanden ist, generiert der Compiler – sowohl von C# als auch von Visual Basic .NET – automatisch einen parameterlosen Konstruktor. Wird jedoch ein parameterbehafteter Konstruktor explizit implementiert, so wird der parameterlose Konstruktor nicht automatisch erzeugt. Wenn dieser benötigt wird, ist er also ebenfalls explizit zu implementieren.

Wie in Visual Basic .NET wird der parameterlose Konstruktor in C# nur dann automatisch erzeugt, wenn kein anderer Konstruktor explizit implementiert wird.

17.2 Aufruf von Konstruktoren

Der folgende Programmcode nutzt obige Klasse Dozent, indem er eine Instanz erzeugt und verwendet. Nach der Verwendung wird die Objektvariable auf null gesetzt, d.h. es gibt nun keinen Verweis mehr auf die Instanz. Der Garbage Collector von .NET wird bei nächster Speicherbereinigung den Finalizer aufrufen. In diesem Fall wird zu Demonstrationszwecken die Garbage Collection mit dem Aufruf System.GC.Collect() erzwungen. Die Garbage Collection läuft aber asynchron in einem Hintergrundthread, d.h. die nach Collect() folgenden Befehle werden vor der Garbage Collection ausgeführt wie man in der folgenden Abbildung erkennen kann, dass die Ausgabe "Routine fertig" vor der Ausgabe des Finalizers erscheint.

Listing: Nutzung der Klasse Dozent

```
   CUI.Headline("Beispiel für Konstruktur und Destruktor");

   Console.WriteLine("Dozent wird erzeugt...");
   var d = new Dozent("Holger Schwichtenberg", ".NET, PowerShell, JavaScript");

   Console.WriteLine("Dozent wird verwendet...");
   d.Themen += ", C#, TypeScript, Entity Framework, ASP.NET";
   Console.WriteLine("Dozent " + d.ID + " (" + d.Name + ")  hat folgende Themen:
");
   foreach (string t in d.Themen.Split(','))
   {
    Console.WriteLine("- " + t.Trim());
   }

   Console.WriteLine("Dozent wird nicht mehr benötigt...");
   d = null;

   Console.WriteLine("Garbage Collection wird erzwungen...");
   System.GC.Collect(); // läuft asynchon
   Console.WriteLine("Routine fertig!");
   Console.ReadLine();
```

```
Beispiel für Konstruktur und Destruktor
Dozent wird erzeugt...
Dozent #1 wurde instanziiert!
Dozent wird verwendet...
Dozent 1 (Holger Schwichtenberg)  hat folgende Themen:
- .NET
- PowerShell
- JavaScript
- C#
- TypeScript
- Entity Framework
- ASP.NET
Dozent wird nicht mehr benötigt...
Garbage Collection wird erzwungen...
Routine fertig!
Dozent #1 wurde vernichtet!
```

Abbildung: Ausgabe des obigen Listings

Info: Die Laufzeitumgebung Common Language Runtime (CLR) von .NET (alle Varianten) enthält einen Garbage Collector (GC), der im Hintergrund (in einem System-Thread) arbeitet und den Speicher aufräumt. Der Speicher wird allerdings nicht sofort nach dem Ende der Verwendung eines Objekts freigegeben, sondern zu einem nicht festgelegten Zeitpunkt bei Bedarf (Lazy Resource Recovery). Beim Aufräumen des Speichers erzeugt der Garbage Collector einen Baum aller Objekte, auf die es aktuell einen Objektverweis gibt. Der Speicher aller nicht mehr erreichbaren Objekte wird freigegeben.

Der Garbage Collector kann von einer Anwendung nur bedingt beeinflusst werden. Die Anwendung kann mit dem Befehl System.GC.Collect() dem Garbage Collector den Auftrag geben, tätig zu werden. Eine Anwendung eine Speicherbereinigung temporär mit GC.TryStartNoGCRegion() unterdrücken.

Der Garbage Collector ruft die Destruktoren (alias Finalizer) der .NET-Objekte auf. Die Reihenfolge des Aufrufs und ob der Finalizer überhaupt aufgerufen wird, ist jedoch nicht deterministisch, d. h., es kann sein, dass ein Finalizer nicht aufgerufen wird. Beim Schließen einer .NET-Anwendung werden die Finalizer der verbliebenen Objekte nicht aufgerufen.

17.3 Primärkonstruktoren (seit C# 12.0)

Die bedeutendste Neuerung in C# 12.0 sind Primärkonstruktoren für Klassen. Alte Hasen unter den C#-Entwicklern werden sich erinnern, dass dieses Sprachfeature bereits im Jahr 2014 als Prototyp für C# 6.0 verfügbar war, dann aber doch gestrichen wurde *www.heise.de/developer/artikel/Microsoft-streicht-Sprachfeatures-aus-C-6-0-und-Visual-Basic-2015-2432073.html*].

Nun, sechs C#-Versionen weiter, kommt Microsoft in C# 12.0 darauf zurück, auch vor dem Hintergrund der Record-Typen, die es seit C# 9.0 mit Primärkonstruktoren gibt:

```
public record Person(int ID, string Name, string Website = "");
```

Ein Primärkonstruktor ist eine Parameterliste direkt hinter dem Typnamen. Seit C# 12.0 ist das auch für Klassendefinitionen möglich:

```
public class Person(int ID, string Name, string Website = "");
```

Solch eine Klasse kann ohne Inhaltsbereich (also geschweifte Klammern) existieren, ist aber wertlos. Anders als bei den in C# 9.0 eingeführten Record-Typen erstellt der Primärkonstruktor nämlich keine öffentlichen Properties in der Klasse, sondern nur private Fields. Wenn man diese Klasse mit Primärkonstruktor in einem Decompiler betrachtet, sieht man zunächst überhaupt keine Verarbeitung der Parameter im Primärkonstruktor:

```
public class Person
{
    public Person(int ID, string Name, string Website = "")
    {
    }
}
```

Das liegt daran, dass die Primärkonstruktorparameter gar nicht verwendet werden. Wir müssen die Klasse z.B. um ToString() erweitern, siehe Listing.

Listing: Klasse mit Primärkonstruktor und Methode ToString()

```
public class Person(int id, string name, string Website = "")
{
 public string Name { get; set; } = name;
 public string Website { get; set; } = website;

 public override string ToString()
 {
  return $"Person #{ID}: {Name} -> {Website}";
 }
}
```

Nun sehen wir im Decompiler, dass ein privates Feld für den Konstruktorparameter id entstanden ist, aber nicht für name und website, da mit diesen lediglich ein Property initialisiert wurde und kein direkter Zugriff mehr auf die Namen aus dem Primärkonstruktor (name und website mit kleinem Anfangsbuchstaben!) erfolgt.

Hinweis: Es entsteht kein privates Field, wenn man einen Konstruktorparameter nur für eine Initialisierung verwendet!

Listing: Dekompilat des vorherigen Listings mit ILSpy

```
public class Person
{
    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private int <id>P;

    public string Name { get; set; }

    public string Website { get; set; }

    public Person(int ID, string name, string website = "")
    {
        <id>P = ID;
        Name = name;
        Website = website;
        base..ctor();
    }

    public override string ToString()
    {
        return $"Person #{<id>P}: {Name} -> {Website}";
    }
}
```

Um öffentlich auf die im Primärkonstruktor übergebenen Daten zugreifen zu können, muss man die Konstruktorparameter für Zuweisungen verwenden, siehe Name und Website im nächsten Listing.

Zu beachten ist, dass Entwicklerinnen und Entwickler nun in der Implementierung von ToString() auf das Property Name und nicht mehr auf den Primärkonstruktorparameter name zugreifen

sollten, denn sonst würde man nachträgliche Namensänderungen (Zuweisungen an das Property Name) nicht bei ToString(). Der C#-Compiler denkt mit und wirft in Fall der Verwendung von name bei ToString() die Warning "CS9124" aus: Parameter 'string name' is captured into the state of the enclosing type and its value is also used to initialize a field, property, or event.". Diese Fehlermeldung gibt es aber nicht bei der Verwendung in ToString(), sondern bei der Initialisierung des Properties:

```
public string Name { get; set; } = name;
```

Auch abgeleitete Klassen dürfen Primärkonstruktoren besitzen. Im nächsten Listing gibt es neben der Klasse Person eine zweite, abgeleitete Klasse Autor mit Primärkonstruktor.

Listing: Primärkonstruktorbeispiel mit und ohne Zuweisung der Primärkonstruktorparameter an öffentliche Properties und Vererbung

```
namespace NET8Konsole.CS12;

/// <summary>
/// Klasse mit Primärkonstruktor
/// </summary>
public class Person(Guid id, string name)
{
 public string Name { get; set; } = name;
 public Person() : this(Guid.Empty, "") {   }
 public override string ToString()
 {
  // Hier Property Name statt Primärkonstruktorparameter name verwenden!
  // Man würde sonst Namensänderungen nicht sehen!
  return $"Person {id}: {Name}";
 }
}

/// <summary>
/// Abgeleitete Klasse mit Primärkonstruktor
/// </summary>
public class Autor(Guid id, string name, string website) : Person(id, name)
{
 public string Website { get; set; } = website;


 public override string ToString() {
  return $"Autor {id}: {Name} -> {Website}";
 }
}

internal class CS12_PrimaryConstructors_Demo
{
 public void Run()
 {
  var p = new Person();
  Console.WriteLine(p.Name);
  Console.WriteLine(p.ToString());
  var a = new Autor(Guid.NewGuid(), "Dr. Holger Schwichtenberg", "www.IT-
Visions.de");
  Console.WriteLine(a.Name);
```

```
  Console.WriteLine(a.Website);
  Console.WriteLine(a.ToString());
 }
}
```

Hinweis: Leider gibt es in C# in Primärkonstruktoren nicht wie TypeScript-Konstruktoren die Möglichkeit, durch die Sichtbarkeiten public und private zu steuern (vgl. *https://kendaleiv.com/typescript-constructor-assignment-public-and-private-keywords/*), welche Sichtbarkeit die resultierenden Datenmitglieder der Klasse erhalten sollen. Ebenso ist keine Einschränkung readonly möglich, die verhindert, dass Programmcode in der Klasse den übergebenen Wert verändert.

18 Aufzählungstypen (Enumeration)

Ein Aufzählungstyp legt unter einem Oberbegriff mehrere Namen fest. Den Namen werden intern Zahlen zugeordnet.

```
public enum Kenntnisse
{
Befriedigend=3,Gut=2,SehrGut=1
}
```

Wenn keine Zahlen im der Typdefinition benannt sind, beginnt die Zählung automatisch bei 0, was in diesem Beispiel nicht so viel Sinn machen würde, in anderen Fällen können die Werte aber aus Entwicklersicht irrelevant sein.

```
public enum Kenntnisse
{
Befriedigend,Gut,SehrGut
}
```

Das folgende Listing zeigt die Verwendung dieses Aufzählungstypen inklusive der Umwandlung zwischen Aufzählungswertname und dem Zahlenwert.

```
   Kenntnisse meineCSharpKenntnisse = Kenntnisse.SehrGut;

   // Umwandlung Aufzählungswert in Zahl
   int note = (int)meineCSharpKenntnisse; // = 1

   Console.WriteLine($"Meine C#-Kenntnisse sind {meineCSharpKenntnisse}, in
Noten: {note}!"); // "SehrGut" 1

   // Umwandlung Zahl in Aufzählungswerz
   Kenntnisse noteAlsText = (Kenntnisse) note; // wandelt 1 in Kenntnisse.SehrGut

   if (noteAlsText == Kenntnisse.SehrGut) { Console.WriteLine("Meine Kenntnisse
sind weiterhin sehr gut!"); };

   switch (noteAlsText)
   {
    case Kenntnisse.Befriedigend:
     Console.WriteLine("Meine Kenntnisse sind noch befriedigend"); break;
    case Kenntnisse.Gut:
     Console.WriteLine("Meine Kenntnisse sind immer noch gut!"); break;
    case Kenntnisse.SehrGut:
     Console.WriteLine("Meine Kenntnisse sind immer noch sehr gut!"); break;
   }
```

Hinweis: Weder C#-Compiler noch Laufzeitumgebung beschweren sich, wenn man zahlen in einem Enumerationswert konvertiert, die es nicht gibt. Beispiel: Kenntnisse unsinnigeNote = (Kenntnisse)42;

Nun liefert ein Zugriff auf unsinnigeNote den Wert 42.

19 Expression-bodied Members

Expression-bodied Members sind neu seit C# 6.0 – es gibt sie nicht in Visual Basic .NET. Methoden und nicht beschreibbare Properties, die nur einen einzigen Ausdruck zurückliefern, kann der C#-Entwickler nun verkürzt unter Einsatz des Lambda-Operators => einen sogenannten **Expression Body** statt eines Blocks in geweiften Klammern (Block Body) schreiben:

```
public string GanzerName => this.Vorname + " " + this.Nachname;
public decimal NeuerEinkauf(decimal wert) => this.Umsatz += wert;
public override string ToString() => this.GanzerName + ": " + this.KontaktStatus;
```

Mit C# 6.0 hatte Microsoft sogenannte "Expression-bodied Members" eingeführt, die bei einzeiligen Methoden und read-only Properties eine verkürzte Lambda-Schreibweise erlauben. Seit C# 7.0 ist dies nun ausgeweitet auf Konstruktoren, Finalizer sowie Getter-, Setter- und Indexer-Routinen. Seit C# 8.0 sind Expression Bodies genauso wie Block Bodies auch in Standardimplementierungen in Schnittstellen (Interfaces) erlaubt.

```
class Dozent
  {
   public int ID { get; set; }
   public string Name { get; set; }
   public bool DOTNETExperte { get; set; }

   public Dozent()  {   }

   // Expression-bodied Constructor
   public Dozent(int ID) => this.ID = ID;

   // Expression-bodied Finalizer
   ~Dozent() => Console.Error.WriteLine("Finalized!");
   // Expression-bodied Getter und Setter
   private Decimal? honorar2;
   public Decimal? Honorar2
   {
    get => this.honorar;
    set => this.honorar = value ?? 1000.00m;
   }

}
```

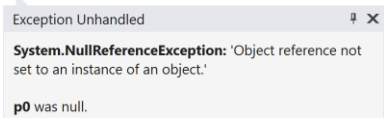
20 Behandlung von null

Zu den häufigsten Fehlern, die Softwareentwickler in C# machen, zählt die fehlende Berücksichtigung, dass Variablen und Klassenattribute den Wert null annehmen können und man auf einem null-Wert keine Objektoperationen ausführen kann. Dann kommt es zum Laufzeitfehler NullReferenceException.

20.1 NullReferenceException

Eine NullReferenceException entsteht sofort, wenn man von einer Objektvariable, die null ist, einen Attributwert abrufen will, einen Attributwert setzen will oder eine Methode aufrufen will:

```
// das führt zum Absturz
Person p0 = null;
Console.WriteLine(p0.Nachname);
```

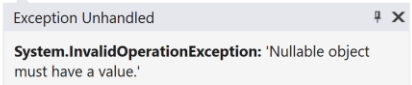


Auch ein Aufruf einer Methode auf einer Zeichenkette, die null ist, führt zur NullReferenceException:

```
string eingabe = null;
string eingabeInKleinbuchstaben = eingabe.ToLower();
```

Auch Wertetypen können null sein, wenn man sie als Nullable Value Type (NVT) in die Datenstruktur Nullable<T> verpackt. Hier kommt es beim Versuch, eine Rechenoperation auf einem null-Wert auszuführen zum Laufzeitfehler: System.InvalidOperationException: 'Nullable object must have a value.'.

```
int? zahl = null;
int zahl2 = zahl.Value+10;
```



20.2 Null-Prüfung und Toleranz gegenüber Null

Zur Vermeidung der NullReferenceException ist es wichtig, **immer** vor dem Zugriff auf ein Attribut oder auf eine Methode bzw. vor einer Rechenoperation mit einer Variablen, die null annehmen kann, sicherzustellen, dass die Variable auf ein Objekt verweist und nicht null ist:

```
// hier kann man sich NICHT sicher sein, dass p nicht null ist
Person p = GetPerson(123);
if (p != null)
{
    Console.WriteLine(p2.Nachname);
}
```

Dies gilt auch für die Weiterverarbeitung einzelner Attribute der Klasse. Angenommen, die Klassendefinition sei:

```
public class Person
 {
  public int ID { get; set; }
  public string Vorname { get; set; }
  public string Nachname { get; set; }
  public string Ort { get; set; }
  public DateTime Geburtstag { get; set; }
  public DateTime? Einstellungsdatum { get; set; }
  public decimal Gehalt { get; set; }

  public Person(int id)
  {
   this.ID = id;
  }
 }
```

Hier ist zu beachten, dass Geburtstag ein normaler Wertetyp ist (also nicht null annehmen kann), aber Einstellungsdatum ein Nullable Value Type (NVT) ist. Geburtstag wird im Standard mit dem 1.1.0001 initialisiert, aber das Einstellungsdatum mit null. Der Abruf des Attributs Year aus dem Geburtstag ist daher eine sichere Operation, das gleiche auf Einstellungsdatum kann aber zur NullReferenceException führen.

Das folgende Listing zeigt fünf Optionen der Behandlung des null-Falls:

- Prüfung mit == null
- Prüfung mit is null
- Weiterreichen des null-Wertes mit ?. (Null-propagating Operator)
- Umwandeln des null-Wertes in einen anderen Wert der gleichen Klasse mit ??
- Umwandeln des null-Wertes in einen beliebigen anderen Wert mit ? … : …

```
Person p2 = GetPerson(123);
if (p2 != null)
   {
    Console.WriteLine(p2.Nachname);

    // Geburtstag ist DateTime, daher kein nicht null als Wert vorkommen
    Console.WriteLine("Geboren im Jahr: " + p2.Geburtstag.Year);

    // Einstellungsdatum ist aber Nullable<DateTime>, daher droht hier ein
Laufzeitfehler
    Console.WriteLine("Eingestellt im Jahr: " + p2.Einstellungsdatum.Value.Year);

    // Richtige Variante 1a mit null-Prüfung
    if (p2.Einstellungsdatum != null)
    {
     Console.WriteLine("Eingestellt im Jahr: " +
p2.Einstellungsdatum.Value.Year);
    }
    // Richtige Variante 1b mit null-Prüfung
    if (!(p2.Einstellungsdatum is null))
    {
```

```
     Console.WriteLine("Eingestellt im Jahr: " +
p2.Einstellungsdatum.Value.Year);
    }

    // Richtige Variante 2 mit ?.
    Console.WriteLine("Eingestellt im Jahr: " + p2.Einstellungsdatum?.Year);

    // Richtige Variante 3 mit ??
    Console.WriteLine("Eingestellt im Jahr: " + (p2.Einstellungsdatum ??
default(DateTime)));

    // Richtige Variante 4 mit ? :
    Console.WriteLine("Eingestellt im Jahr: " + (p2.Einstellungsdatum != null ?
p2.Einstellungsdatum.Value.ToString() : "Kein Datum"));
}
```

Eine weitere Behandlung des null-Falls ist in C# 8.0 hinzugekommen in Form des Operators "Null Coalescing Assignment" mit ??=. Mit diesem Zuweisungsoperator kann der C#-Softwareentwickler eine Zuweisung ausführen, wenn eine Variable den Wert null hat.

Statt

```
p = p ?? new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

oder

```
if (p == null) p = new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

kann man nun auch prägnanter schreiben:

```
p ??= new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

20.3 Null-Referenz-Prüfung / Non-Nullable Reference Types (C# 8.0)

Bereits im September 2017 [*www.heise.de/developer/meldung/Programmiersprachen-C-8-soll-Fehler-mit-null-verhindern-3835949.html*] hatte Microsoft für C# 8.0 angekündigt: Referenztypen sollen nicht mehr automatisch "nullable" sein; die Möglichkeit, den Wert null zuzuweisen soll der Entwickler explizit deklarieren müssen.

Nach einiger Diskussion hat sich Microsoft aber zunächst entschlossen, diese Neuerung nicht zum Standard, sondern zu einer Option des C#-Compilers zu machen. In den Projektvorlagen für neue .NET-Projekte ab .NET 6.0 bzw. Visual Studio 2022 sind die Nullable Reference Types im Standard aktiv!

Achtung: Die Namensgebung des in C# 8.0 eingeführten Features ist nicht glücklich gewählt von Microsoft. Microsoft nennt das Feature offiziell "Nullable Reference Types" und die Einstellung heißt *<nullable>enable</nullable>* bzw. *#nullable enable*. Allerdings waren Referenztypen schon vor C# 8.0 immer "nullable" und dies es auch in den aktuellen C#-Versionen im Standard immer noch – das steht auch im ersten Satz der Dokumentation (siehe Abbildung). "nullable enable" ist also sehr missverständlich, denn dies schaltet den Standard aus. Richtig ist, bei dem neuen Feature von "Non-Nullable Reference Types" zu sprechen, wie dieses Kapitel daher auch heißt.

Nullable reference types

11/10/2021 • 15 minutes to read • +10

Prior to C# 8.0, all reference types were nullable. *Nullable reference types* refers to a group of features introduced in C# 8.0 that you can use to minimize the likelihood that your code causes the runtime to throw System.NullReferenceException. *Nullable reference types* includes three features that help you avoid these exceptions, including the ability to explicitly mark a reference type as *nullable*:

- Improved static flow analysis that determines if a variable may be `null` before dereferencing it.
- Attributes that annotate APIs so that the flow analysis determines *null-state*.
- Variable annotations that developers use to explicitly declare the intended *null-state* for a variable.

Abbildung: Unglückliche Namensgebung für das neue Feature bei Microsoft [learn.microsoft.com/en-us/dotnet/csharp/nullable-references]

Praxishinweis: Die neue Null-Referenz-Prüfung des C# 8.0-Compilers ist ein sinnvolles Instrument, um Null-Referenz-Fehler zur Laufzeit zu verhindern. Die Aktivierung dieser neuen Prüfung für bestehenden Programmcode ist aber ein größeres Projekt, denn die meisten Softwareentwickler wird der C# 8.0-Compiler mit sehr vielen Warnungen konfrontieren. Es ist daher sinnvoll, dieses neue Konzept erstmal an einzelnen Bibliotheken oder Programmteilen zu erproben.

Seit .NET 6 aktiviert Microsoft in allen Projektvorlagen im Standard die Nicht-Nullbaren-Referenztypen, siehe Zeile 7 im nachstehender Bildschirmabbildung einer .csproj-Datei. Der Entwickler kann dies aber wieder auf "disable" setzen oder die Zeile einfach löschen.

CS10.csproj

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3    <PropertyGroup>
4      <OutputType>Exe</OutputType>
5      <TargetFramework>net6.0</TargetFramework>
6      <ImplicitUsings>enable</ImplicitUsings>
7      <Nullable>enable</Nullable>
8    </PropertyGroup>
9
10 </Project>
```

Abbildung: Eine C#-Projektdatei, die mit der .NET 6-Projektvorlage für Konsolenanwendungen erzeugt wurde und <Nullable>enable</Nullable> enthält

20.3.1 Neue Compiler-Features

Der C#-Compiler bringt seit der Sprachversion 8.0 zur Vermeidung der häufigen Null-Referenz-Laufzeitfehler (Null Reference Exception) drei neue sogenannte Kontexte mit sich. Ein Kontext ist ein Bereich im C#-Programmcode. Ein Kontext kann sich über einzelne Zeilen, ausgewählte Klassen oder auch das ganze Projekt erstrecken.

Bisher galt in C# der Standardkontext mit folgender Bedeutung für Variablen, Fields und Properties:

- Variablen, Fields und Properties, die mit Wertetypen (z.B. int, DateTime, bool) deklariert wurden, können im Standard nicht den Wert Null annehmen. Sie können seit C# 2.0 mit Nullable<T> (bzw. die äquivalent prägnantere Form mit Fragezeichen, z.B. int? oder bool?) "nullable" gemacht werden.
- Variablen, Fields und Properties, die mit Referenztypen (string und eigene Klassen)

deklariert wurden, können immer Null annehmen.

Die drei neuen Kontexte in C# 8.0 sind:

- **Nullable Warning Context:** Der Compiler warnt vor dem Auftreten von Null-Reference-Laufzeitfehlern bei allen Zugriffen auf Variablen, bei denen möglich / nicht sichergestellt ist, dass sie nicht null enthalten bzw. bei denen der null-Fall nicht abgefangen ist.
- **Nullable Annotation Context:** Referenztypen (string, eigene Klassen) sind im Standard nicht mehr nullable (fähig, den null-Wert anzunehmen). Wenn null-Werte explizit gewünscht sind, ist dies mit dem Fragezeichen bei der Typdeklaration anzuzeigen, z.B. string? und Klasse? (Nicht aber erlaubt: Nullable<string> und Nullable<Klasse> wie bei den Nullable Value Types!)
- **Nullable Context:** Allgemein als "Nullable Context" wird ein Kontext bezeichnet, der sowohl Nullable Warning Context als auch Nullable Annotation Context ist, also die Funktionen beider Kontexte in sich vereint.

Hinweis: Ein Kontext ist ein Bereich in Ihrem Programmcode. Ein Kontext kann sich über einzelne Zeilen, ausgewählte Klassen oder auch das ganze Projekt erstrecken.

Die folgende Tabelle stellt die drei Kontextarten gegenüber.

	Nullable Annotation Context	**Nullable Warning Context**	**Nullable Context (= Annotation Context + Warning Context)**
Bedeutung der Deklaration Klasse k;	Non-Nullable	Nullable	Non-Nullable
Bedeutung der Deklaration Klasse? k;	Nullable	Nicht erlaubt (führt zur Warnung)	Nullable
Warnung vor Null-Reference-Laufzeitfehlern	Nein	Ja	Ja
Aktivierung auf Projektebene in der .csproj-Datei	<Nullable> annotations </Nullable>	<Nullable> warnings </Nullable>	<Nullable> enable </Nullable>
Aktivierung in C#-Programmcodedatei (.cs) für die folgenden Zeilen	#nullable enable annotations	#nullable enable warnings	#nullable enable
Deaktivierung in C#-Programmcodedatei (.cs) für die folgenden Zeilen	#nullable disable annotations	#nullable disable warnings	#nullable disable
Zurücksetzung der C#-	#nullable restore annotations	#nullable restore warnings	#nullable restore

	Nullable Annotation Context	Nullable Warning Context	Nullable Context (= Annotation Context + Warning Context)
Programmdatei für die folgenden Zeilen auf die Einstellung auf Projektebene			

Tabelle: Drei neue Kontextarten in C# 8.0

Das folgende Listing zeigt an Beispielen die Auswirkungen der drei neuen Kontextarten.

Listing: Basiswissen zu den Nullable-Kontexten

```
// Normaler Kontext
string name1 = null;
Experte e1 = null;
int id1 = 1;
int? plz1 = null;

// Nullable Context einschalten
#nullable enable
string name2 = null; // Non-Nullable Reference Type -> Warnung!
string? name3 = null; // Nullable Reference Type
Experte e2 = null;// Non-Nullable Reference Type -> Warnung!
Experte? e3 = null;  // Nullable Reference Type
int id2 = 1; // keine Auswirkung auf Value Types!
int? plz2 = null; // keine Auswirkung auf Value Types!
Console.WriteLine(name2.Trim()); // Warnung: Dereference of a possibly null refer
ence
Console.WriteLine(name3.Trim()); // Warnung: Dereference of a possibly null refer
ence
Console.WriteLine(plz2.ToString()); // keine Warnung

// Nullable Context wieder ausschalten
#nullable disable
name2 = null; // keine Warnung
string? name4 = null; // Warnung bei ?

// nur Nullable Annotations Context einschalten
#nullable enable annotations
string name5 = null; // Nullable Reference Type, keine Warnung!
string? name6 = null; // Nullable Reference Type
Console.WriteLine(name5.Trim()); // keine Warnung
Console.WriteLine(name6.Trim()); // keine Warnung
#nullable disable annotations

// nur Nullable Warning Context einschalten
#nullable enable warnings
string name7 = null; // Nullable Reference Type, keine Warnung!
string? name8 = null; // Warnung bei ?, Nullable Reference Type nicht erlaubt
Console.WriteLine(name7.Trim()); // Warnung: Dereference of a possibly null refer
ence
```

```
Console.WriteLine(name8.Trim()); // Warnung: Dereference of a possibly null refer
ence
#nullable disable warnings
```

20.3.2 Compiler erkennt die Programmierfehler nicht

Zum Praxistest wird das Programm im nachstehenden Listing verwendet. Der C#-Compiler übersetzt den Programmcode fehlerfrei und ohne Warnungen.

Einwandfrei funktionieren kann der Programmcode freilich nicht: Bei der Ausführung sieht man direkt zweimal den Laufzeitfehler "NullReferenceException: Object reference not set to an instance of an object."

Hier müsste man null-Prüfungen oder eine Toleranz gegenüber null einbauen.

Listing: Ein Programm mit NullReference-Fehlern

```
using ITVisions;
using System;

namespace CS80
{
 class NullableRefTypes
 {
  public static void Run()
  {
   CUI.MainHeadline(nameof(NullableRefTypes) + ": 1. String");
   try
   {
    string Name = null;
    Print("Guten Tag, " + Name);
    Console.WriteLine($"Ihr Name ist {Name.Length} Zeichen lang!");
   }
   catch (System.Exception ex)
   {
    CUI.PrintError("ERROR: " + ex.Message);
   }

   CUI.MainHeadline(nameof(NullableRefTypes) + ": 2. Person");
   try
   {
    Person p1 = new Person() { ID = 123, Surname = "Schwichtenberg" };
    PrintPerson(p1);
    Person p2 = null;
    PrintPerson(p2);

    p1.Firstname = null;
    string name = p1.Firstname.ToUpper();
    Console.WriteLine(name);
   }
   catch (System.Exception ex)
   {
    CUI.PrintError("ERROR: " + ex.Message);
   }
  }
```

```
 static void Print(string s)
 {
  Console.WriteLine(s.Trim());
 }

 static void PrintPerson(Person p)
 {
  Console.WriteLine($"{p.ID}: {p.ToString()}");
 }

class Person
{
 public int ID { get; set; }
 public string Firstname { get; set; }
 public string Surname { get; set; }

 public Person()
 {  }

 public Person(int ID) : this()
 {
  this.ID = ID;
 }

 public override string ToString()
 {
  return this.Firstname.ToUpper() + " " + this.Surname.ToUpper();
 }
}
}
```

```
C#-Syntaxdemo-Sammlung. (C) Dr. Holger Schwichtenberg 2001-2019
OS Platform: Win32NT
OS Version: Microsoft Windows NT 6.2.9200.0
Anwendung kompiliert für: .NETFramework,Version=v4.7.2
Anwendung läuft auf: 4.8.03752 / 528040
------------------------------------------------
NullableRefTypes: 1. String
Guten Tag,
ERROR: Object reference not set to an instance of an object.
NullableRefTypes: 2. Person
ERROR: Object reference not set to an instance of an object.
```

Abbildung: Ausgabe des obigen Programms

20.3.3 Aktivieren der Null-Referenz-Prüfung

Seit C# 8.0 gibt es die optionale strengere Null-Referenz-Prüfung. Den Nullable Kontext (mit Annotation Context und Warning Context) aktiviert man in einer Programmcodedatei mit

```
#nullable enable // Nullable check for Reference Types
```

Man kann diese Prüfung auch jederzeit wieder deaktivieren mit

```
#nullable disable // Nullable check for Reference Types wieder aus
```

Man kann diese Prüfung auch für ein ganzes Projekt aktivieren. Dies erfolgt in der Projektdatei (.csproj) per:

```
<PropertyGroup>
  <Nullable>true</Nullable>
  …
</PropertyGroup>
```

Auch eine auf Projektebene gesetzte Prüfung kann der Entwickler im Programmcode jederzeit deaktivieren. Der Ausdruck

```
#nullable restore
```

bedeutet, dass die Einstellung auf Projektebene wieder gelten soll.

Hinweis: Diese neue Option ist möglich in .csproj-Dateien sowohl für das klassische .NET Framework als auch in den kompakteren .NET Core-Projektdateien.

Mit der strengeren Null-Referenz-Prüfung kommt es in dem obigen Listing zu neun Warnungen. Da es nur Warnungen sind, kompiliert das Programm weiterhin und es kommt immer noch zu den Laufzeitfehlern.

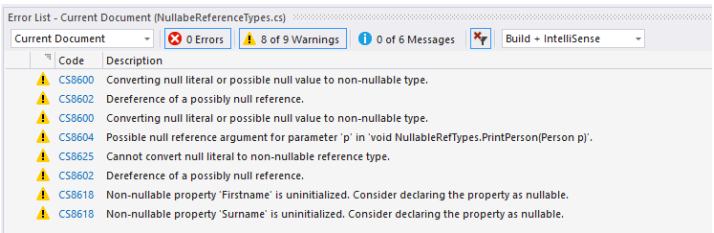


Abbildung: Warnungen bei aktivierter Null-Referenz-Prüfung

Praxistipp: Durch einen Eintrag in der Projektdatei kann man ausgewählte Warnungen zu Fehlern hochstufen, z.B. <WarningsAsErrors>CS8600;CS8602;CS8603;CS8604;CS8625 </WarningsAsErrors>

20.3.4 Verbessertes Programm

Das nächste Listing zeigt das verbesserte Programm, das nun alle strengeren Null-Reference-Prüfung besteht.

In dem Listing wurde geändert:

- Variablen für Referenztypen, die null erlauben sollen, wurden explizit mit einem Fragezeichen versehen, also zu Nullable Reference Types gemacht, z.B. string? und Person?
- Es wurden Null-tolerierende Operatoren eingebaut, z.B. mit den Operatoren ?? und ?.
- Es wurden Null-Prüfungen eingebaut, z.B. if (p == null) { … }
- Es wurden Initialisierungen ergänzt, z.B. Firstname = ""; Surname = "";

- Es wurde der neue sogenannte Null Forgiveness-Operator eingebaut: this.Firstname!.ToUpper() + " " + this.Surname!.ToUpper();

Hinweis: Nullable Reference Types darf man anders als Nullable Values Types nicht mit System.Nullable<T> deklarieren. Erlaubt ist nur die Schreibweise string?, nicht Nullable<string>.

Listing: Verbessertes Programm ohne NullReference-Fehler

```
using ITVisions;
using System;
#nullable enable // Nullable check for Reference Types

namespace CS80
{
 class NullableRefTypesMitPrüfungen
 {
  public static void Run()
  {
   CUI.MainHeadline(nameof(NullableRefTypes) + ": 1. String");
   try
   {
    string? Name = null;
    Print("Guten Tag, " + Name);
    Console.WriteLine($"Ihr Name ist {Name?.Length ?? 0} Zeichen lang!");
   }
   catch (System.Exception ex)
   {
    CUI.PrintError("ERROR: " + ex.Message);
   }

   CUI.MainHeadline(nameof(NullableRefTypes) + ": 2. Person");
   try
   {
    Person p1 = new Person() { ID = 123, Surname = "Schwichtenberg" };
    PrintPerson(p1);
    Person? p2 = null;
    PrintPerson(p2);

    p1.Firstname = null;
    string name = p1.Firstname!.ToUpper();
    Console.WriteLine(name);
   }
   catch (System.Exception ex)
   {
    CUI.PrintError("ERROR: " + ex.Message);
   }
  }

  static void Print(string s)
  {
   Console.WriteLine(s.Trim());
  }
```

```
  static void PrintPerson(Person? p)
  {
   if (p == null) { Console.WriteLine("Person ist leer!"); return; }
   // oder: null coalescing assignment ("compound assigment")
   //p ??= new Person() { ID = -1 };
   Console.WriteLine($"{p.ID}: {p.ToString()}");
  }

  class Person
  {
   public int ID { get; set; }
   public string? Firstname { get; set; }
   public string? Surname { get; set; }

   public Person()
   {
    Firstname = ""; Surname = "";
   }

   public Person(int ID) : this()
   {
    this.ID = ID;
   }

   public override string ToString()
   {
    // Null Forgiveness-Operator zur als Beispiel
    return this.Firstname!.ToUpper() + " " + this.Surname!.ToUpper();
    // besser wäre eine Null-tolerierende Lösung:
    return this.Firstname?.ToUpper() + " " + this.Surname?.ToUpper();
   }
  }
 }
}
```

20.3.5 Null Forgiveness-Operator

Der Null Forgiveness-Operator (!.) unterdrückt Warnungen der Null-Referenz-Prüfung. Er stellt ein Risiko dar, den man nur als letztens Mittel einsetzen sollte, wenn man ganz sicher ist, dass Null nicht vorkommen kann.

In den meisten Fällen sollte der Null Forgiveness-Operator nicht notwendig sein.

Statt

```
   public override string ToString()
   {
    // Null Forgiveness-Operator zur als Beispiel
    return this.Firstname!.ToUpper() + " " + this.Surname!.ToUpper();
   }
```

Besser wäre eine Null-tollerierende Lösung:

```
   public override string ToString()
   {
    return this.Firstname?.ToUpper() + " " + this.Surname?.ToUpper();
```

```
}
```

Oder ein Beheben des Problems:

```
public override string ToString()
{
 if (this.Firstname == null) this.Firstname = "";
 if (this.Surname == null) this.Surname = "";
 return this.Firstname.ToUpper() + " " + this.Surname.ToUpper();

 // Null Forgiveness-Operator zur als Beispiel
 return this.Firstname!.ToUpper() + " " + this.Surname!.ToUpper();
 // besser wäre eine Null-Toleranz:
 return this.Firstname?.ToUpper() + " " + this.Surname?.ToUpper();
}
```

In allen drei o.g. Fällen kommt es zu keiner Compiler-Warnung.

21 Partielle Klassen, Methoden, Properties und Indexer

Mit dem Schlüsselwort partial kann man Aufspaltungen von Code vornehmen, was in der Regel genutzt wird, um Code auf mehrere Dateien zu verteilen:

- Klassen lassen sich aufteilen, indem ein einige Mitglieder in einem Teil liegen und andere Mitglieder in dem anderen Teil
- Bei Methoden, Properties und Indexer kann man die Deklaration von der Implementierung trennen.

21.1 Partielle Klassen

Partielle Klasse gibt es in C# schon sehr lange: seit .NET Framework 2.0 und C# 2.0 (Jahr 2005). Partielle Klassen erlauben dem Entwickler den Programmcode einer Klasse auf mehrere einzelne Klassendefinitionen aufzuteilen. Dabei können die partiellen Klassendefinitionen auch in verschiedenen Dateien existieren. Die verschiedenen Klassendefinitionen werden von dem Compiler zu einer Klasse vereint. Dies bedeutet, dass alle Klassenmitglieder, auch wenn sie in verschiedenen Dateien liegen, sich gegenseitig sehen und nutzen können.

Partielle Klassen erlauben, dass verschiedene Entwickler an einer Klasse arbeiten können bzw. dass ein Teil einer Klasse automatisch durch ein Werkzeug generiert wird, während andere Teile händisch codiert werden. Partielle Klassen werden von verschiedenen Werkzeugen in Visual Studio verwendet, um generierten Programmcode von eigenem Programmcode zu trennen (z.B. in Windows Forms, ASP.NET Webforms, typisierten DataSets, Entity Framework, ASP.NET Core Blazor).

Entwickler können partielle Klassen auch dazu verwenden, den eigenen Code übersichtlicher zu halten. Allerdings gibt es Verfechter der Regel, dass eine Klassendefinition nicht so lang sein sollte, dass man eine Aufteilung auf mehrere Dateien überhaupt in Betracht ziehen müsste (vgl. [*dzone.com/articles/rule-30-%E2%80%93-when-method-class-or*]). Demnach sollte man in solchen Fällen die Funktionalität der großen Klasse nach inhaltlichen Kriterien auf mehrere Klassen aufteilen.

> **Hinweis:** Über das Schlüsselwort `partial` verbunden werden können auf diese Weise aber nur Klassen im Quellcode und innerhalb einer Assembly. Sie können also keine Klasse in einer referenzierten Assembly erweitern. Letzteres ist nur mit Vererbung möglich (sofern die Klasse es erlaubt).

Es gelten folgende Bedingungen für den Einsatz des Schlüsselwortes `partial`:

- `partial` muss klein geschrieben werden
- `partial` muss hinter den Sichtbarkeitsmodifizierern der Klasse stehen
- `partial` muss bei allen Teilklassen angegeben werden

Listing: Datei PartielleKlasse_Teil1.cs

```
namespace CS20
{
 public partial class Buch
 {
  public Buch(string titel, string ISBN)
  {
```

```
   this.ISBN = ISBN;
   // kann Property aus Teil 2 der Klasse verwenden!
   this.Titel = titel;
  }
  public string ISBN;
 }
}
```

Listing: Datei PartielleKlasse_Teil2.cs

```
namespace CS20
{
 public partial class Buch
 {
  public string Titel;

  public override string ToString()
  {
   // kann Property aus Teil 1 der Klasse verwenden!
   return "Buch '" + this.Titel + "' (ISBN " + this.ISBN + ")";
  }
 }
}
```

21.2 Partielle Methoden

Partielle Methoden gibt es seit C# 3.0. Im Rahmen von C# 9.0 wurden sie erweitert.

Im Rahmen eines Teils einer partiellen Klasse kann man eine Methode deklarieren (ohne Implementierung). Im Rahmen eines anderen Teils kann man die Implementierung liefern. So lassen sich die Deklaration und die Implementierung trennen. Die partielle Methode kann gleichwohl in dem Teil, in dem sie nur deklariert ist, aufgerufen werden. Wenn es keine Implementierung in einem anderen Teil gibt, kommt es aber nicht zu einem Fehler. Der Compiler wird vielmehr alle Aufrufe entfernen. Damit kann man partielle Methoden als Hooks einsetzen, um sich in Programmcode einzuklinken. Gerne wird dies benutzt bei Programmcode, der von einem Codegenerator (Assistenten oder Designer) erzeugt wurde. Zum ersten Mal eingesetzt wurde diese Vorgehensweise im LINQ to SQL-Designer.

Hinweis: Partielle Attribute (Properties) gibt es leider bisher nicht.

Es galten folgende Bedingungen für partielle Methoden in C# 3.0 bis 8.0:

- Die Methode darf keinen Rückgabewert (void) haben.
- Beide Teile müssen partial verwenden.
- Die Methode ist automatisch private. Sie dürfen nicht öffentlich sein.
- Eine Sichtbarkeit darf nicht angegeben sein (also auch nicht private).
- Parameter mit out sind nicht erlaubt.
- Sie können statisch sein.

Listing: Beispiel für eine partielle Methode in C# seit Version 3.0

```
public partial class Vorstandsmitglied
 {
  // Automatic Properties
  public string Name { get; set; }
```

```
  public string Aufgabengebiet { get; set; }
  public int Alter { get; set; }
  public string Ort;

  public override string ToString()
  {
   // Partielle Methode - Verwendung
   OnToString();
   return Name;
  }

  // Partielle Methode - Deklaration
  partial void OnToString();
 }

 public partial class Vorstandsmitglied
 {
  // Partielle Methode - Implementierung
  partial void OnToString()
  {
   Console.WriteLine("ToString aufgerufen!");
  }
 }
```

Seit C# 9.0 sind einige dieser Restriktionen gelockert: Rückgabewerte, Sichtbarkeitsangabe und out-Parameter sind erlaubt. Allerdings muss es bei Verwendung dieses Features dann auch zwingend eine Implementierung geben!

Listing: Partielle Methoden alten vs. neuen Typs / Erster Teil der partiellen Klasse:

```
partial class MeineKlasse
 {
  // partielle Methode alten Typs --> keine Implementierung erforderlich!
  partial void M1();

  // partielle Methode neuen Typs, da "private" --> Implementierung erforderlich!
  private partial void M2();

  // partielle Methode neuen Typs, da "int" --> Implementierung erforderlich!
  public partial int M3();
 }
```

Listing: Partielle Methoden alten vs. neuen Typs / Zweiter Teil der partiellen Klasse

```
partial class MeineKlasse
 {
  private partial void M2() { }
  public partial int M3() { return 42; }
 }
```

In diesem Beispiel käme es zu Compilerfehlern, wenn:

- Die Implementierung von M2() oder M3() fehlt: "Partial method xy must have an implementation part because it has accessibility modifiers."
- Bei M3() kein Sichtbarkeitsangabe festgelegt ist: "Partial method xy must have accessibility modifiers because it has a non-void return type.
- Der Rückgabetyp von Deklaration und Implementierung nicht übereinstimmen: "Both partial method declarations must have the same return type."

- Die Sichtbarkeitsangabe von Deklaration und Implementierung nicht übereinstimmen: "Both partial method declarations must have identical accessibility modifiers."

21.3 Partielle Properties und partielle Indexer (ab C# 13.0)

Eine wichtige Neuerung in C# 13.0 sind partielle Properties und Indexer. Auf dieses Sprachfeature warten viele Entwicklerinnen und Entwickler bereits seit der Einführung der partiellen Methoden in C# 3.0. Das C#-Schlüsselwort partial gibt es sogar bereits seit C# 2.0 für Klassen.

Mit partiellen Klassen kann man den Programmcode einer einzigen Klasse auf mehrere Code-Dateien aufspalten - ohne dafür Vererbung zu nutzen. Das ist nicht nur sinnvoll für mehr Übersichtlichkeit bei umfangreichen Klassen, sondern wird vor allem verwendet, wenn ein Teil der Klasse automatisch generiert und der andere Teil der Klasse manuell geschrieben wird. Diese Vorgehensweise kommt in .NET zum Beispiel bei GUI-Bibliotheken wie ASP.NET Webforms und Blazor, beim Reverse Engineering von Datenbanken mit Entity Framework und Entity Framework Core sowie bei Source-Generatoren (z.B. für reguläre Ausdrücke und JSON-Serialisierung) zum Einsatz.

Nun in C# 13.0 können Entwicklerinnen und Entwickler auch Property-Definitionen und Indexer-Definition sowie deren Implementierung mit partial in zwei Dateien trennen. Dabei müssen beide Teile jeweils die gleiche Kombination von Getter und Setter mit den gleichen Sichtbarkeiten realisieren. Ein konkretes Beispiel: Wenn in einem Teil der Klasse ein Property sowohl einen öffentlichen Getter als auch einen öffentlichen Setter besitzt, müssen diese auch im anderen Teil vorhanden und öffentlich sein. Aber während in einem Teil ein automatisches Property verwendet wird, kann im anderen Teil eine explizite Implementierung vorhanden sei.

Die folgenden drei Listings zeigen ein Beispiel einer aufgeteilten Klasse mit partieller Methode und partiellem Property sowie einem partieller Indexer.

Listing: Erster Teil der partiellen Klasse nur mit Definitionen von ID und Print()

```
/// <summary>
/// Erster Teil der partiellen Klasse nur mit Definitionen von ID, Indexer und Pr
int()
/// </summary>
public partial class PersonWithAutoID
{
 // NEU: Partielles Property --> kein "Convert to Full Property"
 public partial int ID { get; set; }
 // NEU: Indexer
 public partial string this[int index] { get; }
 // "Normales Property"
 public string Name { get; set; }
 // Partielle Methode
 public partial void Print();
}
```

Listing: Im zweiten Teil der partiellen Klasse werden Getter und Setter für ID sowie die Methode Print() implementiert

```
/// <summary>
/// Im zweiten Teil der Klasse werden Getter und Setter für ID, der Getter für de
n Indexer sowie die Methode Print() implementiert
/// </summary>
public partial class PersonWithAutoID
```

```
{
 int counter = 0;

 // Implementierung des Partial Property
 private int iD;

 public partial int ID
 {
  get
  {
   if (iD == 0) iD = ++counter;
   return iD;
  }
  set
  {
   if (ID > 0) throw new ApplicationException("ID ist bereits gesetzt");
   iD = value;
  }
 }

 // Implementierung des Partial Indexer
 public partial string this[int index]
 {
  get
  {
   return index switch
   {
    0 => ID.ToString(),
    1 => Name,
    _ => throw new IndexOutOfRangeException()
   };
  }
 }

 // Implementierung der Partial Method
 public partial void Print()
 {
  Console.WriteLine($"{this.ID}: {this.Name}");
 }
}
```

Listing: Nutzer der zusammengesetzten Klasse PersonWithAutoID

```
/// <summary>
/// Client-Klasse für die Demo
/// </summary>
public class CS13_PartialPropertyAndIndexerDemoClient
{
 public void Run()
 {
  CUI.Demo(nameof(CS13_PartialPropertyAndIndexerDemoClient));
  CS13.PersonWithAutoID p = new() { Name = "Holger Schwichtenberg" };
  p.Print(); // 1: Holger Schwichtenberg
  CUI.H2("Versuch, die ID neu zu setzen, führt zum Fehler:");
```

```
  try
  {
   p.ID = 42;
  }
  catch (Exception ex)
  {
   CUI.Error(ex); // System.ApplicationException: ID ist bereits gesetzt
  }
  CUI.Print($"Nutzung des Indexers: {p[0]}: {p[1]} ");
 }
}
```

22 Erweiterungsmethoden (Extension Methods)

Eine Erweiterungsmethode ermöglicht einer Klasse, extern eine Methode anzuheften. *Extern* heißt, dass dies nicht im Rahmen der Klassendefinition selbst erfolgt, sondern in einer anderen Klasse. Damit ist es möglich, Klassen zu erweitern, die man selbst nicht geschrieben hat (z.B. Klassen der .NET-Klassenbibliothek *FCL*). Ein solches Konzept ist bereits aus JavaScript vielen Entwicklern bekannt. Zu beachten ist, dass die Methoden gemäß dem Prinzip der Kapselung nur auf die öffentlichen Attribute und Methoden der Klasse zugreifen können. Durch Einsatz von Reflection (Metadatennutzung) kann diese Beschränkung jedoch umgangen werden (durch Reflection kann man immer auch auf private Mitglieder zugreifen!). Erweiterungen können nur Methoden sein; Fields und Properties können leider nicht nachträglich ergänzt werden.

Tipp: Erweiterungsmethoden können auch auf Schnittstellen angewendet werden, sodass man auf einfache Weise alle Klassen erweitern kann, die eine bestimmte Schnittstelle anbieten. Microsoft hat dies im Rahmen von Language Integrated Query auf die Schnittstelle IEnumerable angewendet, um alle Objektmengenklassen »LINQ-fähig« zu machen.

Hinweis: Mit den Erweiterungsmethoden hat man eine dritte syntaktische Möglichkeit, bestehende Klassen zu erweitern:

1. Vererbung: Möglich seit .NET 1.0, aber nur für Klassen, die Vererbung zulassen (also nicht sealed bzw. NotInheritable sind)
2. Partielle Klassen: Möglich seit .NET 2.0, aber nur für Klassen im gleichen Projekt, die als Partiell gekennzeichnet sind
3. Erweiterungsmethoden: Möglich seit .NET 3.5, für alle Klassen und auch anwendbar auf Schnittstellen

22.1 Entwicklung von Erweiterungsmethoden

Um in C# eine Erweiterungsmethode zu entwickeln, schreibt man:

- eine statische Klasse
- mit einer statischen Methode
- die mindestens einen Parameter besitzt
- der mit this beginnt
- und den Typ der zu erweiternden Zielklasse besitzt

Hinweise:

1. Der Name der Klasse, in der die Erweiterungsmethode implementiert wird, ist im Übrigen egal. Auf diese Weise ist die Anzahl der Erweiterungsmethoden für eine Klasse nicht räumlich und der Menge nach beschränkt. Erweiterungsmethoden können überladen werden, wobei hier die gleichen Bedingungen wie bei normalen Methoden gelten. Erweiterungsmethoden müssen keinen Rückgabewert haben (d. h. void bzw. Sub sind erlaubt).

2. Eine Erweiterungsmethode darf nicht in einer eingebetteten Klasse definiert werden.

3. Die Verwendung von this ist leider wenig intuitiv, zumal this schon mehrere andere Bedeutungen in C# hat. Außerdem muss die Erweiterungsmethode statisch deklariert sein, wenngleich sie nachher eine Instanzmethode ist. Ebenso muss die Klasse statisch sein.
4. Falls Sie von Visual Basic .NET kommen: Die dort übliche Verwendung der Annotation System.Runtime.CompilerServices.ExtensionAttribute funktioniert in C# nicht!

Das folgende Beispiel zeigt die Implementierung einer Erweiterungsmethode Print() für die Schnittstelle IEnumerable. Dadurch erhalten alle Objektmengenklassen in .NET die Methode Print(), die alle enthaltenen Objekte in einer bestimmten Farbe an der Konsole ausgibt (die Ausgabe erfolgt mit ToString() und ist daher darauf angewiesen, dass ToString() in den Objekten sinnvoll implementiert wurde.

Listing: Implementierung der Erweiterungsmethode Print() für die Schnittstelle IEnumerable (in C#)

```
using System.Runtime.CompilerServices;
using System;
using System.Collections;

namespace ITVisions
{
 public static class ITVisionsCollectionExtensions
 {
  // --- Erweiterungsmethode für IEnumerable
  public static void Print(this IEnumerable Menge, ConsoleColor Farbe)
  {
   ConsoleColor VorherigeFarbe = Console.ForegroundColor;
   Console.ForegroundColor = Farbe;
   foreach (object o in Menge)
    Console.WriteLine(o.ToString());
   Console.ForegroundColor = VorherigeFarbe;
  }
 }
}
```

22.2 Nutzung von Erweiterungsmethoden

Wichtig ist, dass in der Klasse, in der die Erweiterungsmethode verwendet wird, der Namensraum der Klasse, in der die Erweiterungsmethode implementiert wurde, durch using bzw. imports eingebunden wird. Sonst kann die Erweiterungsmethode vom Compiler nicht gefunden werden. Dies ist auch der Grund dafür, dass LINQ-Abfrageausdrücke nur dann zur Verfügung stehen, wenn der Namensraum System.Linq eingebunden wurde!

Listing: Anwendung der Methode Print() auf eine Menge, die mit der generischen Mengenklasse List erzeugt wurde (in C#)

```
Imports de.WWWings.Library
…
List<Vorstandsmitglied> Vorstandsmitglieder = new List<Vorstandsmitglied> { HS,
HM, MM };

// Verwendung einer Erweiterungsmethode
Vorstandsmitglieder.Print(ConsoleColor.DarkYellow);
```

22.3 Praxisbeispiele: Erweiterungsmethoden für die Datentypkonvertierung

Mit ein Erweiterungsmethoden kann man die Konvertierung von elementaren Datentypen wesentlich schöner gestalten.

Motivation: Die Konvertierung zwischen elementaren Datentypen gehört zum Alltag eines jeden Softwareentwicklers, denn nicht immer kommen Daten in dem gewünschten Typ im eigenen Programmcode an. Datenbankzugrifftechniken wie DataReader und das untypisierte DataSet liefern Daten aus Datenbankspalten in Form des allgemeinen .NET-Basistyps System.Object. Beim Auslesen einer Textdatei bekommt man alle Daten als Zeichenketten. Ebenso liefern Texteingabefelder in grafischen Benutzeroberflächen üblicherweise System.String. Auch in Verbindung mit dem Netzwerkprotokoll HTTP hantiert man meist mit Zeichenketten.

22.3.1 Eingebaute Konvertierungsfunktionen

Nehmen wir als Beispiel mal eine Zechenkette mit Inhalt "42"

```
string input = "42";
```

Diese Zeichenkette möchte in eine Integer-Zahl umwandeln. Ein einfacher Typecast in C# ist hier nicht die Lösung

```
int x = (int)input;
```

"Cannot onvert type 'string' to 'int'", sagt der Compiler dazu nur.

Es ist die Hilfe der .NET-Klassenklassenbibliothek notwendig, z.B.

- System.Int32.Parse()
- System.Int32.TryParse()
- System.Convert.ToInt32()
- System.Convert.ChangeType()

Das nächste Listing zeigt diese vier Möglichkeiten im Rahmen von Unit Tests. Variante 2 ist eindeutig die beste Lösung, denn bei Variante 1, 3 und 4 kommt es im Fall, dass die Zeichenkette kein gültiger Ganzzahlwert ist zu einem Laufzeitfehler vom Typ System.FormatException ("Input string was not in a correct format."). Wenn die Zeichenkette den Wert null hat, gibt es den Laufzeitfehler vom Typ System.InvalidCastException ("Null object cannot be converted to a value type") bzw. System.ArgumentNullException ("Value cannot be null.").

Diese Fehlerfälle müsste man also explizit abfangen. Man sollte aber das Auftreten einer Ausnahme in .NET wenn immer möglich vermeiden, da Ausnahmen viel Zeit kosten. Diese Zeit fällt zwar kaum ins Gewicht, wenn man Eingaben eines Benutzers in einer Bildschirmmaske prüft. Die Zeit für das Abfangen der Laufzeitfehler ist aber relevant, wenn man einen Datenimport mit 500.000 Datensätzen aus einer Textdatei implementiert und es häufig fehlerhafte Daten gibt.

Listing: In .NET eingebaute Möglichkeiten der Konvertierung zwischen einer Zeichenkette und einer Zahl

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DNP.Kolumne.Folge152.UnitTests
{
```

```
 [TestClass]
 public class StringConversionTests
 {

[TestMethod]
public void StandardKonvertierungen_Parse()
{
 string input = "42";
 int x = System.Int32.Parse(input);
 Assert.AreEqual(42, x);
}

[TestMethod]
public void StandardKonvertierungen_TryParse()
{
 string input = "42";
 bool success = System.Int32.TryParse(input, out int x);
 if (success)
 {
  Assert.AreEqual(42, x);
 }
 else
 {
  // Konvertierung fehlgeschlagen
 }
}

[TestMethod]
public void StandardKonvertierungen_Convert()
{
 string input = "42";
 int x = System.Convert.ToInt32(input);
 Assert.AreEqual(42, x);
}

[TestMethod]
public void StandardKonvertierungen_ChangeType()
{
 string input = "42";
 int x = (int)System.Convert.ChangeType(input, typeof(int));
 Assert.AreEqual(42, x);
}
 }
}
```

22.3.2 Erweiterungsmethoden zum Konvertieren

Der eine oder andere erinnert sich vielleicht noch an die Version Beta 1 von .NET Framework 1.0 im Jahr 2000 – das ist zugegebener Maßen lange her. Einige der heutigen .NET-Entwickler waren da noch nicht geboren. In dieser Beta-Version gab es Konvertierungsmethoden direkt in der System.String-Klasse: ToInt32(), ToDateTime(), ToDecimal() usw. Leider haben es diese Konvertierungsmethoden bis heute in keine einsatzreife Version von .NET geschafft. Mit ein klein

wenig Zutun können diese Konvertierungsmethoden aber selbst erschaffen werden, als eine elegantere Lösung um die Methode TryParse() herum.

Das folgende Listing zeigt die Methoden ToInt32() und ToInt32OrNull(). Beides sind Erweiterungsmethoden für die Klasse System.String. Beide sind daher statische Methoden in einer statischen Klasse und haben ein "this" vor dem ersten Parametertyp – das alles gehört zu den Voraussetzungen für Erweiterungsmethoden in C#. Beide Methoden kapseln den Aufruf von System.Int32.TryParse(). Beide Methoden erlauben die optionale Angabe eines Parameters mit einem Wert, der verwendet wird für den Fall, dass eine Konvertierung in eine Zahl nicht möglich war. Während bei ToInt32() immer eine Zahl zurückkommt (man muss sich also überlegen, was im Fehlerfall eine Zahl ist, an der man erkennt, dass die Konvertierung nicht geklappt hat), erlaubt ToInt32OrNull() die Rückgabe von null. Daher definiert ToInt32() als Rückgabewert System.Int32 und ToInt32OrNull() liefert System.Nullable<System.Int32> alias Int32?.

Listing: Erweiterungsmethoden ToInt32() und ToInt32OrNull()

```
namespace ITVisions
{
 public static class StringExtensions
 {

/// <summary>
  /// Konvertiert eine Zeichenkette nach Int32 oder in NULL-Wert
  /// </summary>
  /// <param name="obj">Zielobjekt</param>
  /// <param name="defaultValue">Rückgabestandardwert für den Fall, das Konvertie
rung nicht erfolgreich ist- Ohne Angabe ist der Rückgabestandardwert NULL.</param
>
  /// <returns>Nullable Int32</returns>
  public static Int32? ToInt32OrNull(this string obj, Int32? defaultValue = null)
  {
   int i;
   if (Int32.TryParse(obj, out i)) return i;
   return defaultValue;
  }

  /// <summary>
  /// Konvertiert eine Zeichenkette nach Int32,
  /// </summary>
  /// <param name="obj">Zielobjekt</param>
  /// <param name="defaultValue">Rückgabestandardwert für den Fall, das Konvertie
rung nicht erfolgreich ist. Ohne Angabe ist der Rückgabestandardwert 0.</param>
  /// <returns>Int32</returns>
  public static Int32 ToInt32(this string obj, Int32 defaultValue = 0)
  {
   int i;
   if (Int32.TryParse(obj, out i)) return i;
   return defaultValue;
  }
}
```

Das nächste Listing zeigt die Nutzung obiger Erweiterungsmethoden. Wichtig ist dabei

```
using ITVisions;
```

Erst durch diesen Namensraumimport werden alle Erweiterungsmethoden in statischen Klassen in diesem Namensraum eingebunden. Die Erweiterungsmethoden funktionieren beide auch für den Null-Fall, wie die Unit Tests im nächsten Listing beweisen. Normale Instanzmethoden in einer Klasse würden hier versagen, denn auf einem Objektverweis, der auf null steht, könnte man keine Methode aufrufen. Erweiterungsmethoden können damit aber umgehen, da sie das Objekt als Parameter erhalten.

Listing: Unit Tests für ToInt32() und ToInt32OrNull()

```
using System;
using ITVisions;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ITV.AppUtil.UnitTests
{

 [TestClass]
 public class StringConversionTests
 {

  [TestMethod]
  public void ToInt32_Valid()
  {
   string input = "42";
   int? x1 = input.ToInt32OrNull();
   Assert.AreEqual(42, x1);

   int x2 = input.ToInt32();
   Assert.AreEqual(42, x2);

   int x3 = input.ToInt32(-1);
   Assert.AreEqual(42, x3);
  }

  [TestMethod]
  public void ToInt32_NotValid()
  {
   string input = "abc";
   int? x1 = input.ToInt32OrNull();
   Assert.AreEqual(null, x1);

   int x2 = input.ToInt32();
   Assert.AreEqual(0, x2);

   int x3 = input.ToInt32(-1);
   Assert.AreEqual(-1, x3);
  }

  [TestMethod]
  public void ToInt32_Null()
  {
   string input = null;
   int? x1 = input.ToInt32OrNull();
```

```
   Assert.AreEqual(null, x1);

   int x2 = input.ToInt32();
   Assert.AreEqual(0, x2);

   int x3 = input.ToInt32(-1);
   Assert.AreEqual(-1, x3);
  }

 }
}
```

22.3.3 Erweiterungsmethoden für Zeichenketten mit null

Diese Anwendbarkeit von Erweiterungsmethoden auf null-Verweise kann man sich auch zu Nutze machen für weitere elegante Erweiterungsmethoden, die null-Fälle in Zeichenketten abhandeln, siehe nächstes Listing.

Listing: Erweiterungsmethoden für die String-Klasse zur Prüfung und Behandlung von null-Werten

```
namespace ITVisions
{
 public static class StringExtensions2
 {

  public static bool IsNullOrEmpty(this string s)
  {
   return (String.IsNullOrEmpty(s));
  }

  public static bool IsNotNullOrEmpty(this string s)
  {
   return (!String.IsNullOrEmpty(s));
  }

  public static string NotNull(this string s, string altString = "")
  {
   if (String.IsNullOrEmpty(s)) return altString;
   return s;
  }
}
```

Mit der hier realisierten Erweiterungsmethode IsNullOrEmpty() kann man anstelle der Nutzung der statischen Methode IsNullOrEmpty() in der Klasse System.String

```
bool b1 = String.IsNullOrEmpty(input);
```

nun deutlich prägnanter schreiben:

```
bool b1 = input.IsNullOrEmpty();
```

Ebenso statt

```
bool b2 = !String.IsNullOrEmpty(input);
```

nun

```
bool b2 = input.IsNotNullOrEmpty();
```

Mit der dort realisierten Erweiterungsmethode NotNull() kann man statt

```
string output1 = input ?? "";
```

auch schreiben

```
string output1 = input.NotNull();
```

Das sind auf den ersten Blick mehr Zeichen, aber die IntelliSense-Eingabeunterstützung im Editor sorgt dafür, dass man den Erweiterungsmethodenaufruf schneller eingeben kann, denn für ?? "" muss man (inklusive der Leerzeichen) sechs Tasten tippen, vier davon mit gedrückter Shift-Taste. Für .NotNull() reichen drei Tastaturanschläge: Punkt, N und Tabulator-Taste. Noch besser wird das Tastaturanschlagsanzahlverhältnis, wenn man danach noch weitere Methoden aufrufen will, was beim Operator ?? eine Klammerung erfordert:

```
int len1 = (input ?? "").Length;
```

daraus wird nun schneller eingebbar:

```
int len1 = input.NotNull().Length;
```

Die Erweiterungsmethode NotNull() unterstützt dabei auch alternative Texte, z.B.

```
string output = input.NotNull("- keine Angabe -");
```

22.3.4 Erweiterungsmethoden für beliebige null-Verweise

Die null-Prüfung kann man leicht auf beliebige Objekte ausdehnen (siehe Listing).

Listing: Erweiterungsmethoden für System.Object zur Prüfung von null-Werten

```
namespace ITVisions
{

 public static class ObjectExtensions
 {
  public static bool IsNull(this object o)
  {
   return (o == null);
  }

  public static bool IsNotNull(this object o)
  {
   return (o != null);
  }

  public static object NotNull(this object o, object defaultObject)
  {
   if (o is null) return defaultObject;
   return o;
  }

}
```

Nun gibt es ja schon mehrere eingebaute Möglichkeiten zur null- bzw. nicht-null-Prüfung in C#:

- bool b1 = input == null; (seit C# 1.0)
- bool b2 = input != null; (seit C# 1.0)
- bool b3 = input is null; (seit C# 7.0)
- bool b4 = input is not null; (seit C# 9.0)

Mit den Erweiterungsmethoden aus obigem Listing kann man nun allerdings mit besserer Eingabeunterstützung schreiben:

- bool b1 = input.IsNull();
- bool b2 = input.IsNotNull();

Das ist aus der Sicht des Autors dieses Buchs auch besser lesbar. Über "bessere Lesbarkeit" kann man aber streiten. Es gibt verschiedene Wahrnehmungstypen unter den Menschen.

Das obige Listing beinhaltet auch zwei Erweiterungsmethoden NotNull() für System.Object. Damit geht nun statt

```
(input ?? new DirectoryInfo(@"t:\download")).CreateIfNotExists();
```

auch dieser Aufruf:

```
input.NotNull(new DirectoryInfo(@"t:\download")).CreateIfNotExists();
```

In diesem Fall kann man jetzt durchaus auch darüber streiten, was eleganter und schneller eingebbar ist.

22.3.5 Universelle Erweiterungsmethode To<T>

Die ADO.NET-Datenzugriffsklassen System.Data.Common.DbDataReader (und die Abkömmlinge wie der SqlDataReader) und System.Data.DataRow (als Teil des DataSet) signalisieren NULL-Spalten in einem Datenbankmanagementsystem nicht als null-Wert in C# bzw. nothing in Visual Basic .NET sondern mit einer Instanz der Klasse System.DBNull.

Auf der Suche nach einer komfortableren Lösung kommt man auf die generische Methode To<T>() in nächsten Listing als Erweiterungsmethode für System.Object. Die Methode prüft zunächst auf DBNull und liefert in diesem Fall null oder einen anderen als optionalen per Parameter defaultValue übergebenen Standardwert zurück. Die Rückgabe von null wird verweigert, wenn der übergebene generische Typ T nicht nullable ist.

Wenn kein DBNull übergeben wurde, dann holt sich die Implementierung der Methode To<T>() über die Klasse System.ComponentModel.TypeDescriptor zunächst einen Konverter von dem Quellobjekttyp in den Zieltyp (Variable targettype) zur Ausführung von ConvertTo(). Wenn es keinen Konverter dafür gibt, versucht die Methode To<T>() es beim Gegenpart, also beim Zieltyp einen Konverter zu bekommen, der ConvertFrom() unterstützt.

Listing: To<T> bietet eine universelle Konvertierung

```
public static T To<T>(this object obj, object defaultValue = null)
 {
  if (obj != null)
  {
   Type targetType = typeof(T);

   // Zieltyp ist gleich dem Quelltyp
   if (obj.GetType() == targetType)
   {
    return (T)obj;
   }

   // DBNull? Dann null zurückgeben
   if (obj == DBNull.Value)
   {
```

```
    if (defaultValue == null && targetType != typeof(string) && (!targetType.IsGe
nericType || targetType.GetGenericTypeDefinition() != typeof(Nullable<>)))
    {
     throw new InvalidOperationException("Cannot convert DBNull to " + targetType
.ToString() + " because it is a non-nullable value type");
    }
    return defaultValue.To<T>();
   }

   // Konvertierung über TypeConverter für aktuelles Objekt
   TypeConverter converter = TypeDescriptor.GetConverter(obj);
   if (converter != null)
   {
    if (converter.CanConvertTo(targetType))
    {
     return (T)converter.ConvertTo(obj, targetType);
    }
   }

   // Konvertierung über TypeConverter für Zieltyp
   converter = TypeDescriptor.GetConverter(targetType);
   if (converter != null)
   {
    if (converter.CanConvertFrom(obj.GetType()))
    {
     return (T)converter.ConvertFrom(obj);
    }
   }
  }

  return (T)obj;
 }
```

Mit To<T> kann man viele Konvertierungsfälle abdecken. Beispiele zeigen die Unit Tests im folgenden Listing.

Listing: Eine Auswahl der Unit Tests für die Konvertierungsmethode To<T>

```
[TestMethod]
 public void ToT_Int16()
 {
  string value = "42";
  string nullValue = null;
  object DBNullValue = DBNull.Value;

  var result1 = value.To<Int16>();
  var result2 = value.To<Int16?>();
  var result3 = nullValue.To<Int16?>();
  var result4 = DBNullValue.To<Int16?>();
  var result5 = DBNullValue.To<Int16>(42);

  Assert.AreEqual(42, result1);
  Assert.AreEqual(42, result2.Value);
  Assert.IsNull(result3);
  Assert.IsNull(result4);
```

```
 Assert.AreEqual(42, result5);
}

[TestMethod]
public void ToT_Bool()
{
 string nullValue = null;
 string value = "true";
 object DBNullValue = DBNull.Value;

 var result1 = value.To<Boolean>();
 var result2 = value.To<Boolean?>();
 var result3 = nullValue.To<Boolean?>();
 var result4 = DBNullValue.To<Boolean?>();
 var result5 = DBNullValue.To<Boolean>(false);

 Assert.AreEqual(true, result1);
 Assert.AreEqual(true, result2.Value);
 Assert.IsNull(result3);
 Assert.IsNull(result4);
 Assert.AreEqual(false, result5);
}

[TestMethod]
public void ToT_String()
{
 string nullValue = null;
 object value = "Holger Schwichtenberg";
 object DBNullValue = DBNull.Value;

 var result1 = value.To<string>();
 var result2 = value.To<string>();
 var result3 = nullValue.To<string>();
 var result4 = DBNullValue.To<string>();
 var result5 = DBNullValue.To<string>("Max Mustermann");

 Assert.AreEqual(value.ToString(), result1);
 Assert.AreEqual(value, result2);
 Assert.IsNull(result3);
 Assert.IsNull(result4);
 Assert.AreEqual("Max Mustermann", result5);
}
```

22.4 Sammlungen von Erweiterungsmethoden

Es gibt im Internet Sammlungen von Erweiterungsmethoden, in denen sie ähnliche Konvertierungsmethoden und andere Erweiterungsmethoden finden:

1. GitHub-Projekt: Z.ExtensionMethods

https://github.com/zzzprojects/Z.ExtensionMethods

2. extensionmethod.net

https://extensionmethod.net

23 Annotationen (.NET-Attribute)

Der Entwickler selbst kann Komponenten, Klassen und Klassenmitglieder mit zusätzlichen Informationen (Metadaten) versehen, die entweder während der Kompilierung oder zur Laufzeit der Anwendung ausgewertet werden können. Typische Beispiele für derartige Zusatzinformationen sind:

- Die Komponente hat die Version x (AssemblyVersionAttribute)
- Instanzen einer Klasse sind serialisierbar (SerializableAttribute)
- Instanzen der Klasse sollen Teil einer Transaktion sein (TransactionAttribute)
- Ein Mitglied einer Klasse ist aus Kompatibilitätsgründen zwar noch vorhanden, sollte aber nicht mehr verwendet werden, weil ein anderes, besseres Mitglied zur Verfügung steht (ObsoleteAttribute)

Leider verwendet Microsoft für diese Metadaten eine stark von der objektorientierten Lehre abweichende Begriffswelt: Die Firma nennt eine derartige Auszeichnung Attribut (engl. Attribute), was einen Namenskonflikt zu dem Begriff Attribut, also dem Datenmitglied einer Klasse darstellt (vgl. für den deutschen Sprachraum [Oesterreich, B.: Objektorientierte Softwareentwicklung, München, Wien: Oldenburg Verlag, 1997, S. 157] und [Schneider, U.; Werner, D.: Taschenbuch der Informatik, München: Fachbuchverlag Leipzig, 2004, S. 277] und für den englischen Sprachraum [Oxford Dictionary of Computing, New York: Oxford University Press, 1997, S. 243]). Die Datenmitglieder einer Klasse heißen bei Microsoft Felder (engl. Fields) und Eigenschaften (engl. Properties). Dabei denkt man doch bei Feldern eher an Arrays. Ein klarer Fall von MINFU (siehe Fussnote 1) der sich in der deutschen Übersetzung besonders schlimm auswirkt.

Hinweis: Mittlerweile verwendet Microsoft auch häufiger den Begriff Annotationen (wie in Java seit Version 5.0). Dieses Buch verwendet ebenfalls Annotation für diese Meta-Daten, während mit "Attribut" ein Datenmitglied einer Klasse bezeichnet wird.

Annotationen werden in Form von Klassen implementiert, die von der Basisklasse System.Attribute abgeleitet sind. Sie haben Namen, die auf Attribute enden, wobei bei ihrer Verwendung das Wort Attribute weggelassen werden kann (z.B. System.ObsoleteAttribute → [Obsolete]). Jeder Entwickler kann eigene Annotationen definieren. Annotationen können ein Verhalten besitzen; sie werden aber erst verarbeitet, wenn ein Typ explizit von einem Host (z.B. einer Entwicklungsumgebung) oder einem anderen Typ via Reflection nach Annotationen gefragt wird.

23.1 Annotationen verwenden

Annotationen können in C# den Typen und den Typmitgliedern in eckigen Klammern vorangestellt werden.

In dem folgenden Beispiel wird die vordefinierte Annotation System.Obsolete einer Methode zugewiesen. System.Obsolete sorgt dafür, dass der Compiler den Entwickler warnt, wenn er eine derart deklarierte Methode aufruft.

Listing: Beispiel für die Anwendung der Annotation System.Obsolete in Visual Basic .NET

```
using System;
```

[1] Auf Basis der Erkenntnis, dass Microsoft regelmäßig Probleme mit der Bezeichnung der eigenen Produkte und Konzepte hat, schuf der amerikanische Autor David S. Platt ein neues Wort: MINFU. Dies ist eine Abkürzung für MIcrosoft Nomenclature Foul-Up.

```
namespace CSharpSprachsyntax.CS10_NET10_2002
{
 public class Annotationen
 {

  public void Run()
  {
   Print("Start");
  }

  [Obsolete("Verwenden Sie bitte Log()!")]
  void Print(string s)
  {
   Console.WriteLine(s);
  }

  void Log(string s, bool mitZeit = false)
  {
   Console.WriteLine((mitZeit ? System.DateTime.Now.ToString() + ": " : "") + s);
  }

 }
}
```

Abbildung: Der Compiler warnt, wenn Sie ein mit [Obsolete] annotierter Methode aufrufen

Das zweite Beispiel zeichnet die Klasse Passagier als serialisierbar aus, d. h., ihre Instanzen können persistiert oder in einen anderen Prozess übertragen werden.

```
[System.Serializable()]
public class Passagier : de.WWWings.Person
{…}
```

Seit C# 3.0 gibt es sogenannte automatische Properties, bei denen der Compiler selbständig ein unsichtbares "Backing Field" für ein Property beim Kompilieren anlegt:

```
public int ID { get; set; }
public string Name { get; set; }
```

Vor C# 7.3 hatte ein Entwickler keine Möglichkeit, Annotationen für das automatisch generierte Backing Field zu vergeben. Das erlaubt C# Version 7.3, indem der Entwickler zwischen die öffnende eckige Klammer und den Namen der Annotation "field:" schreibt, z.B.

```
[field: NonSerialized]
```

Dies ist sinnvoll, da man einige Annotationen wie z.B. [NonSerialized] nicht auf Properties anwenden darf.

Die folgende Listing zeigt ein Anwendungsbeispiel dazu.

Listing: Annotationen für unsichtbare Backing Fields

```
[Serializable]
 public class Autor
 {
  [field: NonSerialized]
  public int AutorenID { get; set; }
  public string Name { get; set; }
  public string Themen;
 }
```

23.2 Annotationen selber schreiben

Eine Annotation schreibt man selbst, indem man eine Klasse implementiert, die von der Basisklasse `System.Attribute` erbt. Eine Annotationsklasse muss keine Mitglieder besitzen.

Eine leere Annotationsklasse ist eine "Markierungsklasse", mit der man eine Typ oder ein Mitglied eines Typs markiert, für einen bestimmten Zweck. Es gibt dann nur Ja (Markierung ist vorhanden) oder Nein (Markierung ist nicht vorhanden). Durch Hinzufügen eines Konstruktors mit Parametern kann man der Annotationsklasse Daten übergeben und damit weitere Informationen transportieren.

Bei der Deklaration einer Annotationsklasse kann man die Annotation [AttributeUsage] verwenden und damit festlegen,

- bei welchen Sprachkonstrukten die Annotation eingesetzt werden kann (Assembly, Klasse, Struktur, Enumeration, Methode, Property, Field, Event, Parameter, Rückgabewert, Generischer Parameter). Durch All gibt es keine Einschränkung.
- Mit AllowMultiple legt man fest, ob die Annotation mehrfach bei ein und demselben Sprachkonstrukt erscheinen darf
- Mit Inherited legt man fest, ob die Annotation an abgeleitete Klassen weitervererbt wird.

Beispiel: Die folgende, selbstdefinierte Annotation ProgVersion ist auf jedem Sprachkonstrukt erlaubt und dient dazu, festzuhalten, mit welcher Programmversion der Code eingeführt wurde.

Listing: Eigene Annotationen implementieren

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = false, Inherited = false)]
public class ProgVersion : System.Attribute
{
 public int Versionsnummer { get; }
 public string Notiz { get; }

 public ProgVersion(int versionsnummer, string notiz = "")
 {
  Versionsnummer = versionsnummer;
  Notiz = notiz;
 }
}
```

Die Annotation ProgVersion wird in folgendem Beispiel angewendet auf C#-Sprachkonstrukte: Properties in der klassischen Schreibweise gibt es seit C# 1.0, automatische Properties erst seit C# 3.0

Listing: Eigene Annotationen verwenden

```
public class EigeneAnnotationenAnwenden
{
 string nachname;

 [ProgVersion(1, "Klassische Property-Deklaration")]
 public string Nachname
 {
  get
  {
   return nachname;
  }

  set
  {
   nachname = value;
  }
 }

 [ProgVersion(3, "Automatisches Property")]
 public string Vorname { get; set; }
}
}
```

Eigene Annotationen haben weder für den Compiler noch die Laufzeitumgebung eine Bedeutung. Entwickler müssen selbst Code schreiben, um die Annotationen per Reflection auszuwerten.

Listing: Eigene Annotationen auswerten

```
public class AnnotationenAuswerten
{
 public static void Run()
 {
  CUI.H1("Auswertung der Annotation auf den Properties");
  var typ = typeof(EigeneAnnotationenAnwenden);
  Console.WriteLine("Klasse: " + typ.FullName);
  var properties = typ.GetProperties();
  foreach (var prop in properties)
  {
   CUI.H3("Property " + prop.Name + ": ");
   if (Attribute.IsDefined(prop, typeof(ProgVersion)))
   {
    foreach (var a in prop.GetCustomAttributesData())
    {
     Console.WriteLine("- " + a.AttributeType.FullName);
     // Schleife über alle Parameter des Konstruktors
     foreach (var arg in a.ConstructorArguments)
     {
      Console.WriteLine("  " + arg.ArgumentType + " = " + arg.Value);
     }
```

```
    // Alternative: Gezielt die Annotation holen und mit Type Cast das Annotatio
nsobjekt erhalten
    var pv = (ProgVersion)prop.GetCustomAttribute(typeof(ProgVersion), false);
    CUI.Print("  Eingeführt in " + pv.Versionsnummer + ": " + pv.Notiz, ConsoleC
olor.Yellow);
   }
  }
 }
}
```

```
Auswertung der Annotation auf den Properties
Klasse: CSharpSprachsyntax.CS10_NET10_2002.EigeneAnnotationenAnwenden
Property Nachname:
- CSharpSprachsyntax.CS10_NET10_2002.ProgVersion
  System.Int32 = 1
  System.String = Klassische Property-Deklaration
  Eingeführt in 1: Klassische Property-Deklaration
Property Vorname:
- CSharpSprachsyntax.CS10_NET10_2002.ProgVersion
  System.Int32 = 3
  System.String = Automatisches Property
  Eingeführt in 3: Automatisches Property
```

Abbildung: Ausgabe der Auswertung der Annotationen auf dem Typ "Beispiel"

23.3 Annotationen mit Typparametern

Neu in C# 11.0 ist, dass .NET-Attribute (alias "Annotationen") generische Parameter ("Generic Attributes") enthalten dürfen. Man schreibt eine generische Klasse und lässt diese – wie bei Attributen üblich – von `System.Attribute` erben:

```
public class GenericAttribute<T>
: System.Attribute
{
…
}
```

Damit kann ein Entwickler dann bei der Attribuierung einer Klasse oder Methode einen Typparameter als generischen Parameter angeben:

```
[GenericAttribute<Person>()]
class CS11_GenericAttribute_Demo
{

public Person p { get; set; }

…

 [GenericAttribute<Person>()]
 public string Print()
 {
  string s = DateTime.Now + ": " + p.ToString();
  Console.WriteLine(s);
  return s;
```

```
 }
}
```

Es ist aber nicht erlaubt, dass ein generischer Typparameter einer Klasse wieder bei einem generischen Attribut eingesetzt wird. Es dürfen beim generischen Attribut nur konkrete Typen genannt werden.

```
public class GenericType<T>
{
 [GenericAttribute<T>()] // nicht erlaubt :-(
 public string Method() => default;
}
```

Hinweis: Als Typparameter hier nicht erlaubt sind: dynamic, Nullable Reference Types, Tupel in C#-Syntax (ValueTupel<T,T> ist aber erlaubt!)

Vor C# 11.0 konnte man einen Typ an ein Attribut nur als normalen Parameter im Konstruktor übergeben:

```
public class TypeAttribute : Attribute
{
 public TypeAttribute(Type t) => ParamType = t;

 public Type ParamType { get; }
}
```

Die Nutzung sah dann so aus:

```
[TypeAttribute(typeof(Person))]
class CS11_TypeAttribute_Demo
{

 public Person p { get; set; }

 [TypeAttribute(typeof(Person))]
 public string Print1()
 {
  string s = DateTime.Now + ": " + p.ToString();
  Console.WriteLine(s);
  return s;
 }
 }
```

24 Generische Klassen

Generische Klassen (Generics) erlauben es, einen oder mehrere Typen, die die Klasse intern verarbeitet, variabel zu halten (Typparameter). Ein typischer Einsatzfall sind generische Objektmengen (siehe Klassen wie List<T> im Namensraum System.Collections.Generic in der .NET-Klassenbibliothek). Generische Objektmengen ermöglichen es, dass der Entwickler einen allgemeinen Mengentyp so prägt, dass die Menge nur Mitglieder einer bestimmten Klasse akzeptiert und dafür eine Typprüfung bereits zur Entwicklungszeit stattfindet.

Neben den in der FCL implementierten generischen Objektmengen kann man in Visual Basic .NET und C# auch selbst generische Klassen erzeugen. In diesem Kapitel wird die Definition und Verwendung eigener generischer Klassen besprochen.

24.1 Definition einer generischen Klasse

Die Unterstützung für generische Klassen wurde in C# ebenso wie in Visual Basic .NET im Rahmen von .NET 2.0 hinzugefügt. Wie in vielen anderen Punkten auch, ist der Unterschied rein syntaktisch: An die Stelle des Of-Operators in runden Klammern tritt ein Klammernpaar aus spitzen Klammern. Die Bedingungen für die generischen Typparameter (Generic Constraints) definiert man mit dem Schlüsselwort where.

Listing: Implementierung einer generischen Klasse in C#

```
public class MitarbeiterzuordnungChefTyp, AssistentTyp>
  where ChefTyp : Mitarbeiter
  where AssistentTyp : Mitarbeiter
 {
  ChefTyp Chef;
  AssistentTyp Assi;

  public MitarbeiterzuordnungChefTyp Chef, AssistentTyp Assi)
  {
   this.Chef = Chef;
   this.Assi = Assi;
  }
}
```

24.2 Verwendung einer generischen Klasse

Bei der Verwendung einer generischen Klasse müssen sowohl bei der Deklaration der Objektvariablen als auch bei der Instanziierung in spitzen Klammern <…> die zu gebrauchenden Typen angegeben werden. In dem folgenden Beispiel wird ein Team aus zwei Piloten gebildet.

In C# kommen anstelle von runden Klammen und dem Schlüsselwort Of die spitzen Klammen zum Einsatz, um die von der Klasse erwarteten Typparameter anzugeben.

Listing: Nutzung einer generischen Klasse in C#

```
Mitarbeiterzuordnung<Pilot,Pilot> CockpitTeam;
Pilot Pilot1 = new Pilot("Müller", "Max")
Pilot Pilot2 = new Pilot("Meier", "Hans");
CockpitTeam = new Mitarbeiterzuordnung<Pilot, Pilot>(Pilot1, Pilot2); // OK!
Passagier Pass1 = new Passagier("Schwichtenberg", "Holger")
' Fehler: CockpitTeam = new Mitarbeiterzuordnung<Pilot, Pilot>(Pilot1, Pass1);
```

24.3 Einschränkungen für generische Typparameter (Generic Constraints)

Ein Problem verbleibt bei der Nutzung generischer Typen: Bei der Deklaration einer Variablen für einen generischen Typ könnte ein Entwickler (versehentlich) Typparameter angeben, für die die Klasse gar nicht vorgesehen ist, beispielsweise ein File-Objekt und ein Directory-Objekt bei der Klasse Mitarbeiterzuordnung.

```
// Das ist Unsinn:
Mitarbeiterzuordnung<System.IO.FileInfo, System.IO.DirectoryInfo> DateiTeam;
```

Um dies zu verhindern, können Bedingungen für die Typparameter (so genannte Generic Constraints) definiert werden. In Visual Basic erfolgt die Festlegung solcher Generic Constraints mit dem Schlüsselwort As hinter dem Typparameternamen in der Of-Deklaration. Nach dem As dürfen in geschweiften Klammern beliebig viele Schnittstellennamen, aber maximal ein Klassenname genannt werden, da die angegebenen Namen additiv wirken und eine Klasse maximal eine Basisklasse besitzen darf. In C# verwendet man das Schüsselwort where.

Listing: Deklaration einer generischen Klasse in C# mit Generic Constraints

```
public class Mitarbeiterzuordnung<ChefTyp, AssistentTyp> where ChefTyp:
Mitarbeiter, new()
                                                    where AssistentTyp:
Mitarbeiter, new()
{
               public ChefTyp Chef;
               public AssistentTyp Assi;
               public Mitarbeiterzuordnung(ChefTyp Chef, AssistentTyp Assi,
de.WWWings.Flug flug)
               {
                      this.Chef = Chef;
                      this.Assi = Assi;
               }
}
```

In Generic Constraints sind folgende Angaben erlaubt:

- eine oder mehrere Schnittstellen
- eine Basisklasse
- Schlüsselwort new (steht für Typen mit parameterlosem Konstruktor)
- Schlüsselwort class (steht für Referenztypen)
- Schlüsselwort structure (steht für Wertetypen)

24.4 Kovarianz für Typparameter

In C# 4.0 hat Microsoft die sogenannte Kovarianz für generische Typen eingeführt. Sie erlaubt es, dass bei einem Typparameter anstelle der eigentlich in einem Methodenparameter genannten Klasse auch eine abgeleitete Klasse übergeben werden kann. Dies deklariert der Entwickler einer generischen Schnittstelle mit dem Schlüsselwort out vor dem Typparameter.

Den Typparameter der Schnittstelle IEnumerable<T> hat Microsoft bereits so deklariert in der .NET-Klassenbibliothek:

```
public interface IEnumerable<out T> : IEnumerable
 { … }
```

In dem folgenden Listing wird eine Klasse Person implementiert und zwei davon abgeleitete Klassen Professor und Student. Danach werden drei generischen Listen mit der Klasse List<T> erzeugt:

- Eine Liste nur mit Professoren
- Eine Liste nur mit Studenten
- Eine Liste mit Professoren und Studenten, die aus den ersten beiden Listen mit AddRange() zusammengesetzt wird.

Danach werden die drei Listen mit der Methode Print() ausgeben. Print() erwartet als zweiten Parameter IEnumerable<Person>.

Die Kovarianz von IEnumerable wirkt hier in zwei Fällen:

- Das von Microsoft implementierte AddRange() auf List<Person> erwartet IEnumerable<Person>. Dank der von Microsoft deklarierten Kovarianz funktioniert auch die Übergabe einer List<Student> und List<Professor>.
- Das selbst implementierte Print() erwartet IEnumerable<Person>. Dank der von Microsoft deklarierten Kovarianz funktioniert auch die Übergabe einer List<Student> und List<Professor>.

Listing: Kovarianz

```
class Person
 {
  public int ID { get; set; }
  public string Name { get; set; }
 }

 class Professor : Person
 {
  public string Fachbereich { get; set; }
 }

 class Student : Person
 {
  public int MatrikelNummer { get; set; }
 }

 class CollectionVarianzDemo_Uni
 {

 public static void Run()
  {
   var hh = new Professor() { ID = 1, Name = "Harald Hastig", Fachbereich =
"Physik" };
   var tl = new Professor() { ID = 2, Name = "Theodor Langweilig", Fachbereich =
"Mathematik" };
   var hs = new Student() { ID = 2, Name = "Hans Streber", MatrikelNummer=123456
};
   var mf = new Student() { ID = 2, Name = "Max Faul", MatrikelNummer = 567890 };

   var ProfListe = new List<Professor>() { hh, tl };
   var StudentenListe = new List<Student>() { hs, mf };
   var AlleUniAngeoerigen = new List<Person>();
```

```
   AlleUniAngeoerigen.AddRange(ProfListe);
   AlleUniAngeoerigen.AddRange(AlleUniAngeoerigen);

   Print("Alle", AlleUniAngeoerigen);
   Print("Professoren", ProfListe); // möglich Dank Kovarianz für IEnumerable<T>
   Print("Studenten", StudentenListe); // möglich Dank Kovarianz für
IEnumerable<T>

  }

  // Kovarianz für IEnumerable<T>; geht nicht mit List<Person>
  public static void Print(string headline, IEnumerable<Person> personen)
  {
   CUI.Headline(headline);
   foreach (var p in personen)
   {
    Console.WriteLine(p.GetType().Name + " #" + p.ID + " heißt " + p.Name);
   }
  }
 }
```

Kovarianz für generische Typparameter wird in Schnittstellendefinitionen festgelegt. Die Kovarianz bezieht sich dann aber auch nur auf die Schnittstellen. Klassen, die diese Schnittstelle implementieren, erhalten nicht diese Kovarianz. Daher kann man in obigem Beispiel bei der Methode Print() den zweiten Parameter nicht mit List<Person> deklarieren, auch wenn List<T> die Schnittstelle IEnumerable<T> implementiert.

Ein zweites Kovarianz-Beispiel zeigt das folgende Listing mit primitiven Typen: Hier kann die Methode Print(IEnumerable<object> c) auch eine List<string> ausgeben.

Listing: Kovarianz

```
/// <summary>
 /// Kontra-Varianz bei Collections
 /// </summary>
 class CollectionVarianzDemo_ObjectString
 {
  /// <summary>
  /// Diese Methode erwartet eine Menge von Objekten
  /// </summary>
  public static void Print(IEnumerable<object> c)
  { Console.WriteLine("Anzahl: " + c.Count()); }

  public static void Run()
  {
   List<string> Namen = new List<string> { "Müller", "Meier", "Schulze" };
   // Die Methode erhält eine Menge von Strings
   // Bisher war das nicht möglich, weil Enumerable<T>
   // nicht Kontra-Varianz untersützte!
   Print(Namen);
  }
```

```
}
```

24.5 Generische Mathematik

Generische Mathematik umfasst eine Reihe von Schnittstellen in .NET im Basisklassennamensraum `System.Numerics`, die es erlauben, mathematische Operationen so zu implementieren, dass sie für beliebige Zahlentypen (Ganzzahlen und gebrochene Zahlen beliebiger Bit-Länge funktionieren).

Die in .NET 6.0 als experimentelles Feature [*https://devblogs.microsoft.com/dotnet/preview-features-in-net-6-generic-math*] enthaltene generischen Mathematikoperationen (INumber<T>, INumberBase<T>, IComparisonOperators<T, T>, IAdditionOperators<T, T, T>, IMultiplyOperators<T, T, T>, ISubtractionOperators<T, T, T> usw.) haben seit .NET 7.0 die Produktionsreife erlangt.

Das nächste Listing zeigt ein aussagekräftiges Beispiel für eine generische mathematische Berechnung in der Mehode Calc() und ein generisches Extrahieren einer Zahl aus einer Zeichenkette in ParseNumber().

Diese beiden generischen mathematischen Methoden werden in der Methode Run() mit vielen verschiedenen Ganz- und Fließkommazahlentypen getestet u.a. mit den .NET 7.0 neu eingeführten Zahlentypen System.Int128 (Ganzzahl, 16 Bytes) und System.Half (Fließkommazahl, 2 Bytes) zum Einsatz.

Listing: Generische Mathematik

```
using System.Globalization;
using System.Numerics;

namespace CS11;

public class CS11_GenericMath
{

 /// <summary>
 /// Generische mathematische Berechnung
 /// </summary>
 T Calc<T>(T x, T y)
  where T : INumber<T> // INumber<T> ist ein neues Interface mit static abstract
Members!
 {
  Console.WriteLine($"Calc {x.GetType().ToString()}/{y.GetType().ToString()}");
  if (x == T.Zero || y <= T.Zero) return T.One;
  return (x + y) * T.CreateChecked(42.24);
 }

 /// <summary>
 /// Generisches Konvertieren einer Zeichenkette in einen beliebigen Zahlentyp
 /// </summary>
 T ParseNumber<T>(string s)
     where T : IParsable<T> // IParsable<T> ist ein neues Interface mit static ab
stract Members!
 {
  return T.Parse(s, CultureInfo.InvariantCulture);
 }
```

```
 public void Run()
 {
  CUI.H2("Calc mit 1 und 2");
  Console.WriteLine($"Ergebnis mit System.Byte: {Calc((byte)1, (byte)2)}"); // 12
6
  Console.WriteLine($"Ergebnis mit System.Int32: {Calc(1, 2)}"); // 126
  Console.WriteLine($"Ergebnis mit System.Int128: {Calc((Int128)1, (Int128)2)}");
 // 126
  Console.WriteLine($"Ergebnis mit System.Single: {Calc((Single)1.0, (Single)2.0)
}"); // 126,72
  Console.WriteLine($"Ergebnis mit System.Double: {Calc(1.0d, 2.0d)}"); // 126,72
  Console.WriteLine($"Ergebnis mit System.Decimal: {Calc(1.0m, 2.0m)}"); // 126,7
20
  Console.WriteLine($"Ergebnis mit System.Half: {Calc((Half)1.0m, (Half)2.0m)}");
 // 126,75

  CUI.H2("ParseNumber 1.00 und 2.00");
  var x = ParseNumber<float>("1.00");
  var y = ParseNumber<float>("2.00");

  Console.WriteLine($"Ergebnis mit System.Single: {Calc(x, y)}"); // 3,6000001
  Console.WriteLine($"Ergebnis mit System.Int32: {Calc(0, 1)}"); // 1
 }
}
```

Der Beitrag der Programmiersprache C# ist an dieser Stelle die Möglichkeit, statische abstrakte Mitglieder in Schnittstellen zu definieren (was seit C# 10.0 experimentell möglich war und seit C# 11.0 offiziell zur Sprachsyntax gehört). Diesen Modifizierer verwendet Microsoft in den Basisklassen wie `INumberBase<T>`.

Listing: Ausschnitt aus INumberBase<T>

```
public interface INumberBase<TSelf>
        : IAdditionOperators<TSelf, TSelf, TSelf>,
          IAdditiveIdentity<TSelf, TSelf>,
          IDecrementOperators<TSelf>,
          IDivisionOperators<TSelf, TSelf, TSelf>,
          IEquatable<TSelf>,
          IEqualityOperators<TSelf, TSelf, bool>,
          IIncrementOperators<TSelf>,
          IMultiplicativeIdentity<TSelf, TSelf>,
          IMultiplyOperators<TSelf, TSelf, TSelf>,
          ISpanFormattable,
          ISpanParsable<TSelf>,
          ISubtractionOperators<TSelf, TSelf, TSelf>,
          IUnaryPlusOperators<TSelf, TSelf>,
          IUnaryNegationOperators<TSelf, TSelf>
        where TSelf : INumberBase<TSelf>?
    {
/// <summary>Gets the value <c>1</c> for the type.</summary>
static abstract TSelf One { get; }

/// <summary>Gets the value <c>0</c> for the type.</summary>
static abstract TSelf Zero { get; }
…
```

```
/// <summary>Tries to parses a string into a value.</summary>
static abstract bool TryParse([NotNullWhen(true)] string? s, NumberStyles style,
IFormatProvider? provider, out TSelf result);
}
```

25 Objektmengen (Arrays und Collections)

Es gibt drei Arten von Objektmengen in C# und Visual Basic .NET:

- Einfache Arrays (typisiert)
- Untypisierte Objektmengen
- Typisierte Objektmengen

25.1 Einfache Arrays

Einfache Arrays sind Instanzen der Klasse System.Array. Alle Arrays sind nun dynamisch bezüglich der Größe, jedoch muss man sie explizit erweitern. Die Anzahl der Dimensionen muss bei der Deklaration festgelegt werden.

> **Tipp:** Die Handhabung der Objektmengen aus dem Namensraum System.Collections ist einfacher als die Verwendung von Arrays. Jedoch erwarten einige Methoden in der .NET-Klassenbibliothek Arrays als Parameter. Man kann aber alle Objektmengen in Arrays umwandeln und so mit Objektmengen arbeiten bis zur Parameterübergabe.

Während man in Visual Basic .NET Arrays mit runden Klammern kennzeichnet, kommen in C# eckige Klammern zum Einsatz. Die Initialisierung erfolgt ebenso wie in Visual Basic .NET mit geschweiften Klammern. In .NET-Arrays beginnt die Zählung der Elemente immer bei 0. Einen wichtigen Unterschied gibt es jedoch zwischen Visual Basic .NET und C#: In C# ist in der Deklaration die Anzahl der Elemente zu nennen, in Visual Basic .NET der höchste Index (also Anzahl – 1). Erlaubte und gleichwertige Deklarationen sind:

```
byte[] lottozahlen1 = new byte[7] { 23, 48, 3, 19, 20, 6, 9 };
byte[] lottozahlen2 = new byte[] { 23, 48, 3, 19, 20, 6, 9 };
byte[] lottozahlen3 = { 23, 48, 3, 19, 20, 6, 9 };
```

Seit C# 12.0 gibt es eine alternative Syntax mit eckigen Klammern (siehe Unterkapitel zu "Collection Expression"):

```
byte[] lottozahlen3 = [23, 48, 3, 19, 20, 6, 9];
```

Microsoft empfiehlt in Coderegel IDE0300 *https://learn.microsoft.com/de-de/dotnet/fundamentals/code-analysis/style-rules/ide0300* den Einsatz dieser neuen Syntax. Dies ist aber keineswegs eine Pflicht!

> **Tipp:** Da es für die VB.NET-Schlüsselwörter ReDim und Preserve kein Äquivalent in C# gibt, muss man in C# auf die .NET-Klassenbibliothek zurückgreifen:
>
> Array.Resize<byte>(ref lottozahlen3, 20);

25.2 Untypisierte Collections

Neben den einfachen Arrays kennt .NET das Konzept der Collections im Namensraum System.Collections, die einfacher zu bedienen bzw. mächtiger sind.

Ursprünglich gab es in .NET Framework 1.0 und 1.1 nur untypisierte Objektmengen wie System.Collections.ArrayList und System.Collections.HashTable. Hier konnte man jeweils ein Objekt eines beliebigen Typs aufnehmen (die Elemente der Liste wurden mit dem allgemeinen Typ System.Object verwaltet), was die Gefahr von Laufzeitfehlern barg. Dennoch wurden Klassen wie ArrayList häufig eingesetzt, da die Verwendung komfortabler als ein einfaches Array war, da man bei den Objektmengen Elemente hinzufügen und entfernen kann, ohne die Größe der Menge

explizit anpassen zu müssen. Die Objektmengen in System.Collections werden nicht durch spezielle Schlüsselwörter in den Sprachen unterstützt.

Während die ursprünglich in .NET 1.0 eingeführten Objektmengen alle untypisiert waren und dadurch konnte es Typfehler geben, hat Microsoft mit .NET 2.0 so genannte generische Objektmengen eingeführt, die typisiert sind. Sie können nur Objekte des im Typparameter genannten Typs aufnehmen.

Praxishinweis: Diese untypisierten Klassen sind seit der Einführung der typisierten Objektmengen in .NET Framework 2.0 quasi bedeutungslos, aber weiterhin auch in allen .NET-Implementierungen enthalten.

25.3 Typisierte Collections

Generische Mengentypen sind neu seit .NET Framework 2.0 (Jahr 2005) und bieten gegenüber den untypisierten Mengentypen den Vorteil, dass eine generische Objektmenge bereits zur Entwicklungszeit auf einen bestimmten Inhaltstyp geprägt werden kann, sodass der Compiler schon feststellt, wenn der Menge Objekte falschen Typs hinzugefügt werden.

Die typisierten Objektmengen (Namensraum System.Collections.Generic) basieren auf generischen Klassen. Bei den generischen Objektmengen wird durch einen Typparameter bei Deklaration bzw. Instanziierung festgelegt, was die Menge aufnehmen darf. Bei generischen Dictionaries gibt es zwei Typparameter: einen für den Schlüssel und einen für den Wert.

Beispiele:

- List<string>: Eine Liste von Zeichenketten
- Stack<int>: Eine LIFO-Struktur (First in, First out) für Ganzzahlen
- SortedList<int, Person>: Ein Verzeichnis von Personen, die über eine Zahl identifiziert werden.
- List<object>: Eine Liste beliebiger Objekte, entspricht ArrayList.

Mengentyp	**Untypisiert (System.Collection)**	**Typisiert, generisch (System.Collection.Generic), seit .NET Framework 2.0**
FIFO-Struktur (First-In-First-Out)	Queue	Queue<Typ>
FIFO-Struktur (First-In-First-Out) mit Prioritäten	–	PriorityQueue<Typ> (seit .NET 6.0)
LIFO-Struktur (Last-In-First-Out)	Stack	Stack<Typ>
Dynamische Menge für beliebige Objekte, Zugriff über Position, doppelte Elemente erlaubt	ArrayList	List<Typ>
Dynamische Menge für Bit-Werte	BitArray	–

Mengentyp	Untypisiert (System.Collection)	Typisiert, generisch (System.Collection.Generic), seit .NET Framework 2.0
Schlüssel-Wert-Paare (Zugriff nur per Schlüssel, keine doppelten Werte erlaubt)	HashTable	Dictionary<Schlüsseltyp,Werttyp>
Schlüssel-Wert-Paare (Zugriff per Schlüssel oder Index, keine doppelten Werte erlaubt)	SortedList	SortedList<Schlüsseltyp,Werttyp>
Doppelt verkettete Liste	–	LinkedList<Typ>
Schlüssel-Wert-Paare (Zugriff per Schlüssel oder Index, keine doppelten Werte erlaubt) mit speziellen Mengenoperationen (z. B. IntersectWith(), ExceptWith(), UnionWith() und IsSubsetOf())	–	HashSet<Typ> (seit .NET Framework 3.5)
Sortiertes Hashset	–	SortedSet<Typ> (seit .NET Framework 4.0)

Tabelle: Wichtige Objektmengentypen in .NET und .NET Core

Das folgende Beispiel zeigt, dass der Compiler bei untypisierten Mengentypen nicht feststellt, wenn in eine Liste von Kunden versehentlich eine Instanz der Klasse Lieferant aufgenommen wird. Für den generischen Mengentyp akzeptiert der Compiler hingegen nur Instanzen der Klasse Kunde und von ihr abgeleitete Klassen (hier: StammKunde).

Listing: Typisierte vs. untypisierte Objektmenge

```
// Untypisierter Mengentyp
System.Collections.Queue Kunden1 = new System.Collections.Queue();
Kunden1.Enqueue(new Kunde());
Kunden1.Enqueue(new StammKunde());
Kunden1.Enqueue(new Lieferant());

// Generischer Mengentyp
System.Collections.Generic.Queue<Kunde> Kunden2 = new
System.Collections.Generic.Queue<Kunde>();
Kunden2.Enqueue(new Kunde());
```

25.4 Collection Initializer

Mengen werden häufig durch die Methode Add() befüllt. C# seit 2008 und Visual Basic seit 2010 bieten hier eine verkürzte Schreibweise mit geschweiften Klammern wie bei einfachen Arrays an (Collection Initializer). Diese Verkürzung funktioniert nur, wenn es eine Add()-Methode in der Mengenklasse gibt!

Initialisierung und Verwendung einer List<string>

```
List<string> beliebteVornamen = new List<string>()
```

```
    {"Leon", "Hannah", "Lukas", "Anna", "Leonie", "Marie", "Niklas", "Sarah",
"Jan", "Laura", "Julia", "Lisa", "Kevin"};

   Console.WriteLine("Anzahl Vornamen: " + beliebteVornamen.Count);
   // Kevin ist nun doppelt, das ist nicht verboten in einer Liste
   beliebteVornamen.Add("Kevin");
   Console.WriteLine("Anzahl Vornamen: " + beliebteVornamen.Count);

   // der erste gefundene Kevin wird entfernt
   beliebteVornamen.Remove("Kevin");

   Console.WriteLine("Anzahl Vornamen: " + beliebteVornamen.Count);

   foreach (string vorname in beliebteVornamen)
   {
    Console.WriteLine(vorname);
   }

   // Das ist nicht möglich, Datentyp stimmt nicht
   //beliebteVornamen.Add(123);
   //beliebteVornamen.Add(DateTime.Now);

   // Das ist möglich, auch wenn inhaltlich unsinnig
   beliebteVornamen.Add(123.ToString());
   beliebteVornamen.Add(DateTime.Now.ToString());
```

25.5 Objektmengen-Initialisierung mit Index

Bisher schon konnte eine Initialisierung von Mengen (z.B. Arrays) mit Indexer [x] = y erfolgen. In C# 13.0 ist eine Objektmengen-Initialisierung auch mit Index vom Ende [^x] = y möglich mit dem Index-Operator ^, den es seit C# 8.0 gibt. Das folgende Listing zeigt Beispiele.

Die neue Syntax ist allerdings nur bei der Objektmengen-Initialisierung möglich, nicht bei anderen Zuweisungen.

Listing: Objektmengen-Initialisierung mit Index von vorne [x] und vom Ende [^x]

```
class Daten
 {
  public int[] Zahlen = new int[10];
 }

public void ImplicitIndexAccess()
{
 CUI.Demo(nameof(ImplicitIndexAccess));

 CUI.H2("Array-Initialisierung mit Indexer von vorne nach hinten");
 var dAlt = new Daten()
 {
  Zahlen = {
       [0] = 0,
       [1] = 1,
       [2] = 2,
       [3] = 3,
```

```
      [4] = 4,
      [5] = 5,
      [6] = 6,
      [7] = 7,
      [8] = 8,
      [9] = 9,
  }
};

foreach (var z in dAlt.Zahlen)
{
 Console.WriteLine(z);
}

CUI.H2("NEU: Array-Initialisierung mit Indexer von hinten nach vorne");
var dNeu = new Daten()
{
 Zahlen = {
       [^1] = 0,
       [^2] = 1,
       [^3] = 2,
       [^4] = 3,
       [^5] = 4,
       [^6] = 5,
       [^7] = 6,
       [^8] = 7,
       [^9] = 8,
       [^10] = 9
 }
};

foreach (var z in dNeu.Zahlen)
{
 Console.WriteLine(z);
}

// erstelle ein Array von int mit 10 Elementen
int[] array1 = new int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
Console.WriteLine(array1.Length);

// das geht nicht
//array1 = {
//       [^1] = 0,
//       [^2] = 1,
//       [^3] = 2,
//       [^4] = 3,
//       [^5] = 4,
//       [^6] = 5,
//       [^7] = 6,
//       [^8] = 7,
//       [^9] = 8,
//       [^10] = 9
// }
```

```
 foreach (var item in array1)
 {
  CUI.LI(item);
 }
}
```

25.6 Dictionary Initializer

Auch Dictionary-Klassen (Mengenklassen mit Name-Wert-Paaren) kann man in verkürzter Schreibweise erstellen. Alternative zu Aufrufen von Add() kann man wahlweise verschachtelte geschweifte Klammern verwenden oder aber innerhalb der geschweiften Klammen die Name-Wert-Zuweisung per [Name] = Wert erledigen.

Listing: Initialisierung von Dictionary-Objekten

```
// Initialisierung mit Add()
SortedDictionary<int, string> dic0 = new();
dic0.Add(10, "www.dotnet-doktor.de");
dic0.Add(21, "www.dotnetframework.de");
dic0.Add(42, "www.dotnet8.de");

// Initialisierung mit geschweiften Klammern
SortedDictionary<int, string> dic1 = new()
{
  { 10, "www.dotnet-doktor.de" },
  { 21, "www.dotnetframework.de" },
  { 42, "www.dotnet8.de" }
};

// Initialisierung mit geschweiften und eckigen Klammern (schon vor C# 12.0
möglich)
SortedDictionary<int, string> dic2 = new()
{
   [10] = "www.dotnet-doktor.de",
   [21] = "www.dotnetframework.de",
   [42] = "www.dotnet8.de"
};
```

25.7 Vereinfachte Initialisierung und Zuweisung für Mengen (Collection Expressions) (seit C# 12.0)

Eine sehr schöne syntaktische Neuerung seit C# 12.0 ist die vereinfachte Syntax für die Initialisierung von Arrays und Listen. Microsoft nannte dieses Sprachfeature ursprünglich Collection Literals, jetzt aber **Collection Expressions**.

> **Hinweis:** Collection Expressions sind bisher (Stand C# 13.0) nicht für Dictionary-Objekte möglich. Es gibt aber für die Zukunft auch eine Idee, Dictionary Expressions einzuführen, siehe *https://github.com/dotnet/csharplang/blob/main/proposals/dictionary-expressions.md*

Mit dieser neuen Syntaxform kann man die bisher sehr heterogene Initialisierungsformen von Objektmengen stark vereinheitlichen im Stil von JavaScript, also mit den Werten in eckigen Klammern, getrennt durch Kommata (siehe Tabelle).

Bisherige Initialisierung	Nun auch möglich
int[] a = new int[3] { 1, 2, 3 };	int[] a = [1,2,3];
Span<int> b = stackalloc[] { 1, 2, 3 };	Span<int> b = [1,2,3];
ImmutableArray<int> c = ImmutableArray.Create(1, 2, 3);	ImmutableArray<int> c = [1,2,3];
List<int> d = new() { 1, 2, 3 };	List<int> d = [1,2,3];
IList<int> e = new List<int>() { 1, 2, 3 };	IList<int> e = [1, 2, 3];
IEnumerable<int> f = new List<int>() { 1, 2, 3 };	IEnumerable<int> f = [1,2,3]; **Es entsteht dabei aber ein Objekt vom Typ ReadOnlyArray<int>!**

Tabelle: Variableninitialisierung mit Collection Expressions seit C# 12.0

Nicht erlaubt ist eine Initialisierung einer Variable die mit var deklariert ist, denn damit ist der Zieltyp nicht klar:

```
// nicht erlaubt
var x = [1, 2, 3]; // Error(active) CS9176 There is no target type for the collec
tion expression
```

Es gibt aber Überlegungen, dies in Zukunft zu ermöglichen und daraus (in diesem Fall) ein List<int> oder Int-Array zu machen, siehe "Natural Element Type" im Dokument *https://github.com/dotnet/csharplang/blob/main/proposals/collection-expressions-next.md*

Die Syntax mit den eckigen Klammern ist nicht nur bei der Erstinitialisierung, sondern auch bei späteren Zuweisungen von Mengen möglich:

```
List<string> sites1, sites2 = ["www.IT-Visions.de"], sites3;
sites1 = ["www.dotnetframework.de", "www.dotnet8.de", "dotnet-lexikon.de",
"www.dotnet-doktor.de"];
sites3 = []; // leere Liste
```

Mit dem Spread-Operator `..` kann man im Rahmen der Initialisierung Mengen in andere Mengen integrieren. Der Spread-Operator sorgt dafür, dass keine verschachtelte, sondern eine flache Liste entsteht!

```
// Array aus den Elementen der Arrays erstellen mit Spread Operator
string[] allSitesAsArray = [.. sites1, .. sites2, "dotnettraining.de", ..
sites3];
// Liste aus den Elementen der Arrays erstellen mit Spread Operator
List<string> allSitesAsList = [.. sites1, .. sites2, "dotnettraining.de", ..
sites3];

// Liste noch mal erweitern
allSitesAsList = [.. allSitesAsList, "powershell-schulungen.de"];

// Auflisten: 7 Sites sind nun in der Liste
foreach (var site in allSitesAsList)
{
 Console.WriteLine(site);
}
```

Es entsteht eine Menge mit diesen sieben Websites, denn neben den fünf in den Variables sites1, sites2 und sites3 enthaltenen Websites wurde noch eine zwei weitere Domainnamen hinzugefügt.

```
www.dotnetframework.de
www.dotnet8.de
dotnet-lexikon.de
www.dotnet-doktor.de
www.IT-Visions.de
dotnettraining.de
powershell-schulungen.de
```

Abbildung: Ausgabe des obigen Listings

Bei Dictionary-Objekten kann man (wie vor C# 12.0) die Initialisierung wahlweise über verschachtelte geschweifte Klammern verwenden oder aber innerhalb der geschweiften Klammen die Name-Wert-Zuweisung per [Name] = Wert erledigen.

Listing: Initialisierung von Dictionary-Objekten

```
  // Initialisierung mit Add()
  SortedDictionary<int, string> dic0 = new();
  dic0.Add(10, "www.dotnet-doktor.de");
  dic0.Add(21, "www.dotnetframework.de");
  dic0.Add(42, "www.dotnet8.de");

  // Initialisierung mit geschweiften Klammern
  SortedDictionary<int, string> dic1 = new() {
   { 10, "www.dotnet-doktor.de" },
   { 21, "www.dotnetframework.de" },
   { 42, "www.dotnet8.de" }
  };

  // Initialisierung mit geschweiften und eckigen Klammern (schon vor C# 12.0
möglich)
  SortedDictionary<int, string> dic2 = new()
  {
   [10] = "www.dotnet-doktor.de",
   [21] = "www.dotnetframework.de",
   [42] = "www.dotnet8.de"
  };
```

25.8 Typparameter

Der Typparameter kann auch ein komplexer Typ sein, z.B. die Klasse "Vorstandsmitglied".

Listing: Initialisierung einer typisierten Objektmenge in C# mit vier Objekten, davon drei als Collection Initializer

```
// Collection Initializer
List<Vorstandsmitglied> Vorstandsmitglieder = new List<Vorstandsmitglied> { HS,
HM, MM };
Vorstandsmitglieder.Add(HP);
```

Der Typparameter kann auch object sein. Generische Objektmengen werden zu untypisierten Mengen, wenn man als Typparameter object angibt. Dann ist List<T> gleichbedeutend mit ArrayList.

Listing: Eine untypisierte Liste mit der generischen Klasse List<T>

```
List<object> liste = new List<object>();
liste.Add(123);
```

```
liste.Add("Holger");
liste.Add(DateTime.Now);
liste.Add(new System.IO.FileInfo(@"c:\temp\log.txt"));
```

25.9 Indexer

Ein Indexer erlaubt einem Softwareentwickler selbst eine Klasse zu schreiben, die sich verhält wie ein Dictionary, also eine beliebige Menge an Name-Wert-Paaren speichert. Dabei sind Datentyp von Namen und Wert beliebig. Man kann für den Namen auch Zahlen verwendet, um ein klassisches Array zu ermöglichen. Indexer sind eine gute Möglichkeit, Klassen erweiterbar zu machen.

Einen Indexer deklariert man wie ein Property, aber mit dem feststehenden Ausdruck this[]:

```
public Typ this[Typ propName]
```

Danach folgen Getter und Setter:

```
  public object this[string propName]
  {
   get => _additionaldata[propName];
   set => _additionaldata[propName] = value;
  }
```

Im folgenden Beispiel wird der Indexer intern auf SortedDictionary<string, object> abgebildet. Hier sind beliebige andere Speicher denkbar.

Listing: Einsatz von Indexern

```
using ITVisions;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpSprachsyntax
{

 class FlexPerson
 {
  public int ID { get; set; }
  public string Name { get; set; }
  public DateTime Geburtstag { get; set; }

  private SortedDictionary<string, object> _additionaldata = new SortedDictionary
<string, object>();

  public object this[string propName]
  {
   get => _additionaldata[propName];
   set => _additionaldata[propName] = value;
  }
 }

 class IndexerClient
 {
```

```
  public static void Run()
  {

   CUI.Headline("Indexer Demo");
   var p = new FlexPerson()
   {
    ID = 123,
    Name = "Holger Schwichtenberg",
    // Geburtstag bleibt unbelegt :-)
    ["Ort"] = "Essen",
    ["Firma"] = "www.IT-Visions.de",
    ["Raucher"] = false
   };

   Console.WriteLine(p.ID + ": " + p.Name);
   Console.WriteLine("arbeitet bei Firma " + p["Firma"] + " in " + p["Ort"]);
  }
 }
}
```

26 Implementierungsvererbung

Anders als in C++, aber wie in Java und C# / Visual Basic ist die Mehrfachvererbung, also die gleichzeitige Ableitung einer Klasse von mehreren anderen Klassen, nicht möglich. Die Implementierungsvererbung stellt alle Attribute, Methoden und Ereignisse auch für die erbende Klasse bereit. Nicht vererbt werden jedoch die Konstruktoren. Zirkuläres Erben (class A : B … class B : A) ist nicht sinnvoll und daher auch nicht erlaubt.

Die Implementierungsvererbung wird angezeigt durch einen Doppelpunkt nach dem Klassennamen. Der Doppelpunkt dient auch der Anzeige von Schnittstellenvererbung, entspricht also sowohl dem Visual Basic .NET-Schlüsselwort Inherits als auch Implements.

Zum Dritten wird der Doppelpunkt eingesetzt, um in einem Konstruktor einen anderen Konstruktor aufzurufen. Nach dem Doppelpunkt kann auf this (aktuelle Klasse) und base (Basisklasse) Bezug genommen werden. Durch diese Syntaxform wird sichergestellt, dass der Aufruf des anderen Konstruktors immer der erste Befehl in einem Konstruktor ist. Die Anforderung, dass der Aufruf eines anderen Konstruktors der erste Befehl sein muss, existiert auch in C#; dort jedoch gibt es dafür keine spezielle Syntax, sondern die Befehlsreihenfolge wird durch den Compiler geprüft.

Sowohl auf Klassen als auch auf Mitgliederebene kann eine Klasse steuern, wie man von ihr erben kann. Im Standard kann man von einer Klasse erben, man muss es aber nicht. Auf Klassenebene bedeutet abstract (Visual Basic .NET: MustInherit), dass eine Klasse nicht direkt verwendet werden kann, sondern nur der Vererbung dient. sealed (Visual Basic .NET: NotInheritable) bedeutet, dass ein Erben nicht möglich ist.

Für Methoden gelten etwas andere Spielregeln: virtual (Visual Basic .NET: Overridable) legt fest, dass eine Unterklasse eine Methode überschreiben (also reimplementieren) darf (siehe Methode Info() im Listing). abstract (Visual Basic .NET: MustOverride) bedeutet, dass die Unterklasse die Methode überschreiben muss (abstrakte Methode). sealed (Visual Basic .NET: NotOverridable) legt fest, dass eine Methode versiegelt ist, also nicht überschrieben werden kann. Da dies die Grundeinstellung ist, müssen sealed bzw. NotOverridable nicht explizit genannt werden.

Listing: Implementierung der Klasse Person in C#

```
namespace de.WWWings
{
 public class Person
 {
  // ========== Attribute (Fields)
  public string PersonalausweisNr;
  public string Vorname;
  public string Nachname;
  //  ========== Errechnete Attribute (Properties)
  public string GanzerName
  {
   get
   {    return this.Vorname + " " + this.Nachname;   }
  }
  // ========== Konstruktoren
  public Person()  {  }
  public Person(string Nachname, string Vorname)
  {
   this.Vorname = Vorname;
   this.Nachname = Nachname;
  }
  // ========== Methoden
  public virtual void Info()
  {   Console.WriteLine("Person: " + this.GanzerName);  }
```

```
 }
}
```

Listing: Implementierung der Klasse Passagier in C#, die von Person erb

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Text;
using de.WWWings.PassagierSystem;
using de.WWWings;
#endregion

namespace de.WWWings.PassagierSystem
{
 public class Passagier : de.WWWings.Person
 {
  // ========== Klassenmitglieder
  public static de.WWWings.PassagierSystem.Passagiere Passagiere = new
Passagiere();
  // ========== Attribute (Fields)
  public de.WWWings.Fluege Fluege = new de.WWWings.Fluege();
  public readonly long PID;
  private de.WWWings.Flug _AktuellerFlug;
  //  ========== Errechnete Attribute (Properties)
  public Flug AktuellerFlug
  {
   get
   {    return this._AktuellerFlug;   }
  }
  // ========== Konstruktoren
  public Passagier(string Name, string Vorname) : base(Name, Vorname)
  {
   this.PID = Passagier.Passagiere.Add(this);
  }
  // ========== Methoden
  public void Buchen(de.WWWings.Flug flug)
  {   this.Fluege.Add(flug.FlugNr, flug);  }
  public void Buchen(string Flugnummer)
  {
   if (!(Flug.Fluege.ContainsKey(Flugnummer)))
   {
    throw new de.NETFly.PassagierSystem.FalscheFlugnummer(this.PID + "/" +
Flugnummer);
   }
   else
   {    this.Buchen(de.WWWings.Flug.Fluege[Flugnummer]);   }
  }
  public Flug CheckIn(string Flugnummer)
  {
   if (!(this.Fluege.ContainsKey(Flugnummer)))
   {
    throw new de.NETFly.PassagierSystem.PassagierNichtAufFlugGebucht(this.PID +
"/" + Flugnummer);
   }
   else
   {    return this.Fluege[Flugnummer];   }
  }
  public override void Info()
  {
  Console.WriteLine("Passagier: " + this.GanzerName);
  }
 }
}
```

27 Schnittstellen (Interfaces)

Während .NET nur die einfache Implementierungsvererbung unterstützt, gibt es Mehrfachvererbung für Schnittstellen, d. h., eine Klasse kann optional eine oder mehrere Schnittstellen implementieren. Eine Schnittstelle kann auch von mehreren anderen Schnittstellen erben.

27.1 Deklaration einer Schnittstelle

Eine Schnittstelle wird in C# durch einen interface-Block deklariert und darf sowohl Attribute (Properties, aber keine Fields!) als auch Methoden enthalten. Konstruktoren sind nicht erlaubt. Modifizierer hinsichtlich der Sichtbarkeit (public, protected, private, private protected etc.) sind ebenfalls nicht erlaubt.

Ausnahme: Standardimplementierungen für Methoden seit C# 8.0, siehe weitere Unterkapitel.

Listing: Definition der Schnittstelle IPilot in C#

```
interface IPilot
 {
  // ====== Attribute
  DateTime FlugscheinSeit { get; set; }
  string FlugscheinTyp { get; set; }
  long Flugstunden { get; set; }
  // ====== Methoden
  void FlugZuweisen(de.WWWings.Flug Flug);
 }
}
```

Listing: Definition der Schnittstelle IPerson in C#

```
interface IPerson
 {
  // ====== Attribute
  string Vorname { get; set; }
  string Name { get; set; }
  long ID { get; set; }
  // ====== Methoden
  void Print();
 }
}
```

27.2 Verwendung von Schnittstellen

Eine Klasse zeigt durch einen Doppelpunkt hinter dem Namen an, dass sie eine Schnittstelle implementieren will.

```
public class Pilot : IPilot
```

Während immer nur eine Implementierungsvererbung möglich ist, können in einer Klasse mehrere Schnittstellen realisiert werden:

```
public class Pilot : IPilot, IPerson
```

> **Hinweis:** Strukturen, die immer auf dem Stack leben (Schlüsselwort ref struct), konnten vor C# 13.0 keine Schnittstellen realisieren.

Eine Klasse kann gleichzeitig eine Implementierungsvererbung und eine Schnittstellenimplementierung mit dem Doppelpunkt angeben.

```
public class Pilot : Mitarbeiter, IPilot
```

Eine Klasse kann gleichzeitig eine Implementierungsvererbung und mehrere Schnittstellenimplementierung mit dem Doppelpunkt angeben.

```
public class Pilot : Mitarbeiter, IPilot, IPerson
```

Hinweis: Der Compiler unterscheidet dabei automatisch, ob der Bezeichner nach dem Doppelpunkt eine Klasse oder eine Schnittstelle ist.

27.3 Standardimplementierungen in Schnittstellen

Seit C# 8.0 ist in Schnittstellen erlaubt, was es in der Programmiersprache Java auch schon seit Version 8 (erschienen im Jahr 2014) gibt: Schnittstellen dürfen nun auch Implementierungen enthalten (Default Interface Members). Diese Implementierungen werden automatisch an alle Klassen weitergegeben, die die Schnittstelle verwenden.

Hinweis: Standardimplementierungen in Schnittstellen funktionieren nur in .NET Core seit Version 3.0. Sie werden nicht unterstützt im klassischen .NET Framework. Es kommt zum Kompilierungsfehler: "CS8701 Target runtime doesn't support default interface implementation."

Praxistipp: Das Einsatzgebiet dieser Sprachfunktion ist die Weiterentwicklung von Schnittstellen (Interface Evolution) für bereits bestehende Klassen, ohne diese Klassen ändern zu müssen. In der Vergangenheit hat man Erweiterungsmethoden für diesen Zweck eingesetzt, vgl. die Erweiterungsmethoden wie Where(), GroupBy() und Select() für die Schnittstelle IEnumerable<T>, die in .NET Framework 3.5 eingeführt wurden.

27.3.1 Realisierung einer Standardimplementierung in einer Schnittstelle

Die Standardimplementierungen in Schnittstellen erfolgen syntaktisch wie die Implementierungen von Methoden in Klassen auch, also mit Sichtbarkeitsmodifizierer (private, protected, internal, public, virtual, abstract, sealed, static, extern und partial) und einem Codeblock in geschweiften Klammern (Block Body) oder einem Lambda-Ausdruck (Expression Body). Im Standard sind die Implementierung virtual, daher auch der alternative Name für Standardimplementierungen in Interfaces: Virtual Extension Methods.

Neben Instanzmethoden können Schnittstellen auch statische Methoden sowie statische Properties und Fields enthalten.

27.3.2 Einfaches Beispiel

Gegeben ist folgende Schnittstelle ILogger.

Listing: Erste Version der Schnittstelle

```
interface ILogger
 {
  string Prefix { get; set; }
  long Count { get; set; }
  // Methode ohne Implementierung
  void Log(LogLevel level, string message);
 }
```

Dazu passend die Implementierung dieser Schnittstelle in der Klasse ConsoleLogger.

Listing: Klasse, die Schnittstelle realisiert

```
class ConsoleLogger : ILogger
 {

  public string Prefix { get; set; } = "LOG:";
  public long Count { get; set; } = 0;
  public void Log(LogLevel level, string message)
  {
   Count++;
   if (level == LogLevel.Info) Console.ForegroundColor = ConsoleColor.White;
   if (level == LogLevel.Warning) Console.ForegroundColor = ConsoleColor.Yellow;
   if (level == LogLevel.Error) Console.ForegroundColor = ConsoleColor.Red;

   Console.WriteLine($"{Prefix} {Count:000} {level}: {message}");
   Console.ResetColor();
  }
 }
```

Diese Klasse ConsoleLogger kann man wie folgt nutzen:

Listing: Erste Version des Nutzers der Klasse

```
  public static void Run()
  {
   ILogger l = new ConsoleLogger();
   l.Log(LogLevel.Info, "C# 8.0 läuft!");
  }
```

Nun könnte man später auf die Idee kommen, dass auch die direkte Übergabe eines Exception-Objekts an die Logger-Klasse eine gute Idee wäre, um im Fehlerfall etwas Programmcode einzusparen. Mit den neuen Standardimplementierungen kann man dies nachträglich realisieren, indem man die Schnittstelle ILogger erweitert.

Listing: Zweite Version der Schnittstelle

```
interface ILogger
 {
  string Prefix { get; set; }
  long Count { get; set; }
  // Methode ohne Implementierung
  void Log(LogLevel level, string message);

  // Methode mit Implementierung mit Block Body
  public void Log(Exception ex)
  {
   Log(LogLevel.Error, ex.Message);
  }

  // Methode mit Implementierung mit Expression Body
  public void LogDetails(Exception ex)
=> Log(LogLevel.Error, ex.ToString());
 }
```

Die Klasse ConsoleLogger muss man nicht verändern. Dennoch stehen die neuen Komfortfunktionen den Nutzern der Klasse nun zur Verfügung.

Listing: Zweite Version des Nutzers der Klasse

```
  public static void Run()
```

```
  {
   ILogger l = new ConsoleLogger();

   l.Log(LogLevel.Info, "C# 8.0 läuft!");

   var ex = new ApplicationException("Ein Test-Fehler!");
   l.Log(ex);
   l.LogDetails(ex);
  }
```

Zu beachten ist, dass die Methoden Log(Exception) und LogDetails(Exception) auf der Variablen l nur zugänglich sind, weil die Variable auf ILogger und nicht auf ConsoleLogger typisiert wurde.

27.3.3 Überschreiben der Implementierung

Eine Klasse, die eine Schnittstelle mit Implementierung realisiert, kann jede der implementierten Methoden auch wieder anders realisieren, also überschreiben. In der folgenden Variante werden Exception-Objekte nicht als Error, sondern als Warnung ausgegeben.

Listing: Überschreiben der Standardimplementierung einer Schnittstelle in der Klasse

```
class ConsoleLogger : ILogger
 {
  public ConsoleLogger()
  {
   ILogger.Prefix = "LOG: ";
  }

  public void Log(LogLevel level, string message)
  {
   // verwendet statische Properties und Methoden der Schnittstelle
   if (ILogger.Count == 0) Console.WriteLine("Protokoll beginnt: " +
DateTime.Now);
   ILogger.Count++;

   if (level == LogLevel.Info) Console.ForegroundColor = ConsoleColor.White;
   if (level == LogLevel.Warning) Console.ForegroundColor = ConsoleColor.Yellow;
   if (level == LogLevel.Error) Console.ForegroundColor = ConsoleColor.Red;

   Console.WriteLine(ILogger.GetLogText(level, message));
   Console.ResetColor();
  }

  // Klasse kann Implementierung überschreiben!
  public void Log(Exception ex)
  {
   Log(LogLevel.Warning, ex.Message);
  }
 }
```

27.3.4 Komplexeres Beispiel

In dem komplexeren Beispiel wird wieder eine ILogger-Schnittstelle geschrieben, dieses Mal aber auch mit statischen Properties und statischen Methoden, die Funktionen für die Implementierung anbieten. Dies ist kein Beispiel für Interface Evolution, denn die Klasse ConsoleLogger greift in

der Log()-Implementierung bewusst auf Implementierungen (ILogger.Count und ILogger.GetLogText()) der Basisschnittstelle zurück.

Hinweise: Die Klasse kann nur auf Implementierungen der Basisschnittstelle zurückgreifen, wenn diese als static deklariert sind. Bei der Verwendung muss der Name der Schnittstelle (ILogger) vorangestellt werden, also ILogger.Count. Diese Verzahnung zwischen Klasse und Schnittstelle ist möglich in C# 8.0, man sollte aber überdenken, ob dies nicht besser ein Anwendungsfall für abstrakte Klassen ist, die es seit C# 1.0 gibt.

Listing: Komplexeres Beispiel für Standardimplementierungen

```
using ITVisions;
using System;

/// <summary>
/// Standardimplementierungen für Methoden in Schnittstellen, komplexeres
Beispiel
/// </summary>
class InterfacesDemo
 {

  public static void Run()
  {
   CUI.MainHeadline("Standardimplementierungen für Methoden in Schnittstellen
(komplexeres Beispiel)");
   ILogger l = new ConsoleLogger();

   l.Log(LogLevel.Info, "C# 8.0 läuft!");

   var ex = new ApplicationException("Ein Test-Fehler!");
   l.Log(ex);
   l.LogDetails(ex);
  }
 }

 enum LogLevel { Info, Warning, Error }

 interface ILogger
 {
  // Methode ohne Implementierung
  void Log(LogLevel level, string message);

  // Methode mit Implementierung mit Block Body
  public void Log(Exception ex)
  {
   Log(LogLevel.Error, ex.Message);
  }

  // Methode mit Implementierung mit Expression Body
  public void LogDetails(Exception ex)
=> Log(LogLevel.Error, ex.ToString());

  // statische Methode  mit Implementierung
  protected static string GetLogText(LogLevel level, string message)
  {
```

```
    return $"{Prefix}{ILogger.Count:000} {level}: {message}";
   }

   // Properties mit Implementierung
   public static string Prefix { get; set; }
   public static int Count { get; set; } = 0;
  }

  class ConsoleLogger : ILogger
  {
   public ConsoleLogger()
   {
    ILogger.Prefix = "LOG: ";
   }

   public void Log(LogLevel level, string message)
   {
    // verwendet statische Properties und Methoden der Schnittstelle
    if (ILogger.Count == 0) Console.WriteLine("Protokoll beginnt: " +
DateTime.Now);
    ILogger.Count++;

    if (level == LogLevel.Info) Console.ForegroundColor = ConsoleColor.White;
    if (level == LogLevel.Warning) Console.ForegroundColor = ConsoleColor.Yellow;
    if (level == LogLevel.Error) Console.ForegroundColor = ConsoleColor.Red;

    Console.WriteLine(ILogger.GetLogText(level, message));
    Console.ResetColor();
   }
  }
}
```

27.4 Statische abstrakte Properties und Methoden in Schnittstellen

Seit C# 11.0 sind in Schnittstellen Deklarationen von Properties und Methoden mit static abstract und static virtual erlaubt (in C# 10.0 war dies schon experimentell möglich).

Beispiel: Es gibt zwei Schnittstellen. IObjectWithID gibt ein statisches Property vom Typ Integer mit Namen ID vor. Die darauf aufbauende Schnittstelle IAbc gibt drei weitere Mitglieder vor:

- Ein formale abstrakte Instanzmethode GetA()
- Eine statische Methode GetB() mit Implementierung
- Eine statische abstrakte Methode GetC()

```
interface IObjectWithID
{
 static abstract int ID { get; set; } // NEU
}

interface IAbc : IObjectWithID
{
 string GetA();
```

```
 static string GetB() => "B";
 static abstract string GetC(); // NEU
}
```

Dazu zeigt folgende Implementierung der Klasse Abc auf Basis von Schnittstelle IAbc, dass man die als static abstract deklarierten Mitglieder nun als statische Mitglieder implementieren muss:

```
class Abc : IAbc
{
 #region Vorgaben der Interfaces
 public string GetA() => "A"; // muss nicht-
statische Implementierung für GetA() liefern
 public static string GetC() => "C"; // muss statische Implementierung für GetC()
 liefern
 public static int ID { get; set; } // muss statische Implementierung für Propery
 ID liefern
 #endregion

 #region zusätzliche Properties
 public static string Text1 { get; set; } = "ABC"; // zusätzliches statisches Mit
glied, nicht aus Interface
 public string Text2 { get; set; } = "ABC"; // zusätzliches Instanzmitglied, nich
t aus Interface
 #endregion
}
```

Dann sind diese Verwendungen möglich:

```
class AbcClient
{
 public static void Run()
 {
  var obj = new Abc();
  Console.WriteLine(obj.GetA()); // Instanzmitglied
  Console.WriteLine(IAbc.GetB()); // statisches Mitglied direkt im Interface
  Console.WriteLine(Abc.ID); // statisches Mitglied - Nutzung via Klassenname
  Console.WriteLine(Abc.GetC()); // statisches Mitglied - Nutzung via Klassenname
  Console.WriteLine(Abc.Text1); // statisches Mitglied
  Console.WriteLine(obj.Text2); // Instanzmitglied
 }
}
```

Randbemerkung: An diesem Sprachfeature hat Microsoft laut Aussage von Microsoft Program Manager Mads Torgersen mehr als 10 Jahre gearbeitet. Erst die Möglichkeit, die Runtime von .NET zu verändern im modernen .NET hat die Umsetzung dann zur Produktreife gebracht (Quelle: .NET Conf 2022, 9.11.2022).

Veränderungen der Runtime wurden im klassischen .NET Framework wegen möglicher Breaking Changes nicht oder nur in kleinen Dosen umgesetzt.

Dieses Sprachfeatures funktioniert nicht in älteren .NET-Versionen. Es kommt die Fehlermeldung "Target runtime doesn't support static abstract members in interfaces.".

28 Namensräume (Namespaces)

Namensräume dienen der hierarchischen Benennung von Typen (Klassen, Strukturen und Enumerationen).

Typen werden in .NET nicht mehr wie in COM durch GUIDs, sondern durch Zeichenketten eindeutig benannt. Diese Zeichenketten sind hierarchische Namen, die aus einem Namensraum (engl. Namespace) und einem Typnamen bestehen. Ein Namensraum kann aus mehreren Hierarchieebenen bestehen. Zur Bildung eines voll qualifizierten .NET-Typnamens werden sowohl Namensraum und Typname als auch die Ebenen innerhalb eines Namensraums durch Punkte getrennt. Über alle Namensräume hinweg kann der Typname mehrfach vorkommen, vergleichbar mit gleichnamigen Dateien in verschiedenen Ordnern in einem Dateisystem.

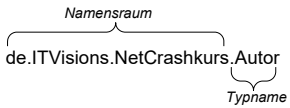


Abbildung: Beispiel für einen voll qualifizierten .NET-Typnamen

28.1 Softwarekomponenten versus Namensräume

Eine einzelne .NET-Softwarekomponente kann beliebig viele Namensräume umfassen und ein Namensraum kann sich über beliebig viele Softwarekomponenten erstrecken. Die Auswahl der Typen, die zu einem Namensraum gehören, sollte nach logischen oder funktionellen Prinzipien erfolgen. Im Gegensatz dazu sollte die Zusammenfassung von Typen zu einer Softwarekomponente gemäß den Bedürfnissen zur Verbreitung der Klassen (Deployment) erfolgen.

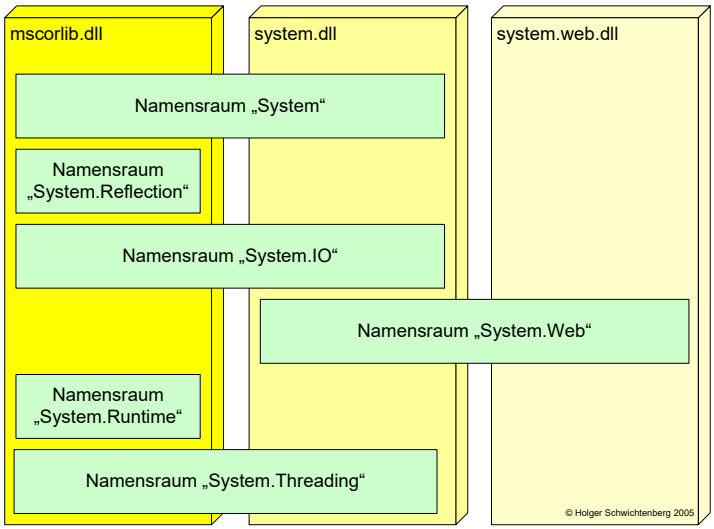


Abbildung: Namensräume versus Softwarekomponenten am Beispiel ausgewählter Teile der .NET-Klassenbibliothek

In .NET können beliebig viele Namensraumhierarchien parallel existieren. Es gibt keinen gemeinsamen Wurzelnamensraum und keine zentrale Registrierung der Namensräume. Die .NET-Klassenbibliothek besitzt zwei Wurzelnamensräume, System und Microsoft.

Da kein globales Verzeichnis aller Namensräume auf einem System existiert, gibt es nicht wie in COM eine einfache Möglichkeit, alle auf einem System vorhandenen .NET-Klassen aufzulisten. Möglich wäre aber die Suche nach .dll- bzw. .exe-Dateien im Dateisystem und eine Einzelprüfung dieser Dateien daraufhin, ob sie .NET-Typen enthalten.

28.2 Vergabe der Namensraumbezeichner

Da keine zentrale Stelle existiert, die die Namensraumbezeichner vergibt, besteht natürlich grundsätzlich die Gefahr, dass zwei Softwareentwickler gleiche Typnamen festlegen. Im CLI-Standard (CLI = Common Language Infrastructure) ist daher vorgesehen, dass der Namensraum mit dem Firmennamen beginnt. Noch eindeutiger wird der Name jedoch, wenn man anstelle des Firmennamens den Internet-Domänennamen verwendet, z.B. de.ITVisions.NetCrashkurs.Autor statt ITVisions.NetCrashkurs.Autor.

Diese Konvention schützt natürlich nicht vor mutwilligen Doppelbenennungen. Für .NET-Anwendungen und -Softwarekomponenten ist deshalb vorgesehen, dass diese digital signiert werden können.

28.3 Vergabe der Typnamen

Auch für die Namensgebung von Typen in der .NET-Klassenbibliothek gibt es Regeln, die im CLI-Standard manifestiert sind. Die Namen für Klassen, Schnittstellen und Attribute sollen Substantive sein. Die Namen für Methoden und Ereignisse sollen Verben sein.

Für die Groß-/Kleinschreibung gilt grundsätzlich PascalCasing, d. h., ein Bezeichner beginnt grundsätzlich mit einem Großbuchstaben und jedes weitere Wort innerhalb des Bezeichners beginnt ebenfalls wieder mit einem Großbuchstaben. Ausnahmen gibt es lediglich für Abkürzungen, die nur aus zwei Buchstaben bestehen. Diese dürfen komplett in Großbuchstaben geschrieben werden (z.B. UI und IO). Alle anderen Abkürzungen werden entgegen ihrer normalen Schreibweise in Groß-/Kleinschreibung geschrieben (z.B. Xml, Xsd und W3c). Attribute, die geschützt (Schlüsselwort Protected) sind, und die Namen von Parametern sollen in camelCasing (Bezeichner beginnt mit einem Kleinbuchstaben, aber jedes weitere Wort innerhalb des Bezeichners beginnt mit einem Großbuchstaben) geschrieben werden.

Einige Programmiersprachen (wie beispielsweise C#) erlauben, dass sich zwei Bezeichner nur hinsichtlich der Groß- und Kleinschreibung unterscheiden können. Es wäre in C# also gültig zu definieren:

```
public class Autor
{
  public string Name;
  public string name;
}
```

Jedoch ist diese Vorgehensweise nicht CTS-konform, weil eine andere, nicht zwischen Groß- und Kleinschreibung unterscheidende (case-sensitive) Sprache diese beiden Attribute nicht unterscheiden könnte. Ein Client in Visual Basic würde nur das erste Mitglied Name sehen; das zweite name bliebe verdeckt. CTS-konform ist jedoch folgende Deklaration, weil in diesem Fall das zweite Attribut nicht nach außen angeboten wird:

```
public class Autor
{
  public string Name;
  private string name;
}
```

28.4 Namensräume deklarieren

Die Deklaration eines Namensraums dient dazu, einen Typ einem Namensraum zuzuordnen. Jeder Typ gehört nur zu genau einem Namensraum.

Die Festlegung des Namensraums für eine Klasse erfolgt in C# seit Version 1.0 durch den Code-Block namespace Name { … }. In einem Namensraum können beliebig viele Typen enthalten sein. Ein Namensraum kann sich über mehrere Dateien und auch mehrere Assemblies erstrecken. Der Namensraum muss aber zu Beginn jeder Datei in jedem Projekt erneut deklariert werden.

```
namespace de.WWWings.PassagierSystem
{
 public class Passagier : de.WWWings.Person
 { … }

 public class Buchgung
 { … }
}
```

Seit C# 10.0 gibt es alternativ zu diesem Block-Stil auch Namensraumdeklaration auf Dateiebene (engl. File-Scoped Namespace). Dabei schreibt man nur noch *namespace Name;* ohne geschweifte

Klammern. Auch hier muss der Namensraum aber zu Beginn jeder Datei in jedem Projekt erneut deklariert werden. Die Namensraumdeklaration auf Dateiebene muss vor allen Typdeklarationen erscheinen.

```
namespace de.WWWings.PassagierSystem;
public class Passagier : de.WWWings.Person
{ … }
public class Buchgung
{ … }
```

Diese Namensraumdeklaration gilt dann für die gesamte Datei. Geschweifte Klammern und Einrückung sind nicht mehr notwendig.

Hinweis: Diese in C# 10.0 eingeführte Vereinfachung basiert auf dieser Erkenntnis des C#-Entwicklungsteams: "Measuring an even broader set of millions of C# files on GitHub shows literally 99.99% of files have just one namespace in them." [*github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-01-13.md#file-scoped-namespaces*]

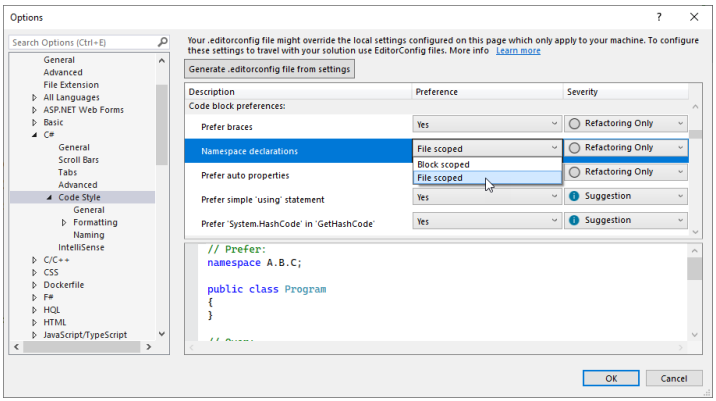


Abbildung: Einstellung der präferierten Verwendung von Namensraumdeklaration auf Dateiebene (File Scoped) oder mit geschweiften Klammern (Block Style) in den Einstellungen von Visual Studio

Hinweis: Anders als bei Visual Basic-Projekten kann man in Visual Studio für C#-Projekte in den Projekteigenschaften keinen Wurzelnamensraum definieren, der allen Namensraumdeklarationen automatisch vorangestellt wird. Der im Tag <RootNamespace> in einer .csproj-Datei wird automatisch bei neu angelegten Klassen als expliziter Namensraum in der Datei eingetragen. Nur in Blazor-Projekten bei Razor Component wird der <RootNamespace> automatisch vorangestellt.

28.5 Import von Namensräumen

Im Normalfall müssen Klassen in .NET immer mit ihrem vollen Namensraum genannt werden. Das optionale Importieren von Namensräumen hat das Ziel, einen Klassennamen mit verkürztem oder ganz ohne Namensraum zu verwenden.

Das Importieren von Namensräumen erfolgt in C# mit dem Schlüsselwort using. Dabei ist es möglich, einen Alias-Namen für einen Namensraum zu vergeben.

```
using System.Collections.Generic;
using GenCol = System.Collections.Generic;
```

Import-Anweisung	Typnutzung
Ohne	System.Collections.Generic.SortedList<string , Flug>
using System.Collections.Generic;	SortedList<string, Flug>
using GenCol = System.Collections.Generic;	GenCol.SortedList<string, Flug>

Tabelle: Beispiele für den Einsatz von Import

Hinweis: Das Schlüsselwort *using* hat in C# eine Doppelbedeutung. Es wird auch für Using-Blöcke beim IDisposable-Muster verwendet (siehe Kapitel "IDisposable / Using-Blöcke").

Seit C# 10.0 gibt es globale Namensraumimporte über **globale Using-Direktiven** (engl. Global Using Directives). Der Zusatz static (vgl. statische Methode als globale Funktionen seit C# 6.0) ist auch bei global using möglich. Ebenso sind Aliase erlaubt.

```
global using System;
global using static System.Console;
global using IS = System.Runtime.InteropServices;
```

Eine solche globale Using-Direktive gilt für alle Dateien in einem Projekt. Somit entfällt es, immer wieder zu Beginn jeder Datei den Namensraum zu importieren.

Hinweis: Ein globaler Namensraumimport darf nicht innerhalb eines mit Block-Syntax deklarierten Namenraums erfolgen (Regel CS8914: "A global using directive cannot be used in a namespace declaration.")

Ein Entwickler kann die globalen Namensraumimporte auch in eine separate Datei auslagern und die Importe damit ganz aus dem aktiven Sichtfeld verbannen. Alternativ dazu kann man Namensräume auch in der Projektdatei .csproj global importieren mit dem Tag <Using> in einer <ItemGroup>, optional auch mit dem Zusatz Static="True" für einen statischen Import (siehe Kapitel "Statische Methode als globale Funktionen"):

```
<Project Sdk="Microsoft.NET.Sdk">
 …
 <ItemGroup>
  <Using Include="System.Runtime.InteropServices" />
  <Using Include="System.Console" Static="True"/>
  <Using Include="BO" />
  <Using Include="BL" />
 </ItemGroup>

</Project>
```

Auf C# 10.0 (oder höher) basierende Projekte haben zudem eine Reihe von Namensräumen, die automatisch importiert werden und nicht mehr explizit importiert werden müssen ("Implizite Namensräume"). Welche dies sind, zeigt die folgende Abbildung abhängig vom aktiven .NET SDK.

- Microsoft.NET.SDK:
  - System
  - System.Collections.Generic
  - System.IO
  - System.Linq
  - System.Net.Http
  - System.Threading
  - System.Threading.Tasks
- Microsoft.NET.SDK.Web:
  - Everything included by Microsoft.NET.SDK, plus:
  - System.Net.Http.Json
  - Microsoft.AspNetCore.Builder
  - Microsoft.AspNetCore.Hosting
  - Microsoft.AspNetCore.Http
  - Microsoft.AspNetCore.Routing
  - Microsoft.Extensions.Configuration
  - Microsoft.Extensions.DependencyInjection
  - Microsoft.Extensions.Hosting
  - Microsoft.Extensions.Logging
- Microsoft.NET.SDK.Worker
  - Everything included by Microsoft.NET.SDK, plus:
  - Microsoft.Extensions.Configuration
  - Microsoft.Extensions.DependencyInjection
  - Microsoft.Extensions.Hosting
  - Microsoft.Extensions.Logging

Abbildung: Liste der automatischen Namensaum-Importe (Quelle: learn.microsoft.com/en-us/dotnet/core/compatibility/sdk/6.0/implicit-namespaces)

ACHTUNG: Implizite Namensräume können Probleme verursachen, wenn die Klassennamen in mehreren Namensräumen vorkommen. Wenn Sie zum Beispiel die Klasse Microsoft.Build.Utilities.Task oder MiracleList.BO.Task verwenden und dafür einen expliziten Namensraumimport einbinden, wird der C#-Compiler seit C# 10.0 meckern: "'Task' is an ambiguous reference". In diesem Fall müssen Sie entweder den Klassennamen vollständig mit Namensraum angeben oder die impliziten Namensräume deaktivieren.

Während vor .NET 6 RC1 die impliziten Namensräume im Standard automatisch aktiv waren und mit *<DisableImplicitNamespaceImports>true</DisableImplicitNamespaceImports>* erst deaktiviert werden mussten, was auch alle bestehenden Projekte betraf, hat sich Microsoft nun eines Besseren besonnen: Die impliziten Namensräume sind nur noch aktiv, wenn in der Projektdatei in einer <PropertyGroup> das Tag *<ImplicitUsings>enable</ImplicitUsings>* vorkommt. Dies ist nur bei mit .NET 6 in Visual Studio 2022 neu angelegten Projekten der Fall; ältere Projekte, die auf .NET6 hochgestuft werden, erhalten das Tag nicht.

In neuen Projekten kann man die impliziten Namensräume durch Löschen des Tags bzw. mit *<ImplicitUsings>disable</ImplicitUsings>* deaktivieren. Alternativ können Sie auch einzelne implizite Namensräume deaktivieren:

<ItemGroup>

Kommentiert [DF1]: Satzsinn bzw. Formulierung ok?

```
  <Using Remove="System.Threading.Tasks" />
</ItemGroup>
```

```
NET6ConsoleMin.csproj
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
   <!--<DisableImplicitNamespaceImports>true</DisableImplicitNamespaceImports>-->
  </PropertyGroup>

 <ItemGroup>
  <Import Remove="System.Threading.Task" />
 <Import Include="System.Runtime.InteropServices" />
 </ItemGroup>

</Project>
```

Abbildung: Implizite Namensräume in der .csproj-Datei

28.6 Verweis auf Wurzelnamensräume

Wurzelnamensräume sollten eindeutig sein. Deshalb ist es empfehlenswert, dem Namensraum die Internet-Domain voranzustellen (z.B. de.WWWings.PassagierSystem). Dabei sollte man Namensdopplungen auch für untergeordnete Namensräume vermeiden, weil es sonst unter bestimmten Bedingungen zweideutige Interpretationen einer Anweisung geben könnte. Insbesondere sollte man die Begriffe System und Microsoft vermeiden, weil damit die FCL-Namensräume verdeckt werden.

Beispiel

Wenn man »versehentlich« einen Namensraum wie de.WWWings.System definiert hat, kann man aus diesem Namensraum heraus nicht mehr auf den FCL-Namensraum System zugreifen (siehe Abbildung),

```
namespace CS20.GlobalDemo.System.IO
{
 class FileInfo
 {
  public string Name;
 }
 class FileInfoTest
 {
  public static void Run()
  {
   System.IO.FileInfo f = new System.IO.FileInfo();
   f.Name = @"c:\te
                    class CS20.GlobalDemo.System.IO.FileInfo

   global::System.IO.FileInfo f2 = new global::System.IO.FileInfo(@"c:\temp\daten.txt");
  }
 }
}
```

Abbildung: Der FCL-Namensraum System ist durch den Namensraum CS20.GlobalDemo.System verdeckt

Das Schlüsselwort global:: übernimmt seit C# 2.0 die gleiche Funktion wie global ab Visual Basic 2005: Mit diesem dem Namensraum vorangestellten Schlüsselwort adressiert man einen Wurzelnamensraum, wenn dieser durch einen untergeordneten Namensraum verdeckt ist.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace CS20.GlobalDemo.System.IO
{
        class FileInfo
        {
                public string Name;
        }
        class FileInfoTest
        {
                public static void Run()
                {
                        System.IO.FileInfo f = new System.IO.FileInfo();
                        f.Name = @"c:\test.txt";


                        global::System.IO.FileInfo f2 = new
global::System.IO.FileInfo(@"c:\temp\daten.txt");
                }
        }
}
```

29 Anonyme Typen

Neu seit C# 3.0 und Visual Basic .NET 9.0 ist, dass man Objekte ohne eine explizite Klassendefinition erzeugen kann. Solche Klassen erhalten automatisch einen Klassennamen von dem Compiler. Dieser Name ist recht kompliziert und nicht zur Verwendung durch den Entwickler gedacht.

Ein anonymer Typ entsteht in C# durch Verwendung von new ohne Klassennamen und in Visual Basic .NET durch New With.

Listing: Anonyme Typen in C# 3.0

```
// Anonyme Typen
var Fluggesellschaft1 = new { Name = "World Wide Wings",
                              Gruendungsdatum = new DateTime(2005, 01, 01),
                              Vorstand = PersonenListe };
Console.WriteLine(Fluggesellschaft1.GetType().FullName);

// 2., gleich aufgebauter anonymer Typ
var Flugzeugbauer = new { Name = "Never Come Back Airline",
                          Gruendungsdatum = new DateTime(1972, 08, 01),
                          Vorstand = new List<Person>() };
Console.WriteLine(Flugzeugbauer.GetType().FullName);

// sind die Typen gleich?
var TypenGleich = Flugzeugbauer.GetType() == Fluggesellschaft1.GetType();
Console.WriteLine("Typen gleich? " +  TypenGleich);
```

Durch die obigen Listings entsteht ein anonymer Typ mit diesem Namen:

```
<>f__AnonymousType0`3[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral
, PublicKeyToken=b77a5c561934e089],[System.DateTime, mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.Collections.Generic.Li
st`1[[NET3.SpracheCSharp.Demo_Sprachfeatures.Vorstandsmitglied, WWWings.Verschie
deneDemos, Version=0.5.0.0, Culture=neutral, PublicKeyToken=null]], mscorlib, Ve
rsion=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

Hinweis: Bei anonymen Typen ist Folgendes zu beachten:

Die Initialisierung kann mit statischen Werten oder Variablen erfolgen.

Zwei auf die o. g. Weise instanziierte Objekte gehören zur gleichen Klasse, wenn sie die gleiche Anzahl und Reihenfolge von Attributen bei der Instanziierung besitzen.

Auf diese Weise instanziierte Objekte können nicht mehr verändert werden, weil alle Property-Attribute nur für den Lesezugriff erzeugt werden.

Auf diese Weise instanziierte Objekte sind nicht serialisierbar, weil es keinen parameterlosen Standardkonstruktor gibt.

Der Name eines anonymen Typen wird bei jedem Kompilierungsvorgang neu vergeben. Man darf sich daher nicht auf das Ergebnis von GetType() verlassen.

Man kann komplexe anonyme Typen durch Verschachtelung erzeugen.

Man kann auch ein Array aus anonymen Typen bilden und – mit einem hier aus Platzgründen nicht gezeigten Trick – auch anonyme Typen in andere Objektmengen aufnehmen.

Anonyme Typen sind nur für lokale Variablen erlaubt. Sie sind nicht einsetzbar als Klassenmitglieder, Parameter von Methoden und Rückgabewerte von Methoden.

Anonyme Typen kann man seit C# 10.0 mit With-Ausdrücken klonen (siehe dazu Kapitel "Strukturen/With-Ausdrücke").

30 Operatorüberladung

Operatorüberladung bedeutet, einem der Standardoperatoren wie +, -, * und = im Zusammenhang mit selbstdefinierten Klassen eine neue Bedeutung zu geben, z.B. ein Flug-Objekt und ein Passagier-Objekt zu addieren, um daraus ein neues Objekt des Typs Buchung zu gewinnen.

> **Wichtig:** Zum Thema Operatorüberladung gibt es geteilte Meinungen. Von einigen Entwicklern wird sie geliebt wegen der Prägnanz. Von anderen Entwicklern wird sie gehasst wegen der Mehrfachbedeutung der Operatoren, die die Lesbarkeit des Programmcodes erschwert. Festzuhalten ist auf jeden Fall, dass man Operatorüberladung nicht zwingend braucht; alles was Operatorüberladung kann, kann man auch durch eine Methode mit einem sprechenden Namen ausdrücken.

C# bietet seit seiner ersten Version eine prägnante Syntax für die Definition einer Operatorüberladung.

Listing: Beispiel für Operatorüberladung in C#

```
namespace de.WWWings
{

 public partial class Flug
 {
  …

  /// <summary>
  /// Operatorüberladung für die Buchung eines Flugs durch Addition eines Flug- u
nd eines Passagier-Objekts.
  /// </summary>
  /// <param name="flug">Flugobjekt</param>
  /// <param name="pass">Passagierobjekt</param>
  /// <returns>Flugobjekt mit hinzugefügten Passagier</returns>
  public static Flug operator +(Flug flug, PassagierSystem.Passagier pass)
  {
   pass.Buchen(flug);
   return flug;
  }

 }
}
```

Seit C# 11.0 besteht auch die Möglichkeit, solch eine Operatorüberladung in einer Schnittstelle zu definieren, um eine Vorgabe bzw. Gemeinsamkeit für alle Implementierungen zu erschaffen, denn erst seit C# 11.0 ist "static abstract" in Schnittstellen erlaubt.

Eine Schnittstelle mit einer Operatorüberladung könnte so aussehen:

Listing: Operatorüberladung in einer Schnittstelle (ab C# 11.0)

```
namespace de.WWWings;

public interface IFlug<TSelf> where TSelf : IFlug<TSelf>
{
 string AbflugOrt { get; set; }
 double Auslastung { get; }
 DateTime Datum { get; set; }
 long FlugNr { get; set; }
```

```
 short FreiePlaetze { get; set; }
 bool Nichtraucherflug { get; set; }
 short Plaetze { get; set; }
 string Route { get; }
 string ZielOrt { get; set; }

 public static abstract Flug operator +(TSelf flug, de.WWWings.PassagierSystem.Pa
ssagier pass);
}
```

Hinweis: Der ein oder andere wird sich sicherlich fragen, warum die Schnittstelle generisch sein muss. Der Grund dafür ist einfach: Man will am Ende ja ein Flug-Objekt einfach mit + zu einem Passagier addieren können. Ohne die generische Implementierung könnte man nur eine Variable vom Typ IFlug zum Passagier addieren.

Die Implementierung der Klasse muss sodann um die Schnittstellenimplementierung

```
: IFlug<Flug>
```

ergänzt werden.

Listing: Beispiel für Operatorüberladung in C#

```
namespace de.WWWings
{

 public partial class Flug : IFlug<Flug>
 {
  …

  /// <summary>
  /// Operatorüberladung für die Buchung eines Flugs durch Addition eines Flug- u
nd eines Passagier-Objekts.
  /// </summary>
  /// <param name="flug">Flugobjekt</param>
  /// <param name="pass">Passagierobjekt</param>
  /// <returns>Flugobjekt mit hinzugefügten Passagier</returns>
  public static Flug operator +(Flug flug,PassagierSystem.Passagier pass)
  {
   pass.Buchen(flug);
   return flug;
  }

 }
}
```

31 Strukturen

Strukturen mit dem Schlüsselwort struct anstelle von class sind eine besondere Form von Klassen. Die .NET-Laufzeitumgebung behandelt diesen Typen als Wertetypen und verwaltet sie im Hauptspeicher auf dem Stack-Speicher (mit First-In-First-Out-Methodik) statt auf dem Heap-Speicher. Der Heap wird in .NET auch "Managed Heap" genannt.

31.1 Wertetyp versus Referenztyp

Grundsätzlich sind alle Typen in .NET echte Klassen, d.h. .NET ist also komplett objektorientiert, weil auch einfache Datentypen wie Zahlen als Objekte aufgefasst werden, auf denen man Methoden ausführen kann. So sind z.B. 5.ToString() und (123.45).ToString() gültige Ausdrücke. Klassen sind üblicherweise Referenztypen, d.h., im Stack wird ein Zeiger auf einen Speicherplatz im Heap vorgehalten.

Für einfache Datentypen ist diese Zwischenstufe jedoch sehr ineffizient. Microsoft hat daher in .NET auch Wertetypen (alias Strukturen) vorgesehen, deren Inhalt direkt auf dem Stack abgelegt werden kann.

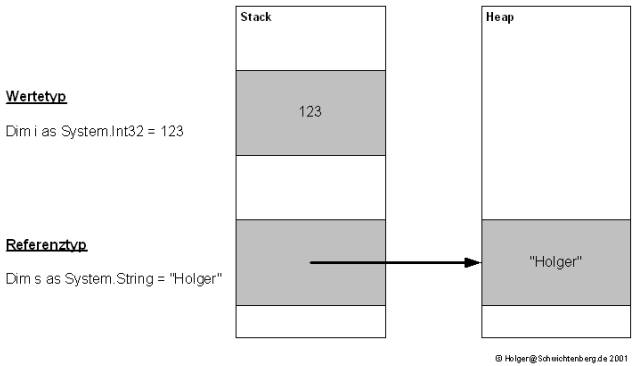


Abbildung: Wertetyp versus Referenztyp im Hauptspeicher

Auch Wertetypen sind als Klassen implementiert und können daher die gleichen Mitglieder wie Klassen besitzen. Ihre Besonderheit besteht jedoch darin, dass sie von System.ValueType erben.

Die folgende Tabelle zeigt die Unterschiede zwischen Wertetyp und Referenztyp. Besonders zu erwähnen ist noch die Klasse System.String. Diese Klasse gehört zwar zu den Referenztypen, verhält sich aber beim Kopieren wie ein Wertetyp.

	Reference Typen (Referenztyp)	**Value Type (Wertetyp/ Strukturen)**	**Nullable Value Type (Werteloser Wertetyp/ Strukturen)**
Verfügbar seit	.NET 1.0	.NET 1.0	.NET 2.0
Standard-Speicherort der Werte	Heap	Stack (können aber in einigen Fällen auch auf dem Heap leben, außer bei ref struct)	Stack (können aber in einigen Fällen auch auf dem Heap leben)
Basisklasse	Direktes oder indirektes Erben von System.Object	Direktes oder indirektes Erben von System.ValueType	Nullable<T>
C#-Sprachkonstrukt zur Definition	class seit C# 9.0 auch record seit C# 10.0 auch mit record class	struct seit C# 10.0 auch mit record struct	struct
Standardwert	null	Abhängig vom Datentyp, 0 bei Zahlen, false bei Boolean und 1.1.0001 bei DateTime	null
Setzen auf null möglich	Ja (bei Aktivierung von Nullable Reference Types ab C# 8 nur bei Verwendung von ? im Typ z.B. string?)	Nein	Möglich
Parameterloser Konstruktor	Möglich	Nicht möglich bis C# 9.0, möglich seit C# 10	Nicht möglich bis C# 9.0, möglich seit C# 10
Initialisierung von Fields und Properties mit Werten	Möglich	Nicht möglich bis C# 9.0, möglich seit C# 10	Nicht möglich bis C# 9.0, möglich seit C# 10
Vererbung von anderen Typen	Ja	Nein	Nein
Implementierung von Schnittstellen	Ja	Ja	Ja
Zirkuläre Referenzen	Ja	Nein	Nein
Abonnement von Ereignissen	Ja	Nein	Nein

	Reference Typen (Referenztyp)	**Value Type (Wertetyp/ Strukturen)**	**Nullable Value Type (Werteloser Wertetyp/ Strukturen)**
Instanziierung	Pflicht	Optional, Instanziierung führt zu Initialisierung	Optional, Instanziierung führt zu Initialisierung
Vergleich	Referenzvergleich, Bei Record-Typen: Wertvergleich	Wertvergleich	Wertvergleich
Kopie	Referenzkopie (flache Wertkopie optional mit MemberwiseClone(), tiefe Kopie muss selbst entwickelt werden)	Wertkopie	Wertkopie

Tabelle: Wertetyp versus Referenztyp

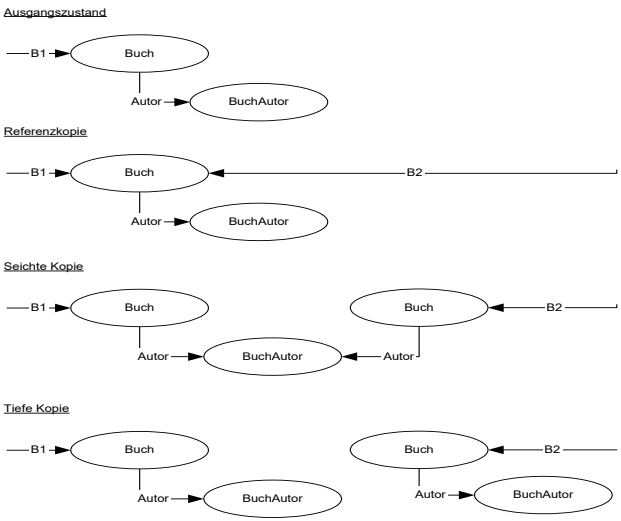


Abbildung: Typen von Objektkopien

Obige Tabelle enthält die allgemeinen Regeln, von denen es aber Ausnahmen gibt. So leben statische Variablen immer auf dem Heap. Auch wird eine Struktur, die Teil einer Klasse ist, auf dem Heap gespeichert. Auch durch das sogenannte Boxing wird eine Struktur auf den Heap gespeichert.

31.2 Deklaration von Strukturen

Eine Struktur wird in C# deklariert mit dem Sprachkonstrukt

```
struct { …. }
```

Eine Struktur kann – wie eine Klasse – Daten (in Form von Fields und Properties), Methoden, Ereignisse, Konstruktoren, Operatoren und auch eingebettete Typen enthalten.

Während Klassen in C# schon immer einen parameterlosen Konstruktor besitzen konnten, ist dies für Strukturen erst seit C# 10.0 erlaubt. Der Entwickler kann den parameterlosen Konstruktor Strukturen seit C# 10.0 selbst definieren. Alternativ erzeugt der Compiler selbst einen parameterlosen Konstruktor, wenn der Entwickler eine Initialisierung von Fields und Properties bei der Deklaration in Strukturen vornimmt. Vor C# 10.0 waren solche Initialisierung in Strukturen verboten!

Listing: Beispiel für eine Struktur

```
struct Experte
 {
  public int ID;
  public string Name { get; set; }
  public List<String> Themen { get; set; } = new List<string>(); // Diese Initial
isierung zieht nach sich, dass es einen Konstruktor in der Struktur geben muss, s
onst Fehler CS8983.

  // erlaubt
  public List<Experte> MitarbeiterTeam { get; set; } = new List<Experte>();
  // nicht erlaubt:
  //public Experte Vorgesetzter { get; set; } //  Error CS0523 Struct member 'Exp
erte.Vorgesetzter' of type 'Experte' causes a cycle in the struct layout

  struct Adresse
  {
   public string Strasse { get; set; }
   public string PLZ { get; set; }
   public string Ort { get; set; }
  }

  public Experte()
  {
   ID = 0;
   Name = "unbekannt";
  }

  public Experte(int id, string name)
  {
   ID = id;
   Name = name;
  }
```

```
 public int ThemenAnzahl { get { return this.Themen.Count; } }
 public string GetThemenString()
 {
  return String.Join(", ", this.Themen);
 }
}
```

Bis einschließlich C# 10.0 gilt, dass ein Konstruktor alle Properties, die keinen Standardwertzuweisung innerhalb der Deklaration, explizit initialisieren muss.

Abbildung: Dieser Konstruktor initialisiert das Property Name nicht, welches hier keine Vorbelegung besitzt

Ein parameterloser Konstruktor in einer Struktur musste vor C# 11.0 alle nicht in der Deklaration initialisierten Fields und Properties explizit mit einem Wert belegen, z.B.

```
public Experte()
{
 ID = 0;
 Name = "unbekannt";
}
```

Das hat sich seit C# 11.0 geändert: Die Datenmitglieder (Fields und Properties) von Strukturen müssen seit C# 11.0 in eigenen Konstruktoren nicht mehr explizit initialisiert werden, wenn diese keine Initialisierungswerte bei der Deklaration besitzen. Seit C# 11.0 werden alle nicht explizit initialisierten Felder und Properties automatisch mit ihren Standardwerten initialisiert! Microsoft nennt das Feature **Auto-Default Structs**.

Ab C# 11.0 kann also die obige Struktur auch einen parameterlosen Konstruktor ohne Code enthalten und auch Konstruktoren, die nicht alle Fields und Properties initialisieren.

Listing: Struktur in C# seit Version 11.0 mit Konstruktoren, die nicht alle Datenmitglieder initialisieren

```
struct Experte
{
 public int ID;
 public string Name { get; set; }
 public List<String> Themen { get; set; } = new List<string>();

 public Experte()
 {
 }

 public Experte(int id)
 {
  ID = id;
 }
```

```
 public Experte(int id, string name)
 {
  ID = id;
  Name = name;
 }

 public int ThemenAnzahl { get { return this.Themen.Count; } }
 public string GetThemenString()
 {
  return String.Join(", ", this.Themen);
 }
}
```

Auch in der aktuellen C#-Version gilt aber noch, dass Initialisierungen von Fields und Properties bei der Deklaration in Strukturen nur möglich sind, wenn man auch einen Konstruktor schreibt, siehe nächste Abbildung.

```
struct Experte   CS8983: A 'struct' with field initializers must include an explicitly declared constructor.
{
 public int ID = 0;
 public string Name { get; set; } = "unbekannt";
 public List<String> Themen { get; set; } = new List<string>();
}
```

Abbildung: Fehlermeldung, weil ID, Namen und Themen initialisiert sind, aber es keinen Konstruktor gibt

31.3 Verwendung von Strukturen

Das folgende Beispiel zeigt sehr eindrucksvoll den Charakter einer Struktur im Vergleich zu einer Klasse. Es wird eine Instanz der Struktur Experte erzeugt und befüllt.

Es wird eine Kopie der Instanz angelegt. Dass dies eine Wertkopie und keine Referenzkopie ist, sieht man bei der Veränderung des Namens in der ursprünglichen Variablen. Die Kopie behält den alten Wert in dem Attribut Name.

Allerdings wirkt sich das Hinzufügen eines Themas zur Eigenschaft Themen auf beide Experten aus, auf das Original und die Kopie. Das liegt daran, dass List<string> ein Referenztyp ist. Bei dem Kopieren der Struktur wird also nur die Referenz auf die Themenliste kopiert.

Listing: Deklaration und Nutzung einer Struktur

```
Experte hs = new Experte();
hs.ID = 1;
hs.Name = "Holger Schwichtenberg";
hs.Themen = new List<string>();
hs.Themen.Add(".NET");
hs.Themen.Add("Web");
hs.Themen.Add("PowerShell");
hs.Themen.Add("Data Access");

Console.WriteLine(hs.Name + " ist Experte für " + hs.ThemenAnzahl + " Themen!");

Experte hs_Klon = hs; // Wertkopie!
Console.WriteLine(hs_Klon.Name + " ist Experte für " + hs_Klon.ThemenAnzahl + " T
hemen!");
```

```
Console.WriteLine("Namensergänzung");
hs.Name = "Dr. " + hs.Name;
Console.WriteLine(hs.Name + " ist Experte für " + hs.ThemenAnzahl + " Themen!");
// mit Dr.
Console.WriteLine(hs_Klon.Name + " ist Experte für " + hs_Klon.ThemenAnzahl + " T
hemen!"); // weiterhin kein Dr.!

Console.WriteLine("Themenergänzung");
hs.Themen.Add("Cloud & Docker");
Console.WriteLine(hs.Name + " ist Experte für " + hs.ThemenAnzahl + " Themen!");
// 5 Themen!
Console.WriteLine(hs_Klon.Name + " ist Experte für " + hs_Klon.ThemenAnzahl + " T
hemen!"); // 5 Themen!
```

```
StrukturenDemo
Holger Schwichtenberg ist Experte für 4 Themen!
Holger Schwichtenberg ist Experte für 4 Themen!
Namensergänzung
Dr. Holger Schwichtenberg ist Experte für 4 Themen!
Holger Schwichtenberg ist Experte für 4 Themen!
Themenergänzung
Dr. Holger Schwichtenberg ist Experte für 5 Themen!
Holger Schwichtenberg ist Experte für 5 Themen!
```

Abbildung: Ausgabe des obigen Listings

31.4 Initialisieren einer Struktur mit default

Auch eine Struktur kann man mit der Zuweisung an `default` initialisieren.

```
Experte StandardExperte = default;
```

Während eine Variable für eine Klasse bei einer Zuweisung an `default` den Wert null erhält, entsteht bei einer Struktur eine Instanz, bei der alle Mitglieder mit Standardwerten belegt sind (siehe Abbildung). ThemenAnzahl liefert dabei einen Laufzeitfehler, denn die Liste Themen ist null und es wird versucht, auf die Anzahl zuzugreifen!

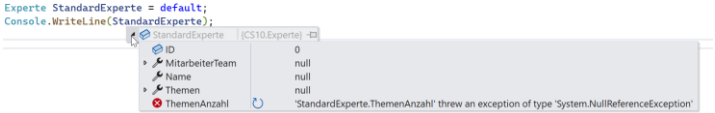


Abbildung: Initialisieren einer Struktur mit default

31.5 Strukturen mit Readonly (seit C# 7.2)

Seit C# 7.2 kann man Strukturdeklarationen mit dem Schlüsselwort readonly versehen. Damit erhält man eine unveränderliche Struktur (Immutable Struct). Damit man eine Struktur aber überhaupt mit Werten befüllen kann, gilt die Unveränderlichkeit erst nach Ende der Konstruktormethode, d.h. im Konstruktor kann man Werte setzen und ändern.

Listing: Deklaration der Readonly-Struktur

```
/// <summary>
/// Struktur, bei der alle Mitglieder Readonly sein müssen
/// </summary>
```

```
public readonly struct AppInfo
  {
   // Setzen und Ändern der Werte nur im Konstruktor erlaubt
   public AppInfo(string name, Version version, DateTime? datum)
   {
    this.Name = name;
    this.Version = version;
    this.Datum = datum;
    this.ObjektErstelltAm = DateTime.Now;
    if (this.Datum == null) this.Datum = this.ObjektErstelltAm;
   }
   // Readonly-Properties: nur Getter
   public string Name { get; }
   public Version Version { get; }
   public DateTime? Datum { get; }
   // Readonly-Fields
   private readonly DateTime ObjektErstelltAm;
   public void IncreaseVersion()
   {
    // nicht erlaubt: this.Version = new Version(this.Version.Major + 1, 0, 0,
0);
   }
  }
```

Listing: Verwendung der Readonly-Struktur

```
var appInfo = new AppInfo("MiracleList", new Version(0, 6, 3, 0), new
DateTime(2017, 11, 10));
Console.WriteLine($"Version {appInfo.Version.ToString()} vom {appInfo.Datum}");
// verboten: appInfo.Version = new Version(0, 6, 4, 0);
```

31.6 Readonly für einzelne Mitglieder einer Struktur (seit C# 8.0)

Seit C# 8.0 kann der Softwareentwickler in Strukturen das readonly-Schlüsselwort auch auf einzelne Mitglieder anwenden.

Der Zusatz readonly bedeutet:

- Für automatische Properties nur mit Getter, dass der Entwickler nur im Konstruktor der Klasse einen Wert setzen kann. Der Zusatz readonly ist nicht erlaubt, wenn es auch einen Setter gibt.
- Für explizite Properties nur mit Getter, dass der Getter den Zustand des Objekts nicht verändern kann (d.h. keine Properties oder Fields verändern)
- Für Methoden, dass die Methode den Zustand des Objekts nicht verändern kann (d.h. keine Properties oder Fields verändern)

Hinweis: Readonly ist nicht erlaubt bei dem Konstruktor!

Bisher hat man als "Readonly Property" solche Properties bezeichnet, die nur einen Getter besitzen, z.B.:

```
// Properties nur mit Getter
public string Name { get;   }
DateTime? _Datum;
public DateTime? Datum
  {
```

```
   get
   {
    if (_Datum == null) Version = new Version(1, 0, 0, 0);
    return _Datum;
   }
  }
```

Diese Sprechweise ist seit C# 8.0 nicht mehr ganz korrekt, weil Properties nun zusätzlich auch noch explizit als readonly deklariert werden können. Die Konsequenz des Zusatzes readonly ist, dass nun im Getter des "Datum"-Properties ein Schreibzugriff auf das Property "Version" nicht mehr erlaubt ist.

```
// Property mit Getter und Setter
public Version Version { get; set; }
// NEU: readonly Auto Property --> Zuweisung nur im Konstruktor!
public readonly string Name { get; }

DateTime? _Datum;
// NEU: readonly --> Darf Zustand des Objekts nicht ändern
public readonly DateTime? Datum
{
 get
 {
  // Zuweisung an Version nicht erlaubt:
  // Version = new Version(1, 0, 0, 0);
  //_Datum ??= DateTime.Now;
  return _Datum;
 }
}
```

Es folgt ein komplettes Beispiel, in dem auch eine Methode mit readonly gezeigt wird.

Listing: Struktur mit einzelnen Readonly-Mitgliedern

```
using System;

namespace CSharpSprachsyntax.CS80_Sep2019
{
 class ReadonlyStructMembersDemo
 {

  public static void Run()
  {
   var appInfo = new AppInfo("MiracleList", new Version(0, 6, 3, 0), new DateTime
(2017, 11, 10));
   Console.WriteLine($"Version {appInfo.Version.ToString()} vom {appInfo.Datum}")
;
  }
 }

 /// <summary>
 /// Struktur, mit einzelnen Readonly-Mitgliedern
 /// </summary>
 public struct AppInfo
 {

  public AppInfo(string name, Version version, DateTime? datum)
```

```
  {
   this.Name = name; // readonly --> Zuweisung nur im Konstruktor
   this.Version = version;
   this._Datum = datum;
   this.ObjektErstelltAm = DateTime.Now;
   if (this.Datum == null) this._Datum = this.ObjektErstelltAm;
  }

  // Readonly-Fields
  private readonly DateTime ObjektErstelltAm;

  // Property mit Getter und Setter
  public Version Version { get; set; }
  // NEU: readonly Auto Property --> Zuweisung nur im Konstruktor!
  public readonly string Name { get; }

  DateTime? _Datum;
  // NEU: readonly --> Darf Zustand des Objekts nicht ändern
  public readonly DateTime? Datum
  {
   get
   {
    // Zuweisung grundsätzlich nicht erlaubt, da Methode readonly
    // Version =?? new Version(0, 0, 0, 0);
    return _Datum;
   }
  }

  // NEU: readonly --> Darf Zustand des Objekts nicht ändern
  public readonly int GetMinorVersion()
  {
   // nicht erlaubt: if (Version == null) Version = new Version(0, 0, 0, 0);
   return this.Version.Minor;
  }

  public int GetMajorVersion()
  {
   // nicht erlaubt, da Name readonly
  // this.Name = "";
   if (_Datum == null) Version = new Version(0, 0, 0, 0);
   return this.Version.Major;
  }
 }
}
```

31.7 With-Ausdrücke

Mit einem With-Ausdruck erstellt man eine Wertkopie (Klon) eines Objekts und kann dabei neue Werte initialisieren. With-Ausdrücke wurden in C# 9.0 für Record-Typen (damals waren Record-Typen immer Klassen) eingeführt. Seit C# 10.0 kann man With-Ausdrücke in folgenden Fällen einsetzen:

- Record-Klassen (Typdefinition mit record oder record class)

- Record-Strukturen (Typdefinition mit record struct oder readonly record struct)
- Normale Strukturen (Typdefinition mit struct)
- Anonyme Typen (keine Typdefinition)

Achtung: With-Ausdrücke erzeugen eine flache Kopie des Objekts, d.h. es werden alle Attribute des Objekts kopiert, also auch Zeiger. Der Inhalt, auf den der Zeiger zeigt, wird aber nicht kopiert. Beispiel: Wenn ein Objekt X, das auf ein anderes Objekt Y zeigt, kopiert wird, dann zeigt die Kopie X' auf das gleiche Objekt Y. Y wird also nicht auch geklont! Wenn Y aber eine Struktur ist, dann erhält X' ein Y'.

Solche "With Expressionens" funktionieren aber nicht mit normalen Klassen, da diese eine Referenzsemantik und keine Wertesemantik haben.

Das folgende Listing zeigt ein aussagekräftiges Beispiel zu With-Ausdrücken. Mit Hilfe der .NET-Klasse System.Runtime.Serialization.ObjectIDGenerator wird eine eindeutige ID für jedes Objekt ermittelt. Daran sieht man, ob man eine Wertkopie oder eine Zeigekopie erhalten hat.

With wird mit Record-Struktur, normaler Struktur und anonymen Typ gezeigt. Die Objekt-ID ist nach dem Klonen jeweils anders, d.h. es wurde wirklich von with immer eine Wertkopie erzeugt.

Jeweils verweisen die drei Objekte auf jeweils ein Universitaet- und ein Firma-Objekt. Wie die Ausgabe der Objekt-IDs zeigt, klont With das Firma-Objekt mit, weil es ein Record-Typ ist. Alle geklonten Objekte verweisen aber auf immer das gleiche Universitaet-Objekt (ID 2).

Listing: Einsatz von With-Ausdrücken

```
namespace CS10;

internal class CS10_WithExpressions
{

 public record struct Firma(int ID, string Vorname)
 {
 }

 public class Universitaet
 {
  public string Name { get; set; }
 }

 public readonly record struct PersonR(int ID, string Vorname, string Name, Firma
 Firma, Universitaet Universitaet, string Status = "unbekannt")
 {
  public int Alter { get; init; } = 0;
 }

 public struct PersonS
 {
  public int ID = 0;
  public string Vorname = "";
  public string Name = "";
  public Firma? Firma = null;
  public Universitaet Universitaet = null;
  public string Status = "unbekannt";
  public int Alter { get; init; } = 0;
```

```
   /// <summary>
   ///  "A 'struct' with field initializers must include an explicitly
   /// declared constructor"
   /// </summary>
   public PersonS()  { }

  public override string ToString()
  {
   return $"{ID}: {Vorname} {Name} {Status}";
  }
 }

 public static void Run()
 {
  CUI.H1(nameof(CS10_WithExpressions));

  var oidg = new System.Runtime.Serialization.ObjectIDGenerator();

  void Print(Object obj)
  {

   Console.WriteLine($"Objekt #{oidg.GetId(obj, out _)}:{obj.ToString()}");
   Console.WriteLine($" - zeigt auf Universität {oidg.GetId(((dynamic)obj).Univer
sitaet, out bool _)}");
   Console.WriteLine($" - zeigt auf Firma {oidg.GetId(((dynamic)obj).Firma, out b
ool _)}");
  }

  var ITVisions = new Firma(1, "www.IT-Visions.de");
  var UniEssen = new Universitaet() { Name = "Universität Duisburg-Essen" };

  CUI.H2("With bei record struct");
  var person1 = new PersonR()
  {
   ID = 123,
   Vorname = "Holger",
   Name = "Schwichtenberg",
   Status = "hält Schulung",
   Firma = ITVisions,
   Universitaet = UniEssen
  };
  Print(person1);
  var person2 = person1 with { Vorname = "Dr. " + person1.Vorname };
  Print(person2);

  CUI.H2("With bei normaler struct");
  var person3 = new PersonS()
  {
   ID = 123,
   Vorname = "Holger",
   Name = "Schwichtenberg",
   Status = "hält Schulung",
```

```
    Firma = ITVisions,
    Universitaet = UniEssen
   };
   Print(person3);
   var person4 = person3 with { Vorname = "Dr. " + person3.Vorname };
   Print(person4);

   CUI.H2("With bei anonymen Typ");
   var person5 = new
   {
    ID = 123,
    Vorname = "Holger",
    Name = "Schwichtenberg",
    Status = "hält Schulung",
    Firma = ITVisions,
    Universitaet = UniEssen
   };
   Print(person5);
   var person6 = person5 with { Vorname = "Dr. " + person5.Vorname };
   Print(person6);
  }
}
```

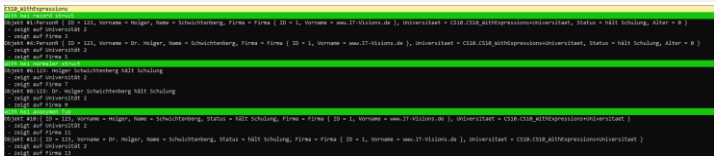


Abbildung: Ausgabe des obigen Listings

31.8 Boxing und Unboxing

Ein Wertetyp kann explizit als ein Referenztyp behandelt werden. Dazu muss der Wertetyp in ein Objekt verpackt werden. Dieser Vorgang wird als Boxing bezeichnet. Der gegensätzliche Vorgang heißt Unboxing.

In C# geschieht Boxing, wenn ein Wertetyp (wie eine Struktur) in einen Objekttyp (wie object) umgewandelt werden muss. Der Boxing-Vorgang führt dazu, dass der Wertetyp auf dem Heap gespeichert wird, anstatt auf dem Stack, um mit dem Objekttyp kompatibel zu sein.

Listing: Boxing und Unboxing

```
/// <summary>
/// Struktur für Boxing-/Unboxing-Demo
/// </summary>
struct Koordinate
{
 public int X;
 public int Y;
 public Koordinate(int x, int y)
 {
  this.X = x;
```

```
  this.Y = y;
 }
 public override string ToString()
 {
  return "x=" + X + " y=" + Y;
 }
}
…
var i = new Koordinate(42,50); // Value Typ auf Stack
   Console.WriteLine(i); // 42/50

   // Boxing --> Heap
   object oi = i;
   Console.WriteLine(oi); // 42/50

   // Unboxing --> Stack
   Koordinate i2 = (Koordinate)oi;

   Console.WriteLine(i); // 42/50
   Console.WriteLine(oi); // 42/50
   Console.WriteLine(i2); // 42/50

   // Ausgangswert verändern
   i = new Koordinate(100, 200);

   Console.WriteLine(i); // 100/200
   Console.WriteLine(oi); // 42/50, weil Zeiger auf den ursprünglichen Wert
   Console.WriteLine(i2); // 42/50, weil Zielwert von dem Zeiger beim Unboxing

   // Kopieroperationen
   Koordinate i3 = i; // Kopiert die Werte 100/200
   object oi3 = oi; // Kopiert den Zeiger auf 42/50
   Console.WriteLine(i3); // 100/200
   Console.WriteLine(oi3); // 42/50

   i.X += 1; // das verändert nur den Speicher von i, aber nicht i3
   ((dynamic)oi3).X += 1; // das verändert den Speicher, auf den oi und oi3 zeige
n!

   Console.WriteLine(i); // 100/200
   Console.WriteLine(i3); // 101/200
   Console.WriteLine(oi); // 43/50
   Console.WriteLine(oi3); // 43/50
```

31.9 Strukturen ausschließlich auf dem Stack (ref struct)

Seit C# 7.2 gibt es auch Strukturen, die immer auf dem Stack leben und niemals auf den Heap wandern können: ref struct. Dieses Verhalten macht ref struct besonders nützlich für leistungskritische Szenarien, da der Overhead der Speicherzuweisung im Heap und die Garbage Collection vermieden werden. Viele Anwendungsentwickler werden aber niemals selbst einen

Typen mit ref struct implementieren, aber von Microsoft in der Basisklassenbibliothek bereitgestellte Implementierungen nutzen.

31.9.1 Einsatz von ref struct

Wenn man einen Typen als ref struct deklariert, ist ein Boxing nicht mehr möglich. Der Einsatz von ref struct ist daher begrenzt, z.B. kann man kein Array und keine List<T> etc. daraus erzeugen.

Andere Beschränkungen von ref struct-Typen wurden in C# 13.0 aufgehoben, d.h. Microsoft den Einsatz von ref struct erweitert. Solche Typen mit können nun:

- Schnittstellen implementieren (Allerdings gilt die Einschränkung, dass die Struktur nicht in den Schnittstellentyp konvertiert werden kann)
- als Typargument genutzt werden (Allerdings muss dazu der generische Typ bzw. die generische Methode where T : allows ref struct verwenden).
- in Iteratoren verwendet werden.
- in synchronen Methoden, die Task oder Task<T> liefern, genutzt werden.

Listing: Beispiel für einen eigenen Typen mit ref struct, der eine Schnittstelle implementiert

```
internal interface IPerson
 {
  int ID { get; set; }
  int Name { get; set; }
 }

ref struct Person : IPerson // NEU seit C# 13.0: ref struct kann Schnittstelle im
plementieren
 {
  public int ID { get; set; }
  public int Name { get; set; }
  // ToString()
  public override string ToString()
  {
   return "Person #" + ID + " " + Name;
  }
 }
}

class Client
{
 public void Run()
 {
  Person p = new Person();
  p.ID = 1;
  p.Name = 2;
  Console.WriteLine(p.ID);
  Console.WriteLine(p.Name);

  // Das ist alles nicht erlaubt!
  // IPerson i = p; // Casting auf Schnittstelle
  // List<Person> PersonList = new(); // List<T>
  // PersonList[] PersonArray = new Person[10]; // Array
```

```
    }
}
```

31.9.2 Einsatz von ref struct in der .NET-Basisklassenbibliothek

Zwei wichtige generische Typen, die Microsoft in .NET Core 2.0 eingeführt hat zur Performance-Optimierung, sind als ref struct implementiert: System.Span<T> und System.ReadOnlySpan<T>. Microsoft hat seit .NET Core 2.0 die Einsatzgebiete dieser Typen kontinuierlich erweitert, z.B. in .NET 9.0:

- Microsoft hat zahlreiche Klassen aus der .NET-Klassenbibliothek, die Parameter-Arrays entgegennehmen (z.B. String.Format(), String.Join(), Console.WriteLine(), APIs im Namensraum System.Drawing), mit zusätzlichen Methodenüberladungen für ReadOnlySpan<T> ausgestattet. Dies vermeidet die bei Arrays üblichen, langsameren impliziten Heap-Allokationen, da ReadOnlySpan<T> auf dem Stack lebt.
- Die Klasse Regex bietet nun eine Methode EnumerateSplits(). Dazu gibt es ein eigenes Unterkapitel weiter oben.
- In der System.IO.File-Klasse können Entwicklerinnen und Entwickler nun direkt mit WriteAllText() Zeichenketten in Form von ReadOnlySpan<char> persistieren.
- Analog gibt es bei WriteAllBytes() eine neue Überladung für Bytefolgen, die als ReadOnlySpan<byte> vorliegen.
- ReadOnlySpan<T> bietet nun die Methoden StartsWith() und EndWith() wie die Klasse System.String.

32 Record-Typen

Record-Typen sind ein neuer Untertypus von Klassen. Record-Typen sind Referenztypen (also nicht zu verwechseln mit Strukturen), die aber eine Wertesemantik besitzen. Sie sind ein Zwischending zwischen normalen Klassen und Strukturen.

Record-Typen können auf einfache Weise als unveränderbare Instanzen (Immutable) deklariert werden. Aber nicht jeder Record-Typ ist automatisch unveränderbar.

Hinweis: Immutable Objects sind automatisch immer thread-safe, d.h. sie können beim Multi-Threading gleichzeitig in mehreren Threads verwendet werden ohne die Gefahr von Seiteneffekten (Race Conditions).

Vererbung von anderen Record-Typen ist möglich (aber nicht von normalen, mit "class" definierten Klassen). Die Vererbung kann mit sealed unterbunden werden.

Record-Typen kann man mit With-Ausdrücken klonen.

Tipp: Record-Typen kann man auch in .NET Framework und .NET Standard verwenden. Dazu ist der im Kapitel "Init Only Setters in .NET Framework und .NET Standard" beschriebene Trick bezüglich der Init Only Setter notwendig. Ohne dies meckert der Compiler bei Verwendung des Schlüsselwortes record, dass er die Klasse System.Runtime.CompilerServices.IsExternalInit nicht finden können.

32.1 Records deklarieren

Record-Typen werden mit dem Schlüsselwort "record" eingeleitet. In einem frühen Entwurf von C# 9.0 hatte Microsoft hier "data class" verwendet. Dies ist aber in der endgültigen Version von C# 9.0 nicht mehr erlaubt. Seit C# 10.0 ist alternativ und synonym zu "record" auch "record class" möglich.

Das folgende Listing zeigt einen Record-Typen "Person" und einen abgeleiteten Record-Typen "Dozent".

Praxistipp: Wie bei Klassen, kann auch bei Records mit dem Zusatz "sealed" eine Vererbung verhindert werden.

Listing: Records seit C# 9.0

```
record Person
  {
   private int ID { get; init; }
   public string Vorname { get; set; }
   public string Name { get; set; }
   public string Status = "unbekannt";
   public Person()
   {

   }
   public Person(int id, string vorname, string name)
   {
    this.ID = id;
    this.Vorname = vorname;
    this.Name = name;
   }
  }
```

```
  record Dozent : Person
  {
   public List<string> Themen { get; set; } = new();
   public Dozent(int id, string vorname, string name) : base(id, vorname, name) {
}
}
```

Das folgende Listing zeigt, was der C#-Compiler daraus erzeugt. Man sieht, dass zu beiden Record-Typen erheblicher Programmcode dazu generiert wird. Insbesondere wurde erzeugt:

- Eine C#-Klasse mit vier Init Only Properties.
- Ein Konstruktor mit vier Parametern mit Zuweisung an die Properties.
- Die Protected-Methode PrintMembers(), die den Inhalt des Objekts in einem StringBuilder liefert (ohne dabei Reflection einzusetzen!). Es wird aber nur die oberste Ebene der öffentlichen Attribute (Field und Properties) ausgegeben (keine Unterobjekte)!
- Das Überschreiben von ToString(), das den Klassennamen und den Inhalt des Objekts via Aufruf von PrintMembers() liefert.
- Die Impementierung der Operatorüberladung für Gleichheit (==) und Ungleichheit (!=) sowie der Methode Equals(). Es findet ein flacher Vergleich (nur die oberste Ebene) statt.
- Die Implementierung einer öffentlichen Methode Clone(), die eine Inhaltskopie erstellt (auch hier flache Kopie ohne Einsatz von Reflection).

Hinweis: Dieser Record-Typ ist nicht immutable. Er würde immutable, indem man Init Only-Properties verwendet (get; init; statt get; set;).

Listing: Dekompilat des Record-Typen "Person" mit ILSpy

```
// CS90.CS90_Records.Person
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;

private class Person : IEquatable<Person>
{
	protected virtual Type EqualityContract
	{
		[System.Runtime.CompilerServices.NullableContext(1)]
		[CompilerGenerated]
		get
		{
			return typeof(Person);
		}
	}

	public int ID
	{
		get;
		init;
	}

	public string Vorname
	{
```

```
        get;
        set;
    }

    public string Name
    {
        get;
        set;
    }

    public string Status
    {
        get;
        set;
    }

    public Person()
    {
        Status = "unbekannt";
        base..ctor();
    }

    public Person(int id, string vorname, string name)
    {
        Status = "unbekannt";
        base..ctor();
        ID = id;
        Vorname = vorname;
        Name = name;
    }

    public override string ToString()
    {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append("Person");
        stringBuilder.Append(" { ");
        if (PrintMembers(stringBuilder))
        {
            stringBuilder.Append(" ");
        }
        stringBuilder.Append("}");
        return stringBuilder.ToString();
    }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append("Vorname");
        builder.Append(" = ");
        builder.Append((object?)Vorname);
        builder.Append(", ");
        builder.Append("Name");
        builder.Append(" = ");
        builder.Append((object?)Name);
```

```
            builder.Append(", ");
            builder.Append("Status");
            builder.Append(" = ");
            builder.Append((object?)Status);
            return true;
        }

        [System.Runtime.CompilerServices.NullableContext(2)]
        public static bool operator !=(Person? r1, Person? r2)
        {
            return !(r1 == r2);
        }

        [System.Runtime.CompilerServices.NullableContext(2)]
        public static bool operator ==(Person? r1, Person? r2)
        {
            return (object)r1 == r2 || (r1?.Equals(r2) ?? false);
        }

        public override int GetHashCode()
        {
            return
(((EqualityComparer<Type>.Default.GetHashCode(EqualityContract) * -1521134295 +
EqualityComparer<int>.Default.GetHashCode(ID)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Vorname)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Name)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Status);
        }

        public override bool Equals(object? obj)
        {
            return Equals(obj as Person);
        }

        public virtual bool Equals(Person? other)
        {
            return (object)other != null && EqualityContract ==
other!.EqualityContract && EqualityComparer<int>.Default.Equals(ID, other!.ID) &&
EqualityComparer<string>.Default.Equals(Vorname, other!.Vorname) &&
EqualityComparer<string>.Default.Equals(Name, other!.Name) &&
EqualityComparer<string>.Default.Equals(Status, other!.Status);
        }

        public virtual Person <Clone>$()
        {
            return new Person(this);
        }

        protected Person(Person original)
        {
            ID = original.ID;
            Vorname = original.Vorname;
            Name = original.Name;
            Status = original.Status;
```

```
        }
}
```

Listing: Dekompilat des Record-Typen "Dozent" mit ILSpy

```
// CS90.CS90_Records.Dozent
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;

private class Dozent : Person, IEquatable<Dozent>
{
        protected override Type EqualityContract
        {
                [System.Runtime.CompilerServices.NullableContext(1)]
                [CompilerGenerated]
                get
                {
                        return typeof(Dozent);
                }
        }

        public List<string> Themen
        {
                get;
                set;
        }

        public Dozent(int id, string vorname, string name)
        {
                Themen = new List<string>();
                base..ctor(id, vorname, name);
        }

        public override string ToString()
        {
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.Append("Dozent");
                stringBuilder.Append(" { ");
                if (PrintMembers(stringBuilder))
                {
                        stringBuilder.Append(" ");
                }
                stringBuilder.Append("}");
                return stringBuilder.ToString();
        }

        protected override bool PrintMembers(StringBuilder builder)
        {
                if (base.PrintMembers(builder))
                {
                        builder.Append(", ");
```

```
            }
            builder.Append("Themen");
            builder.Append(" = ");
            builder.Append(Themen);
            return true;
        }

        [System.Runtime.CompilerServices.NullableContext(2)]
        public static bool operator !=(Dozent? r1, Dozent? r2)
        {
            return !(r1 == r2);
        }

        [System.Runtime.CompilerServices.NullableContext(2)]
        public static bool operator ==(Dozent? r1, Dozent? r2)
        {
            return (object)r1 == r2 || (r1?.Equals(r2) ?? false);
        }

        public override int GetHashCode()
        {
            return base.GetHashCode() * -1521134295 +
EqualityComparer<List<string>>.Default.GetHashCode(Themen);
        }

        public override bool Equals(object? obj)
        {
            return Equals(obj as Dozent);
        }

        public sealed override bool Equals(Person? other)
        {
            return Equals((object?)other);
        }

        public virtual bool Equals(Dozent? other)
        {
            return base.Equals(other) &&
EqualityComparer<List<string>>.Default.Equals(Themen, other!.Themen);
        }

        public override Person <Clone>$()
        {
            return new Dozent(this);
        }

        protected Dozent(Dozent original)
            : base(original)
        {
            Themen = original.Themen;
        }
}
```

32.2 Record-Typen mit Primärkonstruktor

Eine Besonderheit von Record-Typen ist, dass man die Deklaration radikal verkürzen kann. Anstelle des oben geschriebenen Programmcodes, kann man die beiden Record-Typen "Person" und "Dozent" auch in einer einzigen Programmcodezeile erzeugen. Syntaktisch schreibt man dabei nur einen Konstruktor mit vorangestelltem Schlüsselwort *record*. Man nennt diesen Konstruktor den Primärkonstruktor.

```
public record Person(int ID, string Vorname, string Name, string Status =
"unbekannt");

public record Dozent(int ID, string Vorname, string Name, string Status =
"unbekannt", List<string> Themen = null) : Person(ID, Vorname, Name, Status);
```

Wie man sieht, ist dabei auch Vererbung möglich! Die erbende Record-Klasse nimmt nach dem Doppelpunkt Bezug auf den Konstruktor des gewünschten Basis-Record-Typs.

Hierbei stehen in "Person" automatisch vier Properties und ein Konstruktor. In "Dozent" steht ein Property und ein Konstruktor (siehe folgendes Listing des Dekompilats).

Der Unterschied zur expliziten Langdeklaration ist aber, dass nun

- ID ein öffentliches Property ist
- Alle Properties mit Init Only Setter deklariert sind, d.h. die Werte nach der Konstruktionsphase nicht mehr änderbar (immutable) sind! Das Verändern eines Objekts beim Klonen mit with-Ausdruck ist aber weiterhin möglich, weil dies zu Konstruktionsphase des neuen Objekts zählt.
- Als Standardwert eines Konstruktorparameters kann nur ein statischer Wert verwendet werden, der zur Kompilierungszeit ausgewertet werden kann. Ein Wert, der erst zur Laufzeit entsteht (z.B. DateTime.Now) ist nicht möglich ("CS1736 Default parameter value for XY must be a compile-time constant").
- Es gibt eine Methode Deconstruct(), die den Zustand des Objekts in Einzelvariablen zerlegt. Die Einzelvariablen werden in der gleichen Reihenfolge wie im Konstruktor zurückgegeben. Das Deconstruct()-Verfahren wurde im Zusammenhang mit Tupeln in C# 7.0 eingeführt. Die Dekonstruktion verwendet man so (mit dem Unterstrich übergeht der Entwickler Werte, die ihn nicht interessieren):

```
// Nutzung von Deconstruct()
var (_, v, _, s) = hs;
Console.WriteLine("Vorname: " + v + " Status: " + s);
```

Hinweis: Record-Typen rein in der Kurzschreibweise sind automatisch immutable, da alle Properties mit Init-Only-Setter angelegt werden.

Listing: Dekompilat mit ILSpy

```
// CS90.CS90_Records.Person
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;

public class Person : IEquatable<Person>
{
    protected virtual Type EqualityContract
    {
        [System.Runtime.CompilerServices.NullableContext(1)]
        [CompilerGenerated]
        get
```

```
        {
            return typeof(Person);
        }
    }

    public int ID
    {
        get;
        init;
    }

    public string Vorname
    {
        get;
        init;
    }

    public string Name
    {
        get;
        init;
    }

    public string Status
    {
        get;
        init;
    }

    public Person(int ID, string Vorname, string Name, string Status =
"unbekannt")
    {
        this.ID = ID;
        this.Vorname = Vorname;
        this.Name = Name;
        this.Status = Status;
        base..ctor();
    }

    public override string ToString()
    {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append("Person");
        stringBuilder.Append(" { ");
        if (PrintMembers(stringBuilder))
        {
            stringBuilder.Append(" ");
        }
        stringBuilder.Append("}");
        return stringBuilder.ToString();
    }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append("ID");
        builder.Append(" = ");
        builder.Append(ID.ToString());
        builder.Append(", ");
        builder.Append("Vorname");
        builder.Append(" = ");
        builder.Append((object?)Vorname);
        builder.Append(", ");
        builder.Append("Name");
        builder.Append(" = ");
```

```
        builder.Append((object?)Name);
        builder.Append(", ");
        builder.Append("Status");
        builder.Append(" = ");
        builder.Append((object?)Status);
        return true;
    }

    [System.Runtime.CompilerServices.NullableContext(2)]
    public static bool operator !=(Person? r1, Person? r2)
    {
        return !(r1 == r2);
    }

    [System.Runtime.CompilerServices.NullableContext(2)]
    public static bool operator ==(Person? r1, Person? r2)
    {
        return (object)r1 == r2 || (r1?.Equals(r2) ?? false);
    }

    public override int GetHashCode()
    {
        return (((EqualityComparer<Type>.Default.GetHashCode(EqualityContract) *
-1521134295 + EqualityComparer<int>.Default.GetHashCode(ID)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Vorname)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Name)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Status);
    }

    public override bool Equals(object? obj)
    {
        return Equals(obj as Person);
    }

    public virtual bool Equals(Person? other)
    {
        return (object)other != null && EqualityContract ==
other!.EqualityContract && EqualityComparer<int>.Default.Equals(ID, other!.ID) &&
EqualityComparer<string>.Default.Equals(Vorname, other!.Vorname) &&
EqualityComparer<string>.Default.Equals(Name, other!.Name) &&
EqualityComparer<string>.Default.Equals(Status, other!.Status);
    }

    public virtual Person <Clone>$()
    {
        return new Person(this);
    }

    protected Person(Person original)
    {
        ID = original.ID;
        Vorname = original.Vorname;
        Name = original.Name;
        Status = original.Status;
    }

    public void Deconstruct(out int ID, out string Vorname, out string Name, out
string Status)
    {
        ID = this.ID;
        Vorname = this.Vorname;
        Name = this.Name;
        Status = this.Status;
```

```
    }
}
```

Ein Record-Typ in der Kurzschreibweise mit Primärkonstruktor darf durchaus auch noch einen normalen Klassenblock mit weiteren Properties, Fields und Methoden beinhalten. Auch ein weiterer Konstruktor ist möglich; dieser muss aber dann den automatisch generierten Konstruktor mit this(parameterliste) aufrufen. Auch die Implementierung von Schnittstellen (z.B. IDisposable) ist möglich.

Hinweis: Record-Typen in der Kurzschreibweise können die Immutability verlieren, wenn der Entwickler Properties mit normalen Settern oder beschreibbare Fields ergänzt, wie man dies im folgenden Listing sieht.

Eine weitere Einschränkung ist, dass die Dekonstruktion nur für Properties, die in Kurzschreibweise erschaffen wurden in der Reihenfolge wie im Konstruktor funktioniert.

Listing: Kurzschreibweise eines Record-Typs: Primärkonstruktor + eigene Zusätze

```
  public record Person(int ID, string Vorname, string Name, string Status =
"unbekannt") : IDisposable
  {
   public Geschlecht Geschlecht { get; set; }
   public int Alter { get; set; }
   public Firma Firma { get; set; }

   /// <summary>
   /// Eigener Konstruktur muss generierten Konstruktor mit this() aufrufen!
   /// </summary>
   /// <param name="ID"></param>
   public Person(int ID) : this(ID, "unbekannt", "unbekannt")
   {
   }

   public string GetAnrede() => Geschlecht switch
   {
    Geschlecht.f => "Sehr geehrte Frau " + Name,
    Geschlecht.m => "Sehr geehrter Herr " + Name,
    _ => "Hallo " + Name
   };

   public void Dispose()
   {
    Console.WriteLine("Dispose!");
   }
  }
```

In diesem Fall kann man die Zusatzproperties Geschlecht und Alter nicht im Konstruktor, aber via Objektinitialisierung in geschweiften Klammern befüllen:

```
Person hs1d = new Person(123, "Holger", "Schwichtenberg", "verheiratet") { Alter
= 48, Geschlecht = Geschlecht.m };
```

Diese Properties Alter und Geschlecht sind auch später noch änderbar, weil sie als normale Properties mit get; set; deklariert sind.

32.3 Records verwenden

Das folgende Listing zeigt einen Nutzer der beiden Record-Typen:

- Es wird eine Instanz des Record-Typen "Person" erstellt.
- Die Instanz wird mit Hilfe von ToString() und Console.WriteLine() ausgeben.
- Es wird eine Instanz des Record-Typen "Dozent" erstellt.
- Der Objektverweis wird kopiert durch die Zuweisung dozent = hs. Dies ist eine Referenzkopie wie bei Instanzen von Klassen üblich.
- Einer der Objektverweise wird verändert (das ist nur möglich, wenn der Record in der Langschreibweise und nicht mit Init Only Setter geschrieben wurde)
- Die Ausgaben für beide Objektverweise sind gleich. Dies belegt, dass dozent und hs auf die gleiche Speicherstelle verweisen.
- Nun wird der Dozent-Record geklont mit *with* ohne Veränderung (via sogenanntem "With-Ausdruck"). Dies ist eine Wertkopie.
- Die Ausgaben sind gleich.
- Nun wird der Dozent-Record geklont mit *with* mit Veränderung von Status und Themen (wieder eine Wertkopie).
- Die Ausgaben sind nicht mehr gleich.

Listing: CS90_Records.cs

```
public static void CS90Records_Client()
  {
   CUI.MainHeadline(nameof(CS90Records_Client));

   CUI.Headline("Record-Instanz von 'Person' erstellen");
   Person mm = new Person(123, "Max", "Müller");
   if (mm != null) CUI.PrintSuccess("OK!");
   CUI.Headline("ToString()");
   var ausgabe = mm.ToString();
   Console.WriteLine(ausgabe);
   // oder direkt:
   Console.WriteLine(mm);

   CUI.Headline("Record-Instanz von 'Dozent' erstellen");
   Dozent hs = new Dozent(123, "Holger", "Schwichtenberg");
   hs.Themen.Add(".NET");
   hs.Themen.Add("C#");
   hs.Themen.Add("JavaScript/TypeScript");
   hs.Themen.Add("DevOps");
   if (hs != null) CUI.PrintSuccess("OK!");

   CUI.Headline("Kopie des Objektverweises erstellen");
   var dozent = hs;
   hs.Status = "Original";
   hs.Themen.Add("PowerShell");
   Console.WriteLine(hs);
   Console.WriteLine(dozent);
   if (dozent == hs) CUI.PrintSuccess("Dozent und hs haben gleiche Inhalte!");
   else CUI.PrintWarning("Dozent und hs haben NICHT gleiche Inhalte!");

   CUI.Headline("Kopie der Instanz erstellen mit with");
   Dozent hsKlon1 = hs with { };
```

```
   // Person hsKlon = hs.Clone(); // geht nicht, Clone() wird erst durch Compiler
erzeugt!!!
   Console.WriteLine(hs);
   Console.WriteLine(hsKlon1);
   if (hsKlon1 == hs) CUI.PrintSuccess("Klon1 ist exakt gleich");
   else CUI.PrintWarning("Klon2 ist verändert!");

   CUI.Headline("Kopie der Instanz erstellen mit with und Veränderung");
   Dozent hsKlon2 = hs with { Status = "geklont" };
   hsKlon2.Themen.Add("Java");
   Console.WriteLine(hs);
   Console.WriteLine(hsKlon2);
   if (hsKlon2 == hs) CUI.PrintSuccess("Klon2 ist exakt gleich");
   else CUI.PrintWarning("Klon2 ist verändert!");
  }
```

```
CS90Records Client
Record-Instanz von 'Person' erstellen
ToString()
Person { Vorname = Max, Name = Müller, Status = unbekannt }
Person { Vorname = Max, Name = Müller, Status = unbekannt }
Record-Instanz von 'Dozent' erstellen
Kopie des Objektverweises erstellen
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = Original, Themen = System.Collections.Generic.List`1[System.String] }
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = Original, Themen = System.Collections.Generic.List`1[System.String] }
Dozent und hs haben gleiche Inhalte!
Kopie der Instanz erstellen mit with
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = Original, Themen = System.Collections.Generic.List`1[System.String] }
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = Original, Themen = System.Collections.Generic.List`1[System.String] }
Klon1 ist exakt gleich
Kopie der Instanz erstellen mit with und Veränderung
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = Original, Themen = System.Collections.Generic.List`1[System.String] }
Dozent { Vorname = Holger, Name = Schwichtenberg, Status = geklont, Themen = System.Collections.Generic.List`1[System.String] }
Klon2 ist verändert!
```

Abbildung: Ausgabe des obigen Listings

32.4 Überschreiben von ToString()

Bereits in C# 9.0 war es möglich, auch in einem Record-Typen Methoden zu überschreiben, auch wenn diese Methoden Teil der automatischen Codegenerierung für den Record waren, z.B. ToString(). Damit wurde die automatische Implementierung außer Kraft gesetzt.

Seit C# 10.0 ist es nun, aber ausschließlich bei ToString(), erlaubt, dass dabei das Schlüsselwort "sealed" eingesetzt wird. Das bedeutet, dass ein Record-Typ verhindern kann, dass davon erbende Record-Typen ToString() wieder überschreiben mit der automatischen Implementierung. Folglich gilt eine sealed ToString()-Implementierung auch für alle abgeleiteten Record-Typen.

> **Hinweis:** Dieses Sprachfeature funktioniert nur bei Record-Klassen, nicht bei Record-Strukturen, da Record-Strukturen nicht erben können!

Listing: Record-Typ mit überschriebenen ToString() mit Zusatz "sealed"

```
public record class Person(int ID, string Vorname, string Name, string Status =
"unbekannt") : IDisposable
  {
   public Geschlecht Geschlecht { get; set; }
   public int Alter { get; set; }
   public Firma Firma { get; set; }

   // Eigene ToString()-Implementierung möglich
   // erst seit C# 10.0 kann die auch sealed sein und gilt dann auch für
abgeleitete Klasse "Dozent"
   public sealed override string ToString()
```

```
    {
      return $"Person #{ID}: {Vorname} {Name}";
    }
}
```

Wenn nun Dozent von Person erbt

```
public record Dozent(int ID, string Vorname, string Name, string Status =
"unbekannt", List<string> Themen = null) : Person(ID, Vorname, Name, Status);
```

Dann wird auch eine Instanz von Dozent immer ToString() in Person aufrufen.

```
Dozent hs = new Dozent(123, "Holger", "Schwichtenberg") { Themen = new
List<string>() };
Console.WriteLine(hs);
```

Die letzte Zeile gibt also aus:

Person #123: Holger Schwichtenberg

32.5 Record Structs

Ein Record in C# 9 ist immer eine Klasse. Seit C# 10.0 drei Arten von Record-Typen:

- *record class*: Dies ist gleichbedeutend mit der Verwendung von record ohne Zusatz. Es entsteht wie bisher eine Klasse, also ein Referenztyp auf dem Heap. Alle per Primärkonstruktor erzeugten Properties haben einen Init Only Setter, d.h. das entstehende Objekt ist immutable (sofern nicht explizit Properties mit Setter hinzugefügt wurden).
- *record struct*: Hier entsteht eine Struktur, also ein Wertetyp auf dem Stack (implizit erbend von System.ValueType). Anders bei einer record class haben alle per Primärkonstruktor erzeugten Properties einen normalen Setter, d.h. das Objekt ist mutable.
- *readonly record struct*: Auch hier entsteht eine Struktur, also ein Wertetyp auf dem Stack (implizit erbend von System.ValueType). Alle per Primärkonstruktor erzeugten Properties haben einen Init Only Setter, d.h. das Objekt ist immutable.

Hinweis: Im Gegensatz zu einer Record-Klasse kann eine Record-Struktur nicht erben! Auch gibt es keinen EqualityContract und keine Null-Prüfungen im generierten Code einer Record-Struktur.

Name	Struktur	Record-Struktur	Record-Klasse	Klasse
Seit C#-Version (Jahr)	1.0 (2001)	10.0 (2021)	9.0 (2020)	1.0 (2001)
Typart	Wertetyp	Wertetyp	Referenztyp	Referenztyp
Zuweisungssemantik	Wert	Wert	Wert	Referenz
Speicherort	Stack	Stack	Heap	Heap
Primärkonstruktor möglich	Nein	Ja	Ja	Nein
Codegenerierung für ToString(), Dekonstruktio	Nein	Ja	Ja	Nein

Name	Struktur	Record-Struktur	Record-Klasse	Klasse
n und Vergleich				
Vererbung	Nicht möglich	Nicht möglich	Möglich	Möglich
Veränderbar/ Mutable	struct xy	record struct xy	record class xy oder record xy (sofern kein Primärstruktur verwendet wird und keine Init-Only-Setter)	class xy
Unveränderbar /Immutable	readonly struct xy (geht auch ohne "readonly, wenn alle Properties mit Init-Only-Setter deklariert werden)	readonly record struct xy	record class xy oder record xy (sofern Primärstruktur verwendet wird oder alle Properties mit Init-Only-Setter deklariert werden)	class xy (sofern alle Properties mit Init-Only-Setter deklariert werden)

Tabelle 1: Klassen und Strukturen: Übersicht über die verschiedenen Typ-Arten in C#

Aus dieser Deklaration einer record struct

```
public record struct Person(int ID, string Vorname, string Name, string Status =
"unbekannt")
  {
   public int Alter { get; set; } = 0;
  }
```

wird der nachstehend abgedruckte Programmcode mit Properties mit Getter und Init-Only-Setter inklusive Equals()-Implementierung, Operator-Überladung für == und !=, Ausgabe aller Datenmitglieder bei ToString() sowie Deconstruct()-Implementierung.

Hinweis: Zu beachten ist, dass grundsätzlich in einer Record-Struktur die zusätzliche Fields und Properties mit primitiven Typen, die nicht Teil des Primärkonstruktors sind, explizit initialisiert werden müssen, vgl. C#-Regeln CS0171 (für Fields) bzw. CS0843 (für Properties): "must be fully assigned before control is returned to the caller".

Bei einer readonly Record-Struktur müssen alle zusätzlichen Properties Init Only Properties (get; init;) sein. Die Regel CS8341 ("Auto-implemented instance properties in readonly structs must be readonly.") ist etwas fehlleitend, den der Zusatz "readonly" ist zwar möglich, hilft alleine aber nicht. Dies sieht man bei den Properties Alter und Wohnort. "Readondly" müsste in diesem Fall zwingend bei Fields deklariert werden. Weiterhin gilt jedoch die

Initialisierungspflicht. Da für Fields gleichzeitig die Regel CS0191 "A readonly field cannot be assigned to" gilt, kriegt man den Einsatz von Fields und Primärkonstruktor nicht in Einklang.

Listing: Generierter Programmcode aus einer record struct

```
// CS10.CS10_RecordTypen.Person
using System;
using System.Collections.Generic;
using System.Text;

public struct Person : IEquatable<Person>
{
    public int ID { get; set; }

    public string Vorname { get; set; }

    public string Name { get; set; }

    public string Status { get; set; }

    public int Alter { get; set; }

    public Person(int ID, string Vorname, string Name, string Status =
"unbekannt")
    {
        this.ID = ID;
        this.Vorname = Vorname;
        this.Name = Name;
        this.Status = Status;
        Alter = 0;
    }

    public override string ToString()
    {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append("Person");
        stringBuilder.Append(" { ");
        if (PrintMembers(stringBuilder))
        {
            stringBuilder.Append(' ');
        }
        stringBuilder.Append('}');
        return stringBuilder.ToString();
    }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append("ID = ");
        builder.Append(ID.ToString());
        builder.Append(", Vorname = ");
        builder.Append((object?)Vorname);
        builder.Append(", Name = ");
        builder.Append((object?)Name);
        builder.Append(", Status = ");
        builder.Append((object?)Status);
        builder.Append(", Alter = ");
        builder.Append(Alter.ToString());
        return true;
    }

    public static bool operator !=(Person left, Person right)
    {
        return !(left == right);
    }
```

```
    public static bool operator ==(Person left, Person right)
    {
        return left.Equals(right);
    }

    public override int GetHashCode()
    {
        return (((EqualityComparer<int>.Default.GetHashCode(ID) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Vorname)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Name)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Status)) * -1521134295 +
EqualityComparer<int>.Default.GetHashCode(Alter);
    }

    public override bool Equals(object obj)
    {
        return obj is Person && Equals((Person)obj);
    }

    public bool Equals(Person other)
    {
        return EqualityComparer<int>.Default.Equals(ID, other.ID) &&
EqualityComparer<string>.Default.Equals(Vorname, other.Vorname) &&
EqualityComparer<string>.Default.Equals(Name, other.Name) &&
EqualityComparer<string>.Default.Equals(Status, other.Status) &&
EqualityComparer<int>.Default.Equals(Alter, other.Alter);
    }

    public void Deconstruct(out int ID, out string Vorname, out string Name, out
string Status)
    {
        ID = this.ID;
        Vorname = this.Vorname;
        Name = this.Name;
        Status = this.Status;
    }
}
```

Hingegen entsteht aus

```
public readonly record struct Person2(int ID, string Vorname, string Name, string
 Status = "unbekannt")
{
 //Regel CS0843: Auto-
implemented property  must be fully assigned before control is returned to the ca
ller
 public int Alter { get; init; } = 0;
}
```

dann der nachstehende Code mit Init Only Setter-basierten Properties (auch mit Equals()-Implementierung sowie Operator-Überladung für == und !=).

Listing: Generierter Programmcode aus einer readonly record struct

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

public readonly struct Person2: IEquatable<Person_ImmutableRecordStructs>
{
    public int ID { get; init; }

    public string Vorname { get; init; }

    public string Name { get; init; }
```

```
    public string Status { get; init; }

    public int Alter { get; init; }

    public Person_ImmutableRecordStructs(int ID, string Vorname, string Name,
string Status = "unbekannt")
    {
        this.ID = ID;
        this.Vorname = Vorname;
        this.Name = Name;
        this.Status = Status;
        Alter = 0;
    }

    public override string ToString()
    {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append("Person_ImmutableRecordStructs");
        stringBuilder.Append(" { ");
        if (PrintMembers(stringBuilder))
        {
            stringBuilder.Append(' ');
        }
        stringBuilder.Append('}');
        return stringBuilder.ToString();
    }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append("ID = ");
        builder.Append(ID.ToString());
        builder.Append(", Vorname = ");
        builder.Append((object?)Vorname);
        builder.Append(", Name = ");
        builder.Append((object?)Name);
        builder.Append(", Status = ");
        builder.Append((object?)Status);
        builder.Append(", Alter = ");
        builder.Append(Alter.ToString());
        return true;
    }

    public static bool operator !=(Person2left, Person2right)
    {
        return !(left == right);
    }

    public static bool operator ==(Person2left, Person2right)
    {
        return left.Equals(right);
    }

    public override int GetHashCode()
    {
        return (((EqualityComparer<int>.Default.GetHashCode(ID) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Vorname)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Name)) * -1521134295 +
EqualityComparer<string>.Default.GetHashCode(Status)) * -1521134295 +
EqualityComparer<int>.Default.GetHashCode(Alter);
    }

    public override bool Equals(object obj)
    {
```

```
        return obj is Person2&& Equals((Person_ImmutableRecordStructs)obj);
    }

    public bool Equals(Person2other)
    {
        return EqualityComparer<int>.Default.Equals(ID, other.ID) &&
EqualityComparer<string>.Default.Equals(Vorname, other.Vorname) &&
EqualityComparer<string>.Default.Equals(Name, other.Name) &&
EqualityComparer<string>.Default.Equals(Status, other.Status) &&
EqualityComparer<int>.Default.Equals(Alter, other.Alter);
    }

    public void Deconstruct(out int ID, out string Vorname, out string Name, out
string Status)
    {
        ID = this.ID;
        Vorname = this.Vorname;
        Name = this.Name;
        Status = this.Status;
    }
}
```

33 Immutable Objects

Als Immutable Object wird in der objektorientierten Lehre ein Objekt bezeichnet, dessen Zustand nach der Erzeugung nicht mehr verändert werden kann. Normalweise sind alle Objekte in C# veränderbar (mutable).

Hinweis: Immutable Objects sind automatisch immer thread-safe, d.h. sie können beim Multi-Threading in mehreren Threads verwendet werden ohne die Gefahr von Seiteneffekten (Race Conditions).

In C# kann man Immutable Objects auf folgende Weisen erstellen:

- Klassen mit Readonly Fields
- Klassen mit Properties mit Init Only Setter

33.1 Immutable Objects auf Basis von Readonly Fields

Das Listing zeigt ein Immutable Object "ImmutablePerson" auf Basis von Fields mit Zusatz "readonly". Dies ist möglich seit C# 1.0.

Die Werte des Objekts können bei der Field-Deklaration und im Konstruktor gesetzt werden.

Listing: ImmutableObjects_Fields.cs

```
using ITVisions;
using System;

namespace Immutable_Fields
{
 class ImmutablePerson
 {
  private readonly int id;
  private readonly string name;
  public readonly DateTime AngelegtAm = DateTime.Now;

  public ImmutablePerson(int id, string name)
  {
   this.id = id;
   this.name = name;
  }

  public int ID
  {
   get { return this.id; }
  }

  public string Name
  {
   get { return this.Name; }
  }

  public override string ToString()
  {
   return "Person " + this.id + ": " + this.Name;
  }
```

```
 }

 class ImmutablePersonClient
 {
  public static void Run()
  {
   CUI.Headline(nameof(ImmutablePersonClient));
   var hs = new ImmutablePerson(123, "Dr. Holger Schwichtenberg");
   Console.WriteLine(hs);
   // nicht möglich: hs.Name = "xy";
  }
 }
}
```

33.2 Immutable Objects auf Basis von Properties mit Init Only Setter

Das Listing zeigt ein Immutable Object "ImmutablePerson" auf Properties mit Init Only Setter. Dies ist möglich seit C# 9.0.

Die Werte des Objekts können nur bei der Property-Deklaration, im Konstruktor und der Objekt-Initialisierung gesetzt werden.

Listing: ImmutableObjects_Properties.cs

```
using ITVisions;
using System;

namespace Immutable_Properties
{
 class ImmutablePerson
 {
  private int id { get; init; }
  private string name { get; init; }
  public DateTime AngelegtAm { get; init; } = DateTime.Now;

  public ImmutablePerson(int id, string name)
  {
   this.id = id;
   this.name = name;
  }

  public int ID
  {
   get { return this.id; }
  }

  public string Name
  {
   get { return this.Name; }
  }

  public override string ToString()
  {
```

```
   return "Person " + this.id + ": " + this.Name;
  }
 }

 class ImmutablePersonClient
 {
  public static void Run()
  {
   CUI.Headline(nameof(ImmutablePersonClient));
   var hs = new ImmutablePerson(123, "Dr. Holger Schwichtenberg") { AngelegtAm =
DateTime.Now };
   Console.WriteLine(hs);
   // nicht möglich: hs.Name = "xy";
  }
 }
}
```

33.3 Immutable Objects auf Basis von Records

Die kürzeste Variante zur Deklaration eines Immutable Objects ist seit C# 9.0 die Deklaration eines Record-Typen in Kurzschreibweise, dann erstellt der Compiler automatisch Properties mit Init Only Setter.

Dabei ist es allerdings nicht möglich, die Eigenschaft *AngelegtAm* innerhalb des Record-Typen im Standard mit DateTime.Now zu belegen, da nur statische Werte als Standardwert in einem Konstruktor erlaubt sind.

Listing: ImmutableObjects_Records.cs

```
using ITVisions;
using System;

namespace Immutable_Records
{
 record ImmutablePerson(int id, string name, DateTime AngelegtAm );

 class ImmutablePersonClient
 {
  public static void Run()
  {
   CUI.Headline(nameof(ImmutablePersonClient));
   var hs = new ImmutablePerson(123, "Dr. Holger Schwichtenberg",DateTime.Now);
   Console.WriteLine(hs);
   // nicht möglich: hs.Name = "xy";
  }
 }
}
```

33.4 Praxisbeispiel: Immutable Objects mit Record-Typen beim Flux-/Redux-Pattern

Das Flux-Pattern ist eine Variante des Observer-Pattern, die in von der Firma Facebook im Jahr 2014 veröffentlicht wurde. Redux ist eine modifizierte Implementierung von Flux, die 2015 erschienen ist (vgl. [*redux.js.org/understanding/history-and-design/prior-art*]).

Redux verwendet "Pure Funktionen" (Pure Functions) im sogenannten "Reducer", die einen Zustand nicht modifizieren, sondern einen neuen Zustand erzeugen (Immutable Objects).

Ohne Record-Typen sieht die Implementierung eines Zustands für einen einfachen Zähler und eines Reducers zum Ändern des Zählers in C# so aus:

```
public class CounterState
{
  public int ClickCount { get; }

  public CounterState(int clickCount)
  {
   ClickCount = clickCount;
  }
 }

public static class Reducers
 {
  [ReducerMethod]
  public static CounterState ReduceIncrementCounterAction(CounterState state,
IncrementCounterAction action) =>
   new CounterState(state.ClickCount + 1);
  [ReducerMethod]
  public static CounterState ReduceDecrementCounterAction(CounterState state,
IncrementCounterAction action) =>
 new CounterState(state.ClickCount - 1);
 }
```

Mit Record-Typen ist dies wesentlich prägnanter implementierbar:

```
public record CounterState(int ClickCount);

public static class Reducers1
 {
  [ReducerMethod]
  public static CounterState ReduceIncrementCounterAction(CounterState state,
IncrementCounterAction action) =>
      state with { ClickCount = state.ClickCount + 1 };

  [ReducerMethod]
  public static CounterState ReduceDecrementCounterAction(CounterState state,
DecrementCounterAction action) =>
      state with { ClickCount = state.ClickCount - 1 };
 }
```

Weitere Teile des Redux-Patterns (Feature, Actions) sind hier nicht wiedergegeben, da sie sich durch den Einsatz von Record-Typen nicht ändern. Siehe dazu die Bibliothek Fluxor, die Redux für .NET realisiert: *github.com/mrpmorris/Fluxor*

34 Tupel

Die größte syntaktische Erweiterung in C# 7.0 betrifft Tupel. Tupel, also "Listen endlich vieler, nicht notwendigerweise voneinander verschiedener Objekte" [*de.wikipedia.org/wiki/Tupel*]. Tupel haben den Vorteil, dass man eine Datenstruktur definieren und mit Werten befüllen kann, ohne dafür extra eine Klasse oder eine Struktur zu deklarieren. So kann zum Beispiel eine Methode mit einem Tupel mehrere Werte zurückliefern, ohne ref oder out in der Parameterliste zu verwenden und ohne extra eine Klasse oder Struktur für den Rückgabetyp zu schreiben.

Hinweis: Tupel sind Werttypen (wie Strukturen).

34.1 Alte Tupelimplementierung mit System.Collections.Tupel

Tupel können C#-Entwickler seit .NET Framework 4.0 durch die generische .NET-Klasse System.Collections.Tupel verwenden. Diese Klasse unterstützt Tupel mit bis zu acht Elementen (also Oktupel), die über die Field-Attribute Item1, Item2 bis Item8 abgerufen werden können.

```
Tuple<int, string, bool> dozent = new Tuple<int, string, bool>(1, "Holger
Schwichtenberg", true);
Console.WriteLine($"Dozent mit der ID{dozent.Item1}: {dozent.Item2} {(
dozent.Item3 ? "ist ein .NET-Experte!": "")}");
```

34.2 Neue Tupelimplementierung in der Sprachsyntax

In C# 7.0 hat sich Microsoft entschlossen, die Tupel-Unterstützung direkt in der Sprachsyntax zu verankern. Ein Tupel deklariert der Entwickler mit runden Klammern bei der Zuweisung zu einer Variablen:

```
var dozent2 = (1, "Holger Schwichtenberg", true);
```

Die Datentypen der Elemente ergeben sich hier aus den zugewiesenen Werten. In diesem Fall sind die Elemente weiterhin Item1, Item2 bis Item8 benannt. Der Entwickler kann aber in der Deklaration auch sprechende Namen angeben und diese dann verwenden:

```
var dozent3 = (ID: 1, Name: "Holger Schwichtenberg", DOTNETExperte: true);
Console.WriteLine($"Dozent mit der ID{dozent3.ID}: {dozent3.Name}
{(dozent3.DOTNETExperte ? "ist ein .NET-Experte!" : "")}");
```

Auch hier erfolgt die Typvergabe durch Typableitung aus den angegebenen Werten. Wer explizit Kontrolle über die Typen der Tupelelemente möchte, kann die folgende Syntax nutzen:

```
 (int ID, string Name, bool DOTNETExperte) dozent4 = (ID:1, Name:"Holger
Schwichtenberg", DOTNETExperte:true);
```

Im Zuweisungsteil (rechts des Gleichheitszeichens) ist die Wiederholung der Namen optional. Ein Tupel kann an allen Stellen zum Einsatz kommen, wo Typnamen erlaubt sind, also auch bei Attributen einer Klasse und Rückgabewerten einer Methode. Die maximale Anzahl der Elemente pro Tupel ist nicht dokumentiert [*learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7#tuples*]. Im Test funktionierte ein Tupel mit 50 Elementen, was hinsichtlich der Übersichtlichkeit schon grenzwertig ist.

Hinweis: Für die Realisierung der Tupel benötigt der C#-Compiler eine .NET-Klasse mit Namen System.ValueTuple. Diese ist in .NET Framework seit Version 4.7 bzw. .NET Core seit Version 2.0 enthalten. Ältere .NET-Versionen müssen ein NuGet-Paket [*packages.nuget.org/packages/System.ValueTuple*] installieren. Ohne dies kommt es zum Kompilerfehler "Predefined type 'System.ValueTuple is not defined or imported". Der Name

"ValueTuple" weist darauf hin, dass die neuen Tupel als Value Types auf dem Stack gespeichert werden, während die alten Tupel (System.Collections.Tupel) als Reference Types im Heap residieren.

34.3 Tupel-Dekonstruktion

Tupel lassen sich in ihre Einzelelemente via Dekonstruktion zerlegen. Das nächste Listing zeigt vier Varianten der Dekonstruktion des Rückgabewertes der Methode GetDozent(), die ein dreiteiliges Tupel liefert. In den ersten drei Fällen entstehen jeweils drei einzelne Variablen. Im vierten Fall kommt die Discard-Variable, deren Namen nur aus dem Unterstrich _ besteht, zum Einsatz. Sie zeigt an, dass ein Element verworfen werden soll, d.h. zur Weiterverarbeitung nicht bereitsteht.

Listing: Vier Varianten der Dekonstruktion eines Tupels

```
// Dekonstruktion eines Tupel
(int ID1, string Name1, bool DOTNETExperte1) = GetDozent();
Console.WriteLine(ID1);
Console.WriteLine(Name1);
Console.WriteLine(DOTNETExperte1);

// Dekonstruktion eines Tupel: var möglich
(var ID2, var Name2, var DOTNETExperte2) = GetDozent();
Console.WriteLine(ID2);
Console.WriteLine(Name2);
Console.WriteLine(DOTNETExperte2);

// Dekonstruktion eines Tupel: verkürzte Form des Einsatzes von var
var (ID3, Name3, DOTNETExperte3) = GetDozent();
Console.WriteLine(ID3);
Console.WriteLine(Name3);
Console.WriteLine(DOTNETExperte3);

// Dekonstruktion eines Tupel: Werte ignorieren mit _
var (_, Name4, DOTNETExperte4) = GetDozent();
Console.WriteLine(Name4);
Console.WriteLine(DOTNETExperte4);

...

static (int ID, string Name, bool DOTNETExperte) GetDozent()
{
 return (ID: 1, Name: "Holger Schwichtenberg", DOTNETExperte: true);
}
```

Die Dekonstruktion ist auch auf Instanzen von Klassen anwendbar, wenn diese eine Methode Deconstruct() anbieten, siehe Listing.

Listing: Klassendekonstruktion

```
class Dozent
{
 public int ID { get; set; }
 public string Name { get; set; }
 public bool DOTNETExperte { get; set; }
```

```
  public void Deconstruct(out int ID, out string Name, out bool DOTNETExperte)
  {
   ID = this.ID;
   Name = this.Name;
   DOTNETExperte = this.DOTNETExperte;
  }

  public Dozent()  {   }

  // Expression-bodied Constructor
  public Dozent(int ID) => this.ID = ID;

  // Expression-bodied Finalizer
  ~Dozent() => Console.Error.WriteLine("Finalized!");

  // Expression-bodied Getter und Setter
  private Decimal? honorar2;
  public Decimal? Honorar2
  {
   get => this.honorar;
   set => this.honorar = value ?? 1000.00m;
  }

  private Decimal? honorar;
  public Decimal? Honorar
  {
   get => this.honorar;

   // throw ist nun an Stellen erlaubt, wo Ausdrücke erwartet werden, z.B. ??
und Expression Lambdas
   set => this.honorar = value ??
       throw new ArgumentNullException(nameof(value), "Kein Honorar nicht
erlaubt!");
  }

 public static void ClassDeconstruction()
 {
  CUI.Headline(nameof(ClassDeconstruction));
  // Dozent ist dekonstruierbare Klasse mit Deconstruct()
  var d = new Dozent() { ID = 1, Name = "Holger Schwichtenberg", DOTNETExperte =
true };
  (var ID, var Name, var DOTNETExperte) = d;
  Console.WriteLine(ID);
  Console.WriteLine(Name);
  Console.WriteLine(DOTNETExperte);
 }
```

Seit C# 10.0 gibt es als neues Feature "Mixed Deconstruction". Dies bedeutet, dass man reine Zuweisungen an bestehende Variablen und neue Variablendeklarationen mit Initialisierung in einer Zeile gemischt kann.

```
 // Tupel deklarieren
```

```
  var point = (x: 100, y: 200);

  // schon vor C# 10.0 möglich: Dekonstruktion mit zwei
Deklarationen+Initialisierung für neue Variablen
  (int x, int y) = point;

  // schon vor C# 10.0 möglich: Dekonstruktion mit Zuweisung zu zwei bestehenden
Variablen
  int x1 = 0;
  int y1 = 0;
  (x1, y1) = point;

  // seit C# 10: Dekonstruktion mit Zuweisung und Initialisierung gemischt
möglich
  int x2 = 0;
  (x2, int y2) = point;
```

34.4 Serialisierung von Tupeln

Bei der Serialisierung von Tupeln (siehe folgendes Listing) wird man feststellen, dass die im Programmcode vergebenenen Elementnamen nicht serialisiert werden, sondern nur als "Item1", "Item2", "Item3" usw. dort erscheinen.

Listing: JSON-Serialisierung eines Tupel

```
var dozent7 = (ID: 1, Name: "Holger Schwichtenberg", DOTNETExperte: true);
var json = JsonConvert.SerializeObject(dozent7);
Console.WriteLine(dozent7.ID);
Console.WriteLine(dozent7.Name);
Console.WriteLine(dozent7.DOTNETExperte);
Console.WriteLine("JSON:" + json);
#region Ergebnis
//JSON: { "Item1":1,"Item2":"Holger Schwichtenberg","Item3":true}
#endregion
```

Grund dafür ist, dass die Elementnamen nur "syntaktischer Zucker" des C#-Compilers sind. In Wirklichkeit besitzt die Klasse ValueTupel nur die Elementnamen mit "ItemX". Dies sieht man auch, wenn man den ILSpy [*github.com/icsharpcode/ILSpy*] zum Dekompilieren einsetzt.

```
ValueTuple<int, string, bool> dozent7 = new ValueTuple<int, string, bool>(1, "Holger Schwichtenberg", true);
string json = JsonConvert.SerializeObject(dozent7);
Console.WriteLine(dozent7.Item1);
Console.WriteLine(dozent7.Item2);
Console.WriteLine(dozent7.Item3);
Console.WriteLine("JSON:" + json);
```

Abbildung: Obiger Programmcode mit dem ILSpy dekompiliert

34.5 Vergleich von Tupeln (C# 7.3)

In C# 7.0 hatte Microsoft ValueTupel als leichtgewichtige, unbenannte Datenstruktur eingeführt, die sich auf dem Stack speichert - im Gegensatz zu dem in .NET Framework 4.0 eingeführten Referenztyp System.Collections.Tupel. Nun erst, in C# 7.3, erlaubt die Programmiersprache auch den direkten Vergleich zweier Tupel mit den Vergleichsoperatoren == und !=.

```
var p = (ID: 1, Name: "H. Schwichtenberg", DOTNETExperte: true);
// ...
```

```
if (p == (1, "H. Schwichtenberg", true)){ Console.WriteLine("Er ist es :-)"); }
if (p != (1, "H. Schwichtenberg", true)){ Console.WriteLine("Er ist nicht :-(");}
```

Nicht lediglich eine Variable und ein Tupelausdruck, sondern auch zwei Tupelausdrücke sind jetzt direkt vergleichbar. So kann man nun anstelle von

```
if (x == 1 && y == 2) { Console.WriteLine("x ist 1 und y ist 2!"); }
```

auch formulieren:

```
if ((x, y) == (1, 2)) { Console.WriteLine("x ist 1 und y ist 2!"); }
```

35 Typaliase (seit C# 12.0)

Seit C# 12.0 gibt es mit Typaliasen die Möglichkeit, für einen Typen einen alternativen Namen zu definieren. Ein Alias ist möglich für C#-Typen (z.B. Arrays und Tupel), .NET-Basisklassen/-Strukturen oder eigene Klassen/Strukturen.

Einmal mehr kommt dabei das Schlüsselwort using zum Einsatz.

Wenn Sie schreiben

```
using Numbers = int[];
```

können Sie fortan Numbers anstelle von `int[]` bei Typdeklarationen verwenden:

```
Numbers numbers = new int[10];
```

Allerdings darf man den Alias NICHT bei der Instanziierung verwenden:

```
Numbers numbers = new Numbers;
```

Auch kann man leider keinen Alias definieren mit Hilfe eines Aliases. Das geht also auch nicht:

```
using DbIntList = List<DbInt>;
```

Zweites Beispiel: DbInt als Alias für ein int? bzw. Nullable<int>:

```
using DbInt = int?;
```

Danach ist möglich:

Listing: Verwendung des Alias DbInt

```
public DbInt LoadNumberFromDatabase()
{
 try
 {
  …
 }
 catch (Exception)
 {
  return null;
 }
}

DbInt n;
n = LoadNumberFromDatabase();
Console.WriteLine(n == null ? "null" : n);
```

Drittes Beispiel: Typalias für ein Tupel

```
using Measurement = (string Units, int Distance);
```

Danach ist möglich:

Listing: Verwendung des Alias Measurement

```
public Measurement Add(Measurement m1, Measurement m2)
{
 if (m1.Units == m2.Units)
 {
  return (m1.Units, m1.Distance + m2.Distance);
 }
 else
 {
  throw new Exception("Units do not match!");
```

```
 }
}
…
Measurement m1 = ("m", 100);
Console.WriteLine(m1.Distance + " " + m1.Units);

Measurement m2 = ("m", 42);
Console.WriteLine(m2.Distance + " " + m2.Units);

Measurement m3 = Add(m1, m2);
Console.WriteLine(m3);
```

Viertes Beispiel: Auch können Entwicklerinnen und Entwickler Typaliase für .NET-Klassen definieren, unabhängig davon, ob diese aus der .NET-Basisklassenbibliothek oder einem NuGet-Paket stammen bzw. selbst definiert sind, z.B.

```
using MyPerson = BO.Person;
```

Anders als beim Int-Array-Alias numbers ist mit einem Klassenaliase auch eine Verwendung bei der Instanziierung gestattet:

```
MyPerson p = new MyPerson();
MyPerson[] pArray = new MyPerson[10];
```

Ein Typalias muss am Beginn einer Datei außerhalb von allen Typimplementierungen (Klassen, Strukturen) stehen. Der Typalias darf vor oder nach den using-Anweisungen für Namensraumimporte und vor oder nach der Namensraumdeklaration stehen. Ausnahme: Wenn der Typalias nicht nur für eine Datei, sondern alle Dateien im Projekt gelten soll, dann muss der Alias vor dem Namensraum stehen und zusätzlich das Schlüsselwort global besitzen. Ein Typalias kann nicht für andere Projekte exportiert werden. Er muss in jedem .NET-Projekt einmal deklariert sein, wenn er verwendet wird.

Listing: Globale Typaliase müssen am Anfang einer Datei stehen. Auf die aktuelle Datei beschränkte Typaliase dürfen auch innerhalb eines Namensraums vorkommen.

```
global using Measurement = (string Units, int Distance);
using BO;

namespace BL;

// Typaliase dürfen im Namensraum stehen
using Numbers = int[];
using DbInt = int?;
using MyPerson = Person;

class MeineKlasse
{
…
}
```

36 Funktionale Programmierung in C# (Delegates / Lambdas)

C# unterstützt funktionale Programmierung insbesondere durch Delegates (seit C# 1.0) und Lambda-Ausdrücke (seit C# 3.0).

36.1 Delegates

Delegaten (engl. Delegates) sind typsichere Zeiger auf Funktionen (Funktionszeiger). Durch Delegaten kann der aufzurufende Code variabel gehalten werden. Sie kommen insbesondere zum Einsatz für die Ereignisbehandlung und für asynchrone Methodenaufrufe. Ein Delegat kann auf mehrere Funktionen zeigen (Multicast Delegate). Beim Aufruf des Delegaten werden alle an den Delegaten gebundenen Funktionen aufgerufen.

C# unterstützt die Definition dieser .NET-Funktionszeiger seit Version 1.0 durch das Schlüsselwort delegate. In dem folgenden Listing wird zunächst ein Delegate GrussFunktion definiert, der zwei Zeichenkettenparameter erwartet und eine Zeichenkette zurückliefert. Danach folgen zwei Funktionsimplementierung für den Delegate GrussFunktion, die nacheinander dem Delegate zugewiesen und genutzt werden.

Eine Funktionsimplementierung nimmt – anders als die Vererbung unter Klassen, die man mit dem Doppelpunkt ausdrückt – nicht expliziten Bezug auf den zu implementierenden Delegate. Eine Funktion ist eine gültige Implementierung eines Delegates schon dann, wenn die Methodensignatur (Anzahl und Typ der Parameter) übereinstimmt (in der Fachsprache: Duck Typing oder: "Wenn es quarkt wie eine Ente und watschelt wie eine Ente, dann ist es eine Ente!").

Wichtig: Bei der Zuweisung einer Funktionsimplementierung zu einem Delegate dürfen hinter dem Methodennamen keine runden Klammern verwendet werden und auch keine Parameter angegeben werden. Die runden Klammern bedeuten, die Methoden soll aufgerufen werden. Die Zuweisung einer Funktionsimplementierung soll noch keinen Aufruf der Implementierung darstellen!

Listing: Eigene Delegate mit zwei Implementierungen

```
public class DelegateBeispiella
 {
  // Delegate definieren
  public delegate string GrussFunktion(string name, string vorname);

  static void Run_EigeneDelegates()
  {
   string e; // Ergebnis
   // Zuweisung einer Implementierung an den Delegate
   GrussFunktion g = Hallo;
   // Aufruf der Funktion, die hinter dem Delegate steht
   e = g("Schwichtenberg", "Holger");
   Console.WriteLine(e);
   // Zuweisung einer anderen Implementierung an den Delegate
   g = GutenTag;
   // Aufruf der Funktion, die jetzt hinter dem Delegate steht
   e = g("Schwichtenberg", "Holger");
   Console.WriteLine(e);
  }
```

```
  // Implementierng #1 Delegate GrussFunktion<T,T>
  static public string Hallo(string name, string vorname)
  {
   return "Hallo " + vorname + " " + name + "!";
  }

  // Implementierng #2 Delegate GrussFunktion<T,T>
  static public string GutenTag(string name, string vorname)
  {
   return "Guten Tag " + vorname + " " + name + "!";
  }
}
```

In dem vorhergehenden Beispiel wird zwar die Syntax für Delegates deutlich, aber das gleiche Ergebnis hätte man leichter erzielen können, indem man die Funktionen Hallo() und GutenTag() direkt aufgerufen hätte.

Ein Delegate kann Methodenparameter sein. Das nächste Listing macht das vorherige Beispiel etwas eindruckvoller, indem hier eine Methode DruckeGruss() exisitiert, die Name, Vorname und einen Funktionszeiger vom Typ GrussFunktion erwartet. So kann man im Hauptprogramm immer einfach DruckeGruss() aufrufen mit ganz unterschiedlichen Logiken der Grußerzeugung.

Listing: Eigene Delegate als Methodenparameter

```
public class DelegateBeispiel1b
 {
  // Delegate definieren
  public delegate string GrussFunktion(string name, string vorname);

  static void Run_DelegateAlsParameter()
  {
   DruckeGruss("Schwichtenberg", "Holger", Hallo);
   DruckeGruss("Schwichtenberg", "Holger", GutenTag);
  }

  // Funktion, die eine Funktion vom Typ GrussFunktion erwartet
  public static void DruckeGruss(string name, string vorname, GrussFunktion
grussfunktion)
  {
   var grussText = grussfunktion(name, vorname);
   Console.WriteLine(grussText);
  }

  // Implementierng #1 Delegate GrussFunktion<T,T>
  static public string Hallo(string name, string vorname)
  {
   return "Hallo " + vorname + " " + name + "!";
  }

  // Implementierng #2 Delegate GrussFunktion<T,T>
  static public string GutenTag(string name, string vorname)
  {
   return "Guten Tag " + vorname + " " + name + "!";
  }
```

```
}
```

36.2 Vordefinierte Delegates Action<T> und Func<T>

Die .NET-Klassenbibliothek bietet im Namensraum System, in dem alle Basisdatentypen enthalten sind, insgesamt 32 vordefinierte generische Delegate-Typen.

16 Delegate-Typen ohne Rückgabewert mit Action<T>	16 Delegate-Typen mit Rückgabewert mit Func<T>
Action	Func<TResult>
Action<T>	Func<T,TResult>
Action<T1,T2>	Func<T1,T2,TResult>
Action<T1,T2,T3>	Func<T1,T2,T3,TResult>
Action<T1,T2,T3,T4>	Func<T1,T2,T3,T4,TResult>
Action<T1,T2,T3,T4,T5>	Func<T1,T2,T3,T4,T5,TResult>
Action<T1,T2,T3,T4,T5,T6>	Func<T1,T2,T3,T4,T5,T6,TResult>
Action<T1,T2,T3,T4,T5,T6,T7>	Func<T1,T2,T3,T4,T5,T6,T7,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8>	Func<T1,T2,T3,T4,T5,T6,T7,T8,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,TResult>
Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>	Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>

Tabelle: Vordefinierte generische Delegate-Typen in der .NET-Klassenbibliotek

Das folgende Listing verzichtet auf eine eigene Delegate-Definition und verwendet stattdessen zwei Varianten des vordefinierten Delegate Action und eine Variante von Func.

Listing: Vordefinierte Delegates

```
public class DelegateBeispiel1c
 {

  public static void VordefinierterDelegate()
  {
   Action<string> log = LogToConsole;
   log("Start...");
   // ...
   log("Läuft...");
   // ...
   log("Erfolgreich!");

   Action<int, string, bool> log2 = LogToConsole2;
```

```
   log2(1, "Start...", true);
   // ...
   log2(2, "Läuft...", false);
   // ...
   log2(3, "Erfolgreich!", true);


   Func<string, string, string> gruss = Hallo;
   gruss("Schwichtenberg", "Holger");

   gruss = GutenTag;
   gruss("Schwichtenberg", "Holger");
  }

  // Implementierung des Delegate Action<T>
  public static void LogToConsole(string text)
  {
   Console.WriteLine($"LOG {DateTime.Now.ToShortTimeString()}: {text}");
  }

  // Implementierung des Delegate Action<T>
  public static void LogToConsole2(int ID, string text, bool withTime)
  {
   Console.WriteLine($"LOG {(withTime ? DateTime.Now.ToShortTimeString() : "")}:
{ID:0000}: {text} ");
  }

  // Implementierng #1 Delegate GrussFunktion<T,T>
  static public string Hallo(string name, string vorname)
  {
   return "Hallo " + vorname + " " + name + "!";
  }

  // Implementierng #2 Delegate GrussFunktion<T,T>
  static public string GutenTag(string name, string vorname)
  {
   return "Guten Tag " + vorname + " " + name + "!";
  }
 }
```

Wichtig: Zu beachten ist, dass die Definitionen eines eigenen Delegates

public delegate string GrussFunktion(string name, string vorname);

und die Nutzung eines vordefinierten Delegatetypen

Func<string, string, string> GrussFunktionVordefiierterDelegate;

nicht kompatibel sind, da es sich um verschiedene Typen handelt, auch wenn Parameteranzahl, Parametertypen und Rückgabetyp kompatibel ist.

36.3 Prädikate mit Predicate<T>

Neben Action<T> und Func<T> gibt es auch noch den vordefinierten Delegattypen Predicate<T> aus Zeiten von .NET 1.0. Ein Prädikat ist ein Funktionszeiger (Delegat) auf eine Methode, die true oder false liefert. Predicate<T> entspricht also System.Func<T, bool>.

Prädikate werden zur Auswahl von Elementen in Listen verwendet. Die Objektmengenklassen in der FCL stellen aus historischen Gründen Methoden bereit, die Predicate<T> erwarten, z.B. die Filter-Funktionen der Array-Klasse wie Find(), FindAll(), FindIndex() und FindLast().

Listing: Einsatz von Predicate<T>

```
// Predicate ist .NET 1.x-Stil: System.Predicate<T> _= _System.Func<T,
bool>
  bool FilterZahlenKleiner10(int x)
  {
   Console.WriteLine("Prüfe Zahl: " + x);
   return x < 10;
  }

  public void PredidateDemo()
  {
   // Datenmenge
   int[] Zahlen = { 1, 30, 5, 10, 15, 20, 3, 9 };
   // Verwendung Lambda-Ausdruck
   Predicate<int> filter = FilterZahlenKleiner10;
   var Ergebnis = Array.FindAll(Zahlen, filter);
   // Ausgabe
   foreach (object Zahl in Ergebnis)
   {
    Console.WriteLine(Zahl);
   }
  }
```

36.4 Lambdas

Ein Lambda ist seit C# 3.0 ist eine stark verkürzte Schreibweise für eine anonyme Methode. Technisch gesehen handelt es sich bei den Lambdas um einen Funktionszeiger (Delegates) und zugleich um anonyme Delegaten, da kein expliziter Name für die Delegate-Klasse vergeben wird. Die Namensvergabe erledigt, wie bei anonymen Typen, der Compiler.

Praxishinweis: Lambda-Ausdrücke sind eine elegante Möglichkeit, Code zu schreiben, der kurz und prägnant ist, insbesondere in Situationen, in denen Sie eine schnelle und einfache Funktion benötigen, ohne eine separate Methode zu definieren. Lambdas sind in den letzten Jahren an immer mehr Stellen vorgerückt, an denen zuvor Methoden geschrieben wurden.

Der Rumpf eines Lambdas wird durch den Operator => knapp gehalten. Der Operator => wird gelesen: "abgebildet auf".

Lambdas gibt es in zwei Formen: Einzeilige Lambdas mit nur einem Ausdruck nach dem => (Expression Lambda) und mehrzeilige Lambdas mit einem Befehlsblock in geschweiften Klammern nach dem => (Statement Lambda).

36.4.1 Einzeilige Lambda-Ausdrücke

Das folgende Listing zeigt eine Reihe von einzeiligen Lambda-Ausdrücken, die nur einen Wert zurückliefern.

Hinweis: Lambda-Ausdrücke, die einen Typ auf einem anderen Typ abbilden (also Beispiele 2 bis 4 in dem folgenden Listing), nennt man eine Projektion.

Listing: Beispiele für einzeilige Lambda-Ausdrücke in C#

```
// Lambda-Ausdrücke deklarieren
Func<int> f0 = () => DateTime.Now.Hour;
Func<int, int> f1 = x => x + 1;
Func<string, string> f2 = s => s.ToUpper();
Func<string, int> f3 = s => s.Length;
Func<string, int, string> f4 = (s, i) => s.Substring(0, i);

// Lambda-Ausdrücke verwenden
Console.WriteLine(f0()); // ergibt 42
Console.WriteLine(f1(10)); // ergibt 11
Console.WriteLine(f2("World Wide Wings")); // ergibt WORLD WIDE WINGS
Console.WriteLine(f3("World Wide Wings")); // ergibt 16
Console.WriteLine(f4("World Wide Wings", 10)); // ergibt "World Wide"
```

Seit C# 10.0 gibt es für die Deklaration von Funktionen auf Basis von Lambda-Ausdrücken abgekürzte Syntaxformen auf Basis von Typherleitung. Dabei gibt der Softwareentwickler nur noch den Typ der Parameter an. Der Rückgabetyp ergibt sich aus dem Code.

Die folgenden Programmzeilen zeigen die verkürzte Variante der mit Lambda-Ausdrücken oben deklarierten Funktionen F0 bis f4.

Listing: Lambda-Ausdrücke deklarieren seit C# 10.0 mit Typherleitung

```
var f0b = () => DateTime.Now.Second;
var f1b = (int x) => x + 1;
var f2b = (string s) => s.ToUpper();
var f3b = (string s) => s.Length;
var f4b = (string s, int i) => s.Substring(0, i);
```

Es ist möglich den Rückgabetyp des Lambda-Ausdrucks ("Lambda Return Type") explizit anzugeben. Dies ist zwar in allen obigen Fällen nicht erforderlich, aber einige Entwickler präferieren explizitere Codierung.

Listing: Lambda-Ausdrücke deklarieren seit C# 10.0 mit explizitem Rückgabetyp

```
var f0c = int () => DateTime.Now.Second;
var f1c = int  (int x) => x + 1;
var f2c = string (string s) => s.ToUpper();
var f3c = int (string s) => s.Length;
var f4c = string (string s, int i) => s.Substring(0, i);
```

Der explizite Rückgabetyp ist nur erforderlich, wenn man einen bestimmten Rückgabetyp erzwingen will, z.B. hier.

Listing: Explicit Lambda Return Type

```
var f10 = byte () => 42; // Rückgabetyp wäre sonst int
var f11 = FileSystemInfo () => new DirectoryInfo(@"c:\Windows"); // wäre sonst Di
rectoryInfo
Console.WriteLine(f10());
```

```
Console.WriteLine(f11().FullName);
```

Seit C# 10.0 kann man einem Lambda-Ausdruck auch Annotationen/Attribute für Parameter und Rückgabewert geben:

```
var f12 = [return:NotNull] ([SensitiveData] string name) => "Hallo " + name;
```

36.4.2 Einsatzbeispiele für Lambda-Ausdrücke

Ein Lambda-Ausdruck kann einen vordefiniertem Delegate oder einen eigenen Delegate realisieren.

Listing: Expression Lambda vs. Statement Lambda

```
public delegate string GrussFunktion(string name, string vorname);
public delegate DateTime Berechnung(int type, byte stunden);

public static void LambdaArten()
  {
   // Beispiel für Expression Lambda mit vordefiniertem Delegate
   Func<string, string, string> expressionLambda1 = (name, vorname)
    => $"Guten Morgen {vorname} {name}!";
   // Beispiel für Expression Lambda mit vordefiniertem Delegate
   Func<int, byte, DateTime> expressionLambda2 = (tage, stunden)
    => DateTime.Now.AddDays(tage).AddHours(stunden);
   // Beispiel für Expression Lambda mit eigenem Delegate
   GrussFunktion expressionLambda3 = (name, vorname)
 => $"Guten Morgen {vorname} {name}!";
   // Beispiel für Expression Lambda mit eigenem Delegate
   Berechnung expressionLambda4 = (tage, stunden)
 => DateTime.Now.AddDays(tage).AddHours(stunden);
   // Beispiel für Statement Lambda mit vordefiniertem Delegate
   Func<string, string, string> statementLambda1 = (name, vorname) =>
   {
    return $"Guten Tag {vorname} {name}!";
   };
   // Beispiel für Statement Lambda mit vordefiniertem Delegate
   Func<int, byte, DateTime> statementLambda2 = (tage, stunden) =>
   {
    return DateTime.Now.AddDays(tage).AddHours(stunden);
   };
   // Beispiel für Statement Lambda mit eigenem Delegate
   GrussFunktion statementLambda3 = (name, vorname) =>
   {
    return $"Guten Tag {vorname} {name}!";
   };
   // Beispiel für Statement Lambda mit vordefiniertem Delegate
   Berechnung statementLambda4 = (tage, stunden) =>
   {
    return DateTime.Now.AddDays(tage).AddHours(stunden);
   };
```

```
   Func<string, string, string> statementLambda5 = (name, vorname) =>
   {
    Trace.WriteLine("GutenTag() wurde aufgerufen!");
    return $"Guten Tag {vorname} {name}!";
   };
  }
```

Listing: Delegate-Beispiel mit Lambda-Ausdruck

```
// Deklaration Delegate GrussFunktion(string,string) -> string
public delegate string GrussFunktion(string name, string vorname);

public static void EigenerDelegate()
{

 // Implementierung des Delegate GrussFunktion(string,string) als Statement Lambd
a
 GrussFunktion GutenTag = (name, vorname) =>
 {
  Trace.WriteLine("GutenTag() wurde aufgerufen!");
  return $"Guten Tag {vorname} {name}!";
 };
 // Übergabe des Expression Lambda
 DruckeGruss("Schwichtenberg", "Holger", GutenTag);
}
```

Ein Lambda-Ausdruck muss keinen Namen besitzen, wenn er als Parameter an eine Funktion übergeben wird. In diesem Fall spricht man von einer anonymen Funktion.

Listing: Beispiel für benannte und unbenannte Lambda-Ausdrücke

```
public static void Run()
{
 // Deklaration Lambda-Ausdruck
 Func<string, string, string> gutenMorgen = (name, vorname) => $"Guten Morgen {vo
rname} {name}!";
 // Nutzung des benannten Lambda-Ausdrucks
 DruckeGrussFunc("Schwichtenberg", "Holger", gutenMorgen);
 // Unbenannter Lambda-Ausdruck (anonyme Funktion)
 DruckeGrussFunc("Schwichtenberg", "Holger", (name, vorname) => $"Guten Abend {vo
rname} {name}!");
}

// Funktion, die eine Funktion (string,string) -> string erwartet
public static void DruckeGrussFunc(string name, string vorname, Func<string, stri
ng, string> grussfunktion)
{
 var grussText = grussfunktion(name, vorname);
 Console.WriteLine(grussText);
}
```

Eine anonyme Funktion kann auf alle Variablen der umgebenden Funktion zugreifen. Seit C# 9.0 kann man dies unterbinden mit dem Zusatz static. Der Entwickler kann sich mit dem Zusatz static davor schützen, versehentlich auf Daten der Umgebung zuzugreifen.

Listing: Nicht-statische vs. statische anonyme Funktionen

```
public static void StatischeAnonymeFunktionen()
  {
   string vorname = "Holger";
   string name = "Schwichtenberg";
```

```
   // normale anonyme Funktion: vorname und name sind nutzbar
   DruckeGrussFunc("Guten Abend", (gruss) => $"{gruss} {vorname} {name}!");
   // statische anonyme Funktion: vorname und name sind NICHT nutzbar
  // DruckeGrussFunc("Guten Abend", static (gruss) => $"{gruss} {vorname}
{name}!");
  }

// Funktion, die eine Funktion (string,string) -> string erwartet
public static void DruckeGrussFunc(string Gruss, Func<string, string>
formatGruss)
  {
   var grussText = formatGruss(Gruss);
   Console.WriteLine(grussText);
  }
```

Seit C# 9.0 ist es in Lambda-Ausdrücken auch erlaubt, mit der Discard-Variable _ anzuzeigen, dass man einen Parameter nicht verwenden will.

```
DruckeGrussFunc("Guten Abend", (_) => $"Hallo {vorname} {name}!");
```

Ein weiteres Einsatzbeispiel für Lambda-Ausdrücke ist der Einsatz als Prädikat (Predicate<T>). Das folgende Listing zeigt drei verschiedene Schreibweisen, um alle Vorstandsmitglieder aus einer Liste zu filtern, die eine bestimmte Bedingung erfüllen; die letzte Schreibweise mit Lambda-Ausdrücken ist die kürzeste und eleganteste.

Listing: Prädikate in C#

```
...
// Prädikate klassische Schreibweise
List<Vorstandsmitglied> JungeVorstandsmitglieder1 =
Vorstandsmitglieder.FindAll(AuswahlJunge);
Console.WriteLine("Junge Vorstandsmitglieder: " +
JungeVorstandsmitglieder1.Count);

// Prädikate mit anonymen Methoden
List<Vorstandsmitglied> JungeVorstandsmitglieder2 =
Vorstandsmitglieder.FindAll(delegate(Vorstandsmitglied v) { return v.Alter < 40;
});
Console.WriteLine("Junge Vorstandsmitglieder: " +
JungeVorstandsmitglieder2.Count);

// Prädikate mit Lambda-Ausdruck
List<Vorstandsmitglied> JungeVorstandsmitglieder3 = Vorstandsmitglieder.FindAll(v
=> v.Alter < 40);
Console.WriteLine("Junge Vorstandsmitglieder: " +
JungeVorstandsmitglieder3.Count);
}

// gehört zu Prädikat klassische Schreibweise!
static public bool AuswahlJunge(Vorstandsmitglied v)
{
   return (v.Alter < 40);
}
```

36.4.3 Mehrzeilige Lambdas

Die folgenden Listings zeigen Beispiele für mehrzeilige Lambdas. Man spricht bei den mehrzeiligen Lambdas von "Statement Lambdas" – im Kontrast zu den "Expression Lambdas", die aus einer Zeile bestehen nur einen Wert zurückliefern.

Listing: Beispiel für einen mehrzeiligen Lambda-Ausdruck mit Rückgabewert in C#

```
Predicate<int> ZahlenKleiner10 = x =>
  {
   Console.WriteLine("Prüfe Zahl: " + x);
   return x < 10;
  };


  // Datenmenge
  int[] Zahlen = {1,30, 5, 10, 15, 20, 3, 9};
  // Verwendung Lambda-Ausdruck
  var Ergebnis = Array.FindAll(Zahlen, ZahlenKleiner10);
  // Ausgabe
  foreach (object Zahl in Ergebnis)
  {
   Console.WriteLine(Zahl);
  }
```

Listing: Beispiel für einen mehrzeiligen Lambda-Ausdruck ohne Rückgabewert in C#

```
  // Deklaration Lambda-Ausdruck ohne Rückgabewert
  Action<int> Ausgabe = x =>
   {
    Trace.WriteLine(x);
    Console.WriteLine(x);
   };
  // Datenmenge
  int[] ZahlenReihe = {1,30, 5, 10, 15, 20, 3, 9};
  // Verwendung Lambda-Ausdruck
  Array.ForEach(ZahlenReihe, Ausgabe);
```

36.4.4 Optionale Lambda-Parameter (seit C# 12.0)

Lambdas erlaubten vor C# 12.0 keine optionalen Parameter. Das hat sich in C# 12.0 geändert. Anstelle dieser Funktion mit optionalem Parameter z

```
public decimal Calc(decimal x, decimal y, decimal z = 1)
{
 return (x + y) * z;
}
```

kann ein Entwickler in C# 12.0 nun auch diesen Lambda-Ausdruck schreiben:

```
var calc = (decimal x, decimal y, decimal z = 1) => (x + y) * z;
```

Das geht auch mit Statement Lambdas. Anstelle dieser Methode mit optionalem Parameter color

```
public void Print(object text, ConsoleColor? color = null)
{
 if (color != null) Console.ForegroundColor = color.Value;
 Console.WriteLine(text);
 if (color != null) Console.ResetColor();
}
```

kann nun dieses Statement Lambda treten:

```
var Print = (object text, ConsoleColor? color = null) =>
{
 if (color != null) Console.ForegroundColor = color.Value;
 Console.WriteLine(text);
 if (color != null) Console.ResetColor();
};
```

37 Ereignisse

Klassen oder einzelne Objekte können Ereignisse auslösen, die von anderen abonniert werden können. Zu einem Ereignis kann es beliebig viele Abonnenten in beliebig vielen Objekten geben. In diesem Fall ruft das Objekt Unterroutinen in allen Abonnenten auf, wenn eine bestimmte Situation eintritt.

Die Definition und die Behandlung von Ereignissen ist in C# komplizierter im Vergleich zu der Vorgehensweise in Visual Basic.

37.1 Definition von Ereignissen

Ein Ereignis ist ein Klassenmitglied, das mit event deklariert wird und sich auf einen Delegaten bezieht, entweder einen selbstdefinierten Delegaten oder einen vordefinierten Delegaten wie Action<T> oder EventHandler<T>. Ereignisse können Instanzen zugeordnet sein (nicht statisch) oder der Klasse zugeordnet sein (statisch). Das folgende Beispiel zeigt drei statische Ereignisse der Klasse Passagier.

```
public class Passagier
{
  public delegate void CheckInStartHandler(Passagier p);
…
  // Ereignis für selbtsdefinierten Delegaten
  public static event CheckInStartHandler CheckInStart;
  // Ereignis für vordefinierten Delegaten
  public static event EventHandler<Passagier> CheckInErfolg;
  // Ereignis für vordefinierten Delegaten
  public static event Action<Passagier, string> CheckInFehler;
…
}
```

Tipp: Eine Vereinfachung bei der Deklaration von Ereignissen ist möglich durch die generische Klasse System.EventHandler<T>. EventHandler<T> steht für einen Delegat mit zwei Eingabeparametern: object sender und einem zweiten Parameter vom dem angegebenen Typ T. Die nicht generische Variante System.EventHandler erwartet als zweiten Parameter einen von System.EventArgs abgeleiteten Typen.

37.2 Ereignis auslösen

Ein spezielles Schlüsselwort zum Auslösen eines Ereignisses (wie *RaiseEvent* in Visual Basic .NET) existiert in C# nicht. Zum Auslösen des Ereignisses kann das Ereignis wie eine Methode aufgerufen werden.

```
if (CheckInStart != null)  {  CheckInStart(this);  }
```

Wichtig: Man muss zuvor immer prüfen, ob überhaupt jemand für das Ereignis registriert ist, sonst kommt es zum Laufzeitfehler "System.NullReferenceException: 'Object reference not set to an instance of an object."

Seit C# 6.0 ist die verkürzte Syntax mit Fragezeichen-Punkt-Operator möglich. Auch hier wird ein Absturz vermieden, wenn es keinen Nutzer des Ereignisses gibt.

```
CheckInStart?.Invoke(this);
```

37.3 Ereignisbehandlung

Auch für die Ereignisbehandlung existieren in C# keine speziellen Schlüsselwörter wie AddHandler, WithEvents und Handles in Visual Basic .NET. In C# muss der Delegat instanziiert werden mit der Ereignisbehandlungsroutine als Parameter und diese so gebildete Instanz muss der Ereignisvariablen der Klasse mit dem Operator + hinzugefügt werden.

```
// Ereignisbehandlung mit explizitem Delegaten
Passagier.CheckInStart += new Passagier.CheckInStartHandler(CheckInGestartet);
…
static void CheckInGestartet(Passagier pass)
{       Demo.Print("Check-In beginnt... für " + pass.GanzerName); }
```

C# unterstützt seit Version 2.0 zur Ereignisbehandlung auch anonyme Methoden, mit denen Programmcode direkt einem Delegaten zugewiesen werden kann. Anstelle des Verweises auf eine entsprechende Ereignisbehandlungsroutine kann der Entwickler mit dem Schlüsselwort delegate nun direkt einen Codeblock (anonyme Methode) binden. Wenn mehrere Ereignisse den gleichen Code ausführen sollen, ist die Implementierung der anonymen Methode auf den Aufruf einer Methode zu beschränken.

```
// Ereignisbehandlung mit anonymer Methode
Passagier.CheckInErfolg +=
    delegate (object sender, Passagier p) {
    CUI.PrintWarning("Passagier Check-In OK: " + p.ToString());
   };
```

Seit C# 3.0 sind Lambda-Ausdrücke zur Ereignisbehandlung möglich.

```
// Ereignisbehandlung mit Statement Lambda
Passagier.CheckInFehler +=
(p, text) => {
    CUI.PrintWarning("Passagier Check-In Fehler: " + p.ToString() + " Fehler: " +
text)
}
```

38 IDisposable / Using-Blöcke

IDisposable ist eine zentrale Schnittstelle in .NET seit .NET Framework Version 1.0. Sie dient dazu, ein Standardverfahren anzubieten, bei dem von einem Objekt verwendete Ressourcen aufgeräumt werden. Man spricht vom IDisposable-Muster (Pattern). Mit Using-Blöcken deklariert man einen Bereich, in dem das Objekt und seine Ressourcen benötigt werden.

38.1 Hintergründe zur Speicher- und Ressourcenverwaltung in .NET

Im Gegensatz zu COM verfügt .NET über eine automatische Speicherverwaltung, die in der Common Language Runtime (CLR) implementiert ist. Die CLR enthält einen Garbage Collector (GC), der im Hintergrund (in einem System-Thread) arbeitet und den Speicher aufräumt. Der Speicher wird allerdings nicht sofort nach dem Ende der Verwendung eines Objekts freigegeben, sondern zu einem nicht festgelegten Zeitpunkt bei Bedarf (Lazy Resource Recovery). Beim Aufräumen des Speichers erzeugt der Garbage Collector einen Baum aller Objekte, auf die es aktuell einen Objektverweis gibt. Der Speicher aller nicht mehr erreichbaren Objekte wird freigegeben.

Der Garbage Collector kann von einer Anwendung nur bedingt beeinflusst werden. Die Anwendung kann mit dem Befehl System.GC.Collect() dem Garbage Collector den Auftrag geben, tätig zu werden. Eine Anwendung kann jedoch eine Speicherbereinigung nicht verhindern. Der Garbage Collector ruft die Destruktoren (alias Finalizer) der .NET-Objekte auf. Die Reihenfolge des Aufrufs und ob der Destruktor überhaupt aufgerufen wird, ist jedoch nicht deterministisch, d.h., es kann sein, dass ein Destruktor nicht aufgerufen wird. Beim Schließen einer .NET-Anwendung werden die Destruktoren der verbliebenen Objekte nicht aufgerufen. Um sich von den deterministischen Destruktoren der Sprache C++ abzuheben, spricht man in .NET von Finalisierung statt von Destruktion.

Es gibt aber Klassen, die nicht nur von der CLR verwalteten Speicher verwenden, sondern auch noch weitere ("externe") Ressourcen, die nicht zum Garbage Collector aufgeräumt werden. Dies sind zum Beispiel:

- Unverwalteter Speicher (z.B. in verwendeten COM-Objekten)
- Geöffnete Dateien
- Geöffnete Netzwerkverbindindungen

38.2 Schnittstelle IDisposable

Klassen, bei denen der Aufruf des Destruktors wichtig ist, weil dabei externe Ressourcen freigegeben werden, müssen dem Disposable-Muster folgen und die Schnittstelle System.IDisposable mit der Methode Dispose() implementieren. In Dispose() sind alle externen Ressourcen freizugeben.

Der Nutzer der Klasse muss am Ende der Verwendung der Methode Dispose() aufrufen.

Listing: Deklaration einer Klasse mit IDisposable

```
using ITVisions;
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace CS10
{
 class Dateisystemzugriff : IDisposable
 {
  System.IO.StreamWriter writer = null;
  bool disposed = false;
  string filepath = "";
  public Dateisystemzugriff(string filePath)
  {
   this.filepath = filepath;
   Console.WriteLine($"Ich öffne die Datei {filePath} im Konstruktor...");
   writer = new System.IO.StreamWriter(filePath, true);
  }

  public void Log(string s)
  {
   writer.WriteLine($"{DateTime.Now}: {s}");
  }

  ~Dateisystemzugriff()
  {
   Console.WriteLine($"Finalizer für Instanz
{nameof(DateisystemClient)}#{filepath}...");
   Dispose();
  }

  public void Dispose()
  {
   Console.WriteLine($"Dispose für Instanz
{nameof(DateisystemClient)}#{filepath}...");
   if (disposed) return;

   // Hier externe Ressourcen freigeben
   writer.Close();
   disposed = true;
   GC.SuppressFinalize(this);
  }
 }
}
```

Listing: Verwendung einer Klasse mit IDisposable ohne Using-Block

```
 class DateisystemClient
 {
  public void Run()
  {

   CUI.MainHeadline("Verwendung ohne Using Block");
   Dateisystemzugriff d1 = new Dateisystemzugriff(@"c:\temp\csharplog.txt");
   for (int a = 1; a < 10; a++)
   {
    d1.Log("Meldung # " + a);
    Console.Write(".");
```

```
    System.Threading.Thread.Sleep(10);
   }
   Console.WriteLine();
   d1.Dispose();
  }
 }
```

38.3 Using-Blöcke

Der Aufruf von Dispose() kann leicht von dem Softwareentwickler vergessen werden. Die Programmiersprachen C# und Visual Basic unterstützen daher ein Programmblockkonstrukt mit Namen Using-Block. Der Using-Block wird eingeleitet durch das Schlüsselwort using. Während es in Viusal Basic .NET ein "End Using" gibt, wird in C# der Block durch geschweifte Klammern begrenzt. Am Ende eines Using-Blocks wird für die im Kopf des Blocks angegebenen Variablen automatisch die Dispose()-Methode aufgerufen.

Hinweis: Das Schlüsselwort using hat in C# eine Doppelbedeutung. Es wird auch für Namensraum-Importe verwendet (siehe Kapitel "Namensräume (Namespaces)").

Listing: Verwendung einer Klasse mit IDisposable mit Using-Block

```
class DateisystemClient
 {
  public void Run()
  {
   CUI.MainHeadline("Verwendung mit Using Block");
   using (Dateisystemzugriff d2 = new
Dateisystemzugriff(@"c:\temp\csharplog.txt"))
   {
    for (int a = 1; a < 10; a++)
    {
     d2.Log("Meldung # " + a);
     Console.Write(".");
     System.Threading.Thread.Sleep(10);
    }
    Console.WriteLine();
   }
  }
 }
}
```

38.4 Vereinfachte Using-Deklarationen (C# 8.0)

Seit C# 8.0 ist es möglich, das Schlüsselwort using für Klassen mit IDisposable-Schnittstelle auch zu verwenden ohne einen expliziten Codeblock mit geschweiftem Klammern { … }. In diesem Fall ist die nach using deklarierte Variable gültig bis zum Ende des umgebenden Blocks; erst wenn dieser endet, erfolgt der automatische Aufruf von Dispose().

```
   try
   {
    CUI.MainHeadline("Verwendung mit Using-Deklaration");
    using Dateisystemzugriff d3 = new
Dateisystemzugriff(@"c:\temp\csharplog.txt");
    for (int a = 1; a < 10; a++)
    {
```

```
    d3.Log("Meldung # " + a);
    Console.Write(".");
    System.Threading.Thread.Sleep(10);
   }
   // d3 ist hier noch gültig
   d3.Log("Ende!");
  }// d3 ist ab jetzt ungültig, d3.Dispose() wird aufgerufen
  catch (Exception ex)
  {
   Console.WriteLine(ex);
  }
```

38.5 IDispose für Strukturen auf dem Stack

Seit C# 8.0 können nicht nur Klassen (Schlüsselwort class) und normale Strukturen (Schlüsselwort struct), sondern auch Strukturen, die nur auf dem Stack leben (Schlüsselwort ref struct) das IDisposable-Pattern realisieren. Allerdings nicht mit Verweis auf die Schnittstelle IDisposable, denn Strukturen auf dem Stack können keine Schnittstellen implementieren. In Strukturen auf dem Stack schreibt man einfach eine Dispose()-Methode. Diese lose Dispose()-Implementierung ruft die .NET-Laufzeitumgebung am Ende von using-Blöcken genauso wie die implementierten Dispose()-Methoden der expliziten IDisposable-Schnittstelle.

Listing: Pattern based Disposable / Disposable ref structs

```
public ref struct Ressource  // immer am Stack, nie am Heap
{
 // C# 8.0: Ableiten von IDisposable nicht möglich für structs
 public void Dispose()
 {
  Console.WriteLine("Ressource Dispose");
 }

}
```

39 Exklusive Zugriffe auf Ressourcen mit lock()

Wenn Sie mit paralleler Codeausführung/Multi-Threading arbeiten, werden Sie eine Möglichkeit benötigen, die parallele Ausführung bestimmter Codeteile zu unterbinden, in denen die paralleler Codeausführung zu unerwünschten Ergebnissen führen könnte. Für exklusive Zugriffe gibt es seit der ersten C#-Version das Schlüsselwort lock().

Bei lock() muss man ein Objekt angeben. Dies sollte keins der Datenobjekte sein, die der Code verändern will, sondern ein Objekt das nur zum Zwecke des Sperrvorgangs existiert. Vor C# 13.0 wurde bei lock() üblicherweise ein Objekt des Types System.Object verwendet:

```
public class Counter
{
    private int count = 0;
    private readonly object lockObject = new object();

    public void Increment()
    {
        lock (lockObject)  // Acquire a lock on lockObject
        {
            count++;
        } // Release the lock when the block is exited
    }

    public int GetCount()
    {
        lock (lockObject)
        {
            return count;
        }
    }
}
```

Seit .NET 9.0/C# 13.0 gibt es für das Sperren von Codeblöcken vor dem Zugriff durch weitere Threads eine neue Klasse System.Threading.Lock, die man nun im Standard in Verbindung mit dem lock-Statement in C# verwenden sollte, "for best Performance" wie Microsoft in der Dokumentation *https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock* schreibt.

Das nächste Listing zeigt ein Beispiel mit dem Schlüsselwort lock und der Klasse System.Threading.Lock.

Listing: Ein lock in C# 13.0 mit der neuen Klasse System.Threading.Lock (Quelle des Beispiels: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock)

```
using System;
using System.Threading.Tasks;

namespace NET9_Console.CS13;

public class Account
{
 // Vor C# 13.0 wurde hier System.Object verwendet statt System.Threading.Lock
 private readonly System.Threading.Lock _balanceLock = new();
```

```
 private decimal _balance;

 public Account(decimal initialBalance) => _balance = initialBalance;

 public decimal Debit(decimal amount)
 {
  if (amount < 0)
  {
   throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot
be negative.");
  }

  decimal appliedAmount = 0;
  lock (_balanceLock)
  {
   if (_balance >= amount)
   {
    _balance -= amount;
    appliedAmount = amount;
   }
  }
  return appliedAmount;
 }

 public void Credit(decimal amount)
 {
  if (amount < 0)
  {
   throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount canno
t be negative.");
  }

  lock (_balanceLock)
  {
   _balance += amount;
  }
 }

 public decimal GetBalance()
 {
  lock (_balanceLock)
  {
   return _balance;
  }
 }
}

class AccountTest
{
 static async Task Main()
 {
  var account = new Account(1000);
  var tasks = new Task[100];
```

```
  for (int i = 0; i < tasks.Length; i++)
  {
   tasks[i] = Task.Run(() => Update(account));
  }
  await Task.WhenAll(tasks);
  Console.WriteLine($"Account's balance is {account.GetBalance()}");
  // Output:
  // Account's balance is 2000
 }

 static void Update(Account account)
 {
  decimal[] amounts = [0, 2, -3, 6, -2, -1, 8, -5, 11, -6];
  foreach (var amount in amounts)
  {
   if (amount >= 0)
   {
    account.Credit(amount);
   }
   else
   {
    account.Debit(Math.Abs(amount));
   }
  }
 }
}
```

Der C#-Compiler übrigens dann aus

```
lock (_balanceLock)
{
  _balance += amount;
}
```

einen Aufruf der EnterScope()-Methode in der Klasse System.Threading.Lock:

```
using (balanceLock.EnterScope())
{
  _balance += amount;
}
```

40 Laufzeitfehler

Das Erzeugen und Behandeln von Ausnahmen ist in der Common Language Runtime (CLR), der Laufzeitumgebung von .NET verankert und daher für alle .NET-Sprachen gleich. Exceptions (Ausnahmen) sind .NET-Objekte, wobei es verschiedene Klassen von Ausnahmen geben kann, die in einer Vererbungshierarchie zueinander stehen. Basisklasse ist System.Exception. Jede Ausnahme stellt Informationen wie eine Fehlerbeschreibung (Message) und die Aufrufliste der Methoden (StackTrace) bereit.

Achtung: Eine .NET-Klasse kann – anders als in Java – nicht deklarieren, welche Fehlertypen sie erzeugt und welche vom Nutzer abgefangen werden müssen (Konzept der Checked Exceptions). Der .NET-Entwickler kann Wissen über mögliche Fehlerarten nur aus der Dokumentation entnehmen.

40.1 Fehler abfangen

C# unterstützt das Konstrukt try...catch...finally, um Laufzeitfehler abzufangen. Dabei kann es mehrere Catch-Blöcke mit unterschiedlichen Ausnahmeklassen geben. Ein catch (Exception ex) fängt alle Fehler ab, weil System.Exception die Oberklasse aller Ausnahmen ist.

Listing: Fehlerbehandlung in C#

```
public static void Run()
  {
   IEnumerable<string> inhalt = null;
   var filename = @"c.\temp\daten.txt";
   try
   {
    inhalt = System.IO.File.ReadLines(filename);
   }
   catch (ArgumentException)
   {
    Console.WriteLine("Ungültiger Dateiname!");
   }
   catch (NotSupportedException ex) when (ex.Message.Contains("format"))
   {
    Console.WriteLine("Ungültiges Format!");
   }
   catch (NotSupportedException ex)
   {
    Console.WriteLine("Nicht unterstützt: " + ex.Message);
   }
   catch (Exception ex)
   {
    Console.WriteLine("Anderer Fehler: " + ex.Message);
   }

   // Inhalt verarbeiten...

  }
```

In C# gibt es seit Version 6.0 auch Exception Filter, mit denen der C#-Entwickler nun zusätzlich zu den Exception-Klassen in den catch-Blöcken mit dem Schlüsselwort when zwischen

verschiedenen Fällen differenzieren kann (siehe Listing). Diese Spracheigenschaft gibt es in Visual Basic .NET schon seit dem Jahr 2002.

Listing: Exception Filter in C# 6

```
    try
    {
     var datei = System.IO.File.ReadLines(filename);
    }
    catch (ArgumentException) when (filename == "")
    {
     Console.WriteLine("Ohne Dateiname macht diese Aktion keinen Sinn!");
    }
    catch (ArgumentException ex) when (ex.Message.Contains("Illegales"))
    {
     Console.WriteLine("Ungültige Zeichen im Dateinamen: " + filename);
    }
    catch (ArgumentException ex)
    {
     Console.WriteLine("Ungültige Angabe: " + filename + ":" + ex.Message);
    }
    catch (NotSupportedException ex) when (ex.Message.Contains("format"))
    {
     Console.WriteLine("Ungültiges Format!");
    }
    catch (NotSupportedException ex)
    {
     Console.WriteLine("Nicht unterstützt: " + ex.Message);
    }
     catch (FileNotFoundException ex)
    {
     Console.WriteLine("Datei " + filename + " nicht gefunden");
    }
    catch (Exception ex)
    {
     Console.WriteLine("Anderer Fehler: " + ex.Message);
    }
```

40.2 Fehler auslösen

Die Anweisung throw ExceptionKlasse erzeugt eine Ausnahme. Neben den in der .NET-Klassenbibliothek vordefinierten Ausnahmen (z.B. System.ArithmeticException, System.ArgumentException, System.FormatException) können eigene anwendungsspezifische Ausnahmeklassen durch Ableitung von System.ApplicationException erzeugt werden.

Seit C# 7.0 ist der Einsatz von throw jetzt auch an Stellen erlaubt, an denen Ausdrücke erwartet werden, z.B. nach dem doppelten Fragezeichen und in Lambda-Ausdrücken.

```
   private Decimal? honorar;
   public Decimal? Honorar
   {
    get => this.honorar;

    // throw ist nun an Stellen erlaubt, wo Ausdrücke erwartet werden, z.B. ??
und Expression Lambdas
```

```
    set => this.honorar = value ??
        throw new ArgumentNullException(nameof(value), "Kein Honorar nicht
erlaubt!");
   }
```

40.3 Eigene Fehlerklassen

In C# kann man auch eigene Fehlerklasse definieren, die dann bei throw verwendet werden dürfen.

```
public class FalscheFlugnummer : System.ApplicationException
 {
  public FalscheFlugnummer(string Beschreibung) : base(Beschreibung) { }
 }
 public class PassagierNichtAufFlugGebucht : FalscheFlugnummer
 {
  public PassagierNichtAufFlugGebucht(string Beschreibung) : base(Beschreibung) {
}
 }
```

41 Modul-Initialisierer

Ein Modul-Initialisierer (engl. Module Initializer) ist eine Methode, die beim Laden eines .NET-Moduls (entspricht einer .NET-Assembly) von der .NET-Laufzeitumgebung automatisch aufgerufen wird. Der Aufruf erfolgt vor allen anderen Codeausführungen. Dies bedeutet, dass bei einem Startmodul ein Modul-Initialisierer vor Main() ausgeführt wird. Bei einem DLL-Modul wird der Modul-Initialisierer vor der ersten Methode, die in der DLL aufgerufen wird, ausgeführt.

Ein Modul-Initialisierer ist eine Methode, die folgende Voraussetzungen erfüllen muss:

- kompiliert mit C# 9.0 oder höher
- Runtime .NET 5.0 oder höher
- Methode ist in einer Klasse, die public oder internal ist
- Methode ist public oder internal
- Methode ist statisch (static)
- Methode ist parameterlos
- Methode ist hat keinen Rückgabewert (void)
- Methode ist nicht-generisch
- Methode ist annotiert mit [System.Runtime.CompilerServices.ModuleInitializerAttribute]

> Es darf mehr als einen Modul-Initialisierer in einer Assembly geben! Alle Modul-Initialisierer werden in der Reihenfolge aufgerufen, wie die Runtime sie im Kompilat findet!

Er folgt ein Beispiel.

Listing: Beispiel für einen Modul-Intialisierer

```
public class ModuleInitializerClass
 {
  [ModuleInitializer]
  public static void ModuleInitializer()
  {
   var ass = System.Reflection.Assembly.GetExecutingAssembly().GetName();
   CUI.Print("Modul wird geladen: " + ass.Name + " v" + ass.Version.ToString(),
ConsoleColor.Cyan);
  }
 }
```

Listing: Hauptprogamm im Hauptmodul

```
class Program
 {
  static void Main(string[] args)
  {
   CUI.H1("C# 9.0 Demos");

Console.WriteLine(System.Runtime.InteropServices.RuntimeInformation.FrameworkDesc
ription + " on " +
System.Runtime.InteropServices.RuntimeInformation.OSDescription);

   Console.WriteLine("Ergebnis: " + new Hilfsklassen.Util().GetValue());

   CUI.H1("Fertig!");
  }
 }
```

Listing: Klassen im Modul Hilfsklassen.dll inklusive zwei Modul-Initialisierern

```
using ITVisions;
using System;
using System.Runtime.CompilerServices;

namespace Hilfsklassen
{
 public class Util
 {
  public int GetValue()
  {
   return 42;
  }
 }
 public class ModuleInitializerClass2
 {
  [ModuleInitializer]
  public static void ModuleInitializer()
  {
   var ass = System.Reflection.Assembly.GetExecutingAssembly().GetName();
   CUI.Print("ModuleInitializerClass2: Modul wird geladen: " + ass.Name + " v" +
ass.Version.ToString(), ConsoleColor.Cyan);
  }
 }
 public class ModuleInitializerClass
 {
  [ModuleInitializer]
  public static void ModuleInitializer()
  {
   var ass = System.Reflection.Assembly.GetExecutingAssembly().GetName();
   CUI.Print("ModuleInitializerClass1: Modul wird geladen: " + ass.Name + " v" +
ass.Version.ToString(), ConsoleColor.Cyan);
  }
 }
}
```

In der folgenden Ausgabe des Beispiels sieht man:

- Der Modul-Initialisierer im Hauptmodul wird vor Main() aufgerufen
- Die beiden Modul-Initialisierer im Modul Hilfsklassen.dll werden erst aufgerufen, wenn das Hauptprogramm erstmals auf etwas in der DLL zugreift, also die Methode Util.GetValue() aufruft.

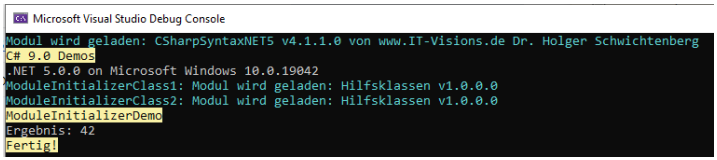


Abbildung: Ausgabe des obigen Beispiels

42 Kommentare und XML-Dokumentation

C# unterstützt drei Arten von Kommentaren:

- Zeilenkommentare, bei denen jede Zeile mit einem // eingeleitet wird
- Blockkommentare, bei denen der Codeblock in /* … */ eingerahmt wird
- XML-Kommentare, bei denen jede Zeile mit /// beginnt.

```
/// <summary>
 /// erbende Klasse
 /// </summary>
 class Experte : Person
 {

  /// <summary>
  /// Kenntnisstand
  /// </summary>
  public Kenntnisse Kenntnisse { get; set; } = Kenntnisse.SehrGut;
  /// <summary>
  /// Themenliste
  /// </summary>
  public List<string> Themen = new List<string>() { ".NET", "C#" };

  /// <summary>
  /// Konstruktor mit Delegation an Basisklasse
  /// </summary>
  /// <param name="name">Name des Experten</param>
  /// <param name="erzeugtAm">Datum der Datensatzerstellung</param>
  /// <param name="kenntnisse">Kenntnisstand</param>
  public Experte(string name, DateTime erzeugtAm, Kenntnisse kenntnisse) :
base(name, erzeugtAm)
  {
   this.Kenntnisse = kenntnisse;
  }

  /// <summary>
  /// Überschriebene Methode zu Ausdruck des Experten
  /// </summary>
  /// <param name="details">Ausdruck von Details</param>
  public override void Drucke(bool details = false)
  {
   base.Drucke(details);
   if (details)
   {
    Console.WriteLine($"Experte für: {String.Join(",", this.Themen)}.");
   }
  }
```

Abbildung: Beispiel für XML-Codekommentare in C#

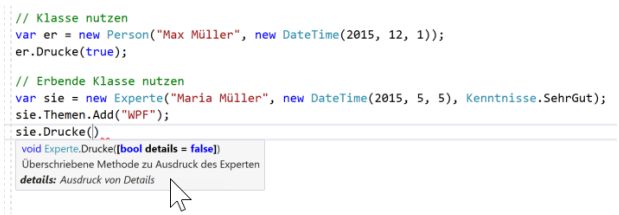


Abbildung: Visual Studio verwendet die XML-Kommentare bei der Eingabehilfe

Praxistipp: Weitere Verwendungsmöglichkeiten der XML-Kommentare ist die Generierung von Hilfedokumenten mit dem Sandcastle Help File Builder (SHFB) [*github.com/EWSoftware/SHFB*] oder die Nutzung in Hilfedokumentation von WebAPIs mit Swagger Open API [*learn.microsoft.com/de-de/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio*].

43 Asynchrone Ausführung mit async und await

In C# 5.0 gab es zwei neue Schlüsselwörter (async und await), die die asynchronen Programmierung erheblich vereinfachen. Eine Methode kann mit async deklarieren, dass sie plant, im Laufe ihrer Ausführung asynchron (ggf. in einem eigenen Thread) weiterzuarbeiten und die Kontrolle an den Aufrufer zurückzugeben. Eine solche asynchrone Methode muss dann ein Task-Objekt (aus der in .NET Framework 4.0 eingeführten Task Parallel Library (TPL)) zurückliefern. Innerhalb der asynchronen Methode wird die Kontrolle dann genau nach dem ebenfalls neuen Schlüsselwort await an den Aufrufer zurückgegeben.

43.1 Verwendung von async und await mit der .NET-Klassenbibliothek

Das nächste Listing zeigt das Beispiel eines asynchronen Datenbankzugriffs mit Connection, Command und DataReader aus ADO.NET. In diesen Klassen gibt es nun zusätzlich zu den bisherigen synchronen Methoden auch asynchrone Methoden. In dem Beispiel ruft das Hauptprogramm Run() eine selbst erstellte asynchrone Methode ReadDataAsync(). In dieser Methode kommen die von der seit ADO.NET 4.5 bereitgestellten asynchronen Methoden OpenAsync() und ExecuteReaderAsync() zum Einsatz, die jeweils mit await aufgerufen werden. Es ist dabei eine Konvention, aber keine Pflicht, dass der Name einer asynchronen Methode auf „async“ endet. Die Ausgabe der Thread-Nummern im Listing dient lediglich dazu, die asynchrone Ausführung in verschiedenen Threads zu belegen (siehe Abbildung).

Listing: Asynchrone Datenbankoperationen mit ADO.NET seit Version 4.5

```
  public static void run()
  {
   Console.WriteLine("Run() #1: Aufruf wird initiiert: Thread=" +
System.Threading.Thread.CurrentThread.ManagedThreadId);
   ReadDataAsync();
   Console.WriteLine("Run() #2: Aufruf ist erfolgt: Thread=" +
System.Threading.Thread.CurrentThread.ManagedThreadId);
  }


  /// <summary>
  /// Asynchroner Download (Rückgabe: nichts)
  /// </summary>
  static private async void ReadDataAsync()
  {
   // Datenbankverbindung asynchron aufbauen
   SqlConnection conn = new SqlConnection(@"data source=.;initial
catalog=WWWings;integrated
security=True;MultipleActiveResultSets=True;App=ADONETClassic");
   await conn.OpenAsync();
   Console.WriteLine("Nach Open Async: Thread=" +
System.Threading.Thread.CurrentThread.ManagedThreadId);
   // Daten asynchron abrufen
   SqlCommand cmd = new SqlCommand("select top(10) * from flug", conn);
   var reader = await cmd.ExecuteReaderAsync();
   Console.WriteLine("Nach ExecuteReaderAsync: Thread=" +
System.Threading.Thread.CurrentThread.ManagedThreadId);
```

```
   // Daten ausgeben
   while (reader.Read())
   {
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine(reader["Abflugort"]);
    Console.ForegroundColor = ConsoleColor.Gray;
   }

   // Verbindung beenden
   conn.Close();
  }
 }
```

```
Run() #1: Aufruf wird initiiert: Thread=10
Run() #2: Aufruf ist erfolgt: Thread=10
Nach Ope Async: Thread=13
Nach ExecuteReaderAsync: Thread=14
Kapstadt
Oslo
Moskau
London
Madrid
Rom
Paris
München
Moskau
Moskau
```

Abbildung: Ausgabe des obigen Listings als Beleg für die asynchrone Ausführung in verschiedenen Thread

43.2 Verwendung von async und await mit eigenen Threads

Das zweite Beispiel zeigt async und await im Einsatz mit der Ausführung einer Aufgabe in einem separaten Thread mithilfe der Task-Klasse von .NET.

```
public async Task<int> MachWasAsync()
{
 Console.WriteLine("MachWasAsync - Start");
 var t = new Task<int>(MachWasIntern);
 t.Start();
  var r = await t;
 Console.WriteLine("MachWasAsync - Ende");
 return r;
}

private int MachWasIntern()
{
 int sum = 0;
 for (int i = 0; i < 10; i++)
 {
  Console.WriteLine(i.ToString());
  sum += i;
 }
 return sum;
}
```

43.3 Weitere Möglichkeiten mit async und await

Seit C# 6.0 darf ein C#-Entwickler die Schlüsselwörter async und await auch in catch- und finally-Blöcken verwenden. Dies ist für Visual Basic .NET nicht vorgesehen.

Asynchrone Methoden, die bisher auf die Rückgabe von Task, Task<T> oder void beschränkt waren, können seit C# 7.0 auch andere Typen zurückgeben, die eine GetAwaiter()-Methode implementieren, die ein Objekt mit der Schnittstelle

```
System.Runtime.CompilerServices.ICriticalNotifyCompletion
```

liefert. So kann ein Entwickler nun zum Beispiel den Typ ValueTask<T> aus dem NuGet-Paket System.Threading.Tasks.Extensions [*https://nuget.org/packages/System.Threading.Tasks.Extensions*] als Rückgabewert verwenden mit dem Vorteil, dass dies ein Value Typ auf dem Stack statt ein Reference Type auf dem Heap ist.

Seit C# 7.1 darf auch die Main()-Routines eines C#-Programms mit async deklariert werden. Folgenden Signaturen sind insgesamt nun bei Main() erlaubt [*https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line*]:

- public static void Main() { }
- public static int Main() { }
- public static void Main(string[] args) { }
- public static int Main(string[] args) { }
- public static async Task Main() { }
- public static async Task<int> Main() { }
- public static async Task Main(string[] args) { }
- public static async Task<int> Main(string[] args) { }

Seit C# 13 können asynchrone Methoden lokale ref-Variablen oder lokale Variablen eines ref struct-Typs deklarieren.

44 Iteratoren

Iteratoren sind ein .NET-Entwurfsmuster zur Erzeugung aufzählbarer Objektmengen, die mit foreach sequentiell vorwärts durchlaufen werden können. Die einfachste Möglichkeit zur Schaffung einer aufzählbaren Menge sind die Collections (siehe Kapitel "Objektmengen"). Darüberhinaus kann der Entwickler eigene aufzählbare Typen mit der Iterator-Implementierung schaffen, was in diesem Kapitel thematisiert ist.

Hinweis: Normale Schleifen mit for(…) und while verwenden keine Iteratoren. Sie greifen auf die Elemente einer Menge über einen Indexer zu.

44.1 Iterator-Implementierung mit yield (Yield Continuations)

Das in C# 2.0 eingeführte Schlüsselwort `yield` vereinfacht die Iterator-Implementierung erheblich. Yield liefert ähnlich wie return einen Wert an den Aufrufer zurück. Anders als beim Einsatz von return beginnt die CLR beim nächsten Aufruf der Methode nicht am Anfang der Routine, sondern setzt die Bearbeitung nach dem `yield` fort. Das nächste Listing zeigt eine einfache Iterator-Klasse, die die deutschen Bundeskanzler aufzählt. Sinn macht ein solcher Iterator, wenn zwischen den Schritten irgendeine Art von Verarbeitung stattfindet, wenn z.B. die Daten aus einem Datenspeicher geholt oder dynamisch berechnet werden.

Listing: Iterator-Implementierung und -Nutzung

```
public class KanzlerListe : IEnumerable
 {
  public IEnumerator GetEnumerator()
  {
   // Logik !!!
   yield return "Adenauer";
   // Logik !!!
   yield return "Erhard";
   // Logik !!!
   yield return "Kiesinger ";
   // Logik !!!
   yield return "Brandt";
   // Logik !!!
   yield return "Schmidt";
   // Logik !!!
   yield return "Kohl";
   // Logik !!!
   yield return "Schröder";
   // Logik !!!
   yield return "Merkel";
   // Ende
   yield break;
  }
 }
class Iteratoren
 {
public static void run()
  {
   KanzlerListe k2 = new KanzlerListe();
   foreach (string s in k2)
   {    Console.WriteLine(s);   }
  }
 }
```

Seit C# 13.0 ist in Methoden, die yield return verwenden, auch die Verwendung von unsafe und ref-Variablen erlaubt.

44.2 Praxisbeispiel für yield

Das vorstehende Beispiel ist nur ein Lernbeispiel. Eine Schleife über eine Menge von Zeichenketten hätte man auch einfacher realisieren können. Ein echtes Praxisbeispiel für den Einsatz von yield finden Sie in der nachstehenden Klasse FlugMengePaging. Diese Klasse implementiert IEnumerable<Flug>, um die in der Datenbank vorhandenen Flüge seitenweise aus der Datenbank auszulesen, wobei die Seitengröße definierbar ist. Der Client soll von dem Paging nichts mitbekommen, wenn er nicht will: Der Client kann mit einer ganz normalen foreach-Schleife über die Datensätze iterieren. Optional kann der Client das Ereignis SeitenWechsel(), das die Klasse FlugMengePaging auslöst, abonnieren und damit über den Seitenwechsel informiert werden.

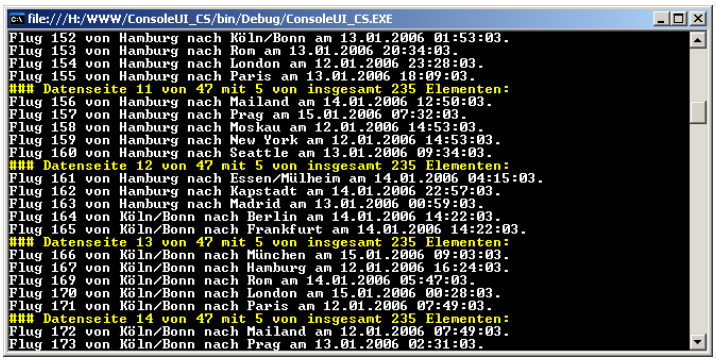


Abbildung: Nutzung der Klasse FlugMengePaging

Das folgende Listing zeigt die Implementierung der Klasse FlugMengePaging, die zwei Generische Klassen der .NET-Klassenbibliothek verwendet:

- Zum einen die generische Variante von IEnumerable: IEnumerable<Flug>
- Zum anderen die generische Klasse EventHandler<> zur Deklaration eines Ereignisses.

Listing: Praxisbeispiel zum Einsatz von Yield, Ereignissen und Generics

```
/// <summary>
/// Klasse für Ereignisparameter beim Paging in der Geschäftslogik
/// </summary>
public class PagingInfo : System.EventArgs
 {
  public long AnzahlObjekteGesamt;
  public long SeitenGroesse;
  public long AnzahlSeiten;
  public long AktuelleSeite;
  public long AnzahlObjekteInAktuellerSeite;

  public PagingInfo(long AnzahlObjekteGesamt, long AnzahlSeiten, long
SeitenGroesse, long AktuelleSeite, long AnzahlInAktuellerSeite)
  {
   this.AnzahlObjekteGesamt = AnzahlObjekteGesamt;
```

```
   this.AnzahlSeiten = AnzahlSeiten;
   this.SeitenGroesse = SeitenGroesse;
   this.AnzahlObjekteInAktuellerSeite = AnzahlInAktuellerSeite;
   this.AktuelleSeite = AktuelleSeite;
  }
 }

 /// <summary>
 /// FlugMenge ist die typisierte Menge von Flug-Objekten, die mithilfe der
Klasse
 /// <see cref="System.Collections.Generic.List"/> implementiert ist. Diese
Variante holt immer
 /// nur eine definierbare Menge (Attribut SeitenGroesse) aus der Datenbank.
 /// </summary>
 public class FlugMengePaging : IEnumerable<Flug>
 {
  private int _SeitenGroesse = 10;
  /// <summary>
  /// Maximale Anzahl von Objekten, die in einer Datenseite abgeholt werden
  /// </summary>
  public int SeitenGroesse
  {
   get { return _SeitenGroesse; }
   set { _SeitenGroesse = value; }
  }
  // Ereignis beim Wechsel der Datenseite
  public event EventHandler<PagingInfo> SeitenWechsel;
  public FlugMengePaging(int SeitenGroesse)
  {
   this.SeitenGroesse = SeitenGroesse;
  }
  #region IEnumerable<Flug> Members
  public IEnumerator<Flug> GetEnumerator()
  {
   int Anzahl = new FlugBLManager().Count();
   int Seiten = Anzahl / SeitenGroesse;

   for (int i = 0; i < Seiten; i++)
   {
    // Nächste Datenseite aus Datenbank abholen
    FlugMenge ff = FlugBLManager.HoleAlle(SeitenGroesse, i * SeitenGroesse + 1);
    // Ereignis auslösen
    if (SeitenWechsel != null) SeitenWechsel(this, new PagingInfo(Anzahl, Seiten,
                                          SeitenGroesse, i + 1, ff.Count));
    // Elemente der aktuellen Seite in einer Schleife zurückgeben
    foreach (Flug f in ff)
    {     yield return f;    }
   }
   yield break;
  }
```

44.3 Asynchrone Streams / await foreach (seit C# 8.0)

Seit C# 8.0 kann der Softwareentwickler asynchrone Iteratoren mit der Schnittstelle System.Collections.Generic.IAsyncEnumerable<T> schaffen und darüber mit await foreach(…) iterieren. Das Beispiel zeigt:

- GetDataSteam(): Simuliert eine datensendende Messstelle; sendet kontinuierlich und endlos alle 250 Millisekunden eine Zahl (hier Zufallszahl). Wie bei synchronen Iteratoren kommt yield zum Einsatz.

- PrintData(): Gibt die eingehenden Zahlen der Messstelle aus. Prüft, ob Abbruch via CancellationTokenSource gefordert wird.
- Main(): Hauptprogramm, das den Datenempfang startet und dann darauf wartet, dass ein Benutzer die EINGABE-Taste bedient, was den Abbruch auslöst.

Listing: Asynchroner Stream

```
using ITVisions;
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace CS80
{
 class AsyncStreamDemos
 {
  public async void Main()
  {
   CUI.Headline(nameof(AsyncStreamDemos));
   CancellationTokenSource cts = new CancellationTokenSource();
   await PrintData(cts);

   CUI.Print("---> Hauptprogramm wartet auf RETURN", ConsoleColor.Yellow);
   Console.ReadLine();

   CUI.Print("Hauptprogramm lößt Abbruch des Datenempfangs aus...",
ConsoleColor.Yellow);
   cts.Cancel();

   CUI.Headline("Hauptprogramm endet!");
  }


  /// <summary>
  /// Empfang der Daten von Stream und Ausgabe
  /// </summary>
  /// <param name="cts">Abbruchoption</param>
  public async Task PrintData(CancellationTokenSource cts)
  {
   // NEU in C# 8.0: await foreach!
   await foreach (var nextValue in GetDataStream())
   {
    CUI.Print($"{nextValue:000000}", ConsoleColor.Cyan);
    if (cts.IsCancellationRequested)
    {
     CUI.PrintError("!!!Abruch der Messdatenausgabe!!!");
     return;
    }
   }
  }

  /// <summary>
  /// Simuliert den Empfang von Daten von einer Messstelle
```

```
  /// Erzeugt dafür 100 Zufallszahlen als Stream, alle 250ms eine neue Zahl
  /// </summary>
  static async IAsyncEnumerable<int> GetDataStream()
  {
   try
   {
    for (; ; )
    {
     await Task.Delay(250);
     yield return new System.Random().Next(1000000);
    }
   }
   finally
   {
    Console.WriteLine("GetDataStream: Finally");
   }
  }
 }
}
```

Abbildung: Ausgabe des obigen Listings. Der Benutzer hat nach kurzer Zeit EINGABE gedrückt.

45 Zeigerprogrammierung

Für die Zeigerprogrammierung bietet C# seit Version 1.0 das Schlüsselwort unsafe. Seit C# 7.0 gibt es eine sicherere Option (Managed Pointer).

45.1 Zeigerprogrammierung mit unsafe

Niemand möchte unsicheren Code schreiben, doch die Programmiersprache C# kennt eine gleichnamige Option (unsafe). Innerhalb von unsicherem Code können in C# Zeiger und Zeigerarithmetik verwendet werden. Diese Operationen werden dann nicht von der Common Language Runtime verifiziert und können zu Programmabstürzen führen. Bei Visual Basic .NET gibt es keine in die Sprachsyntax eingebaute Möglichkeit, Zeiger und Zeigerarithmetik zu nutzen. Das wäre nur über Umwege über die Klassenbibliothek möglich. Wenn Sie derartige Low-Level-Funktionen wirklich nutzen wollten, sollten Sie C# oder C++ / CLI verwenden.

Achtung: Es gibt nur wenige sinnvolle Einsatzgebiete für Zeigerarithmetik in .NET. Ein solcher Fall liegt bei sehr umfangreichen Array-Operationen vor. Da die CLR bei jedem Array-Zugriff die Array-Grenzen prüft, kann durch Einsatz von Zeigerarithmetik ein erheblicher Leistungsgewinn erzielt werden – allerdings auf Kosten der Zuverlässigkeit der Anwendung.

Mit dem Schlüsselwort unsafe können ganze Unterroutinen markiert werden; es besteht auch die Möglichkeit, einen unsafe-Block innerhalb einer Unterroutine zu erzeugen. Voraussetzung für die Kompilierung einer Anwendung mit unsicherem Code ist die Verwendung der Compiler-Option /unsafe.

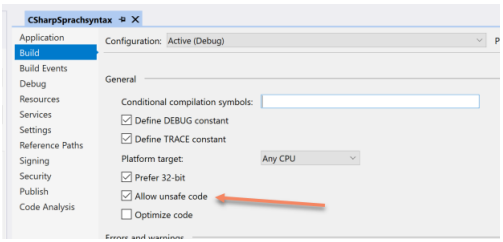


Abbildung: Einstellen der Compileroption "unsafe" in den Projekteigenschaften in Visual Studio

Listing: Unsicherer Code in C#

```
class Zeiger
 {
  unsafe static void ZeigerTest(int* x) // x ist ein Zeiger auf ein Integer32
  {
   int* y; // y ist ein Zeiger auf ein Integer32
   int z = 10; // z ist ein Integer32
   y = &z; // y zeigt auf den Speicherplatz von z
   *x = *x * *y; // Der Platz, auf den x zeigt, soll mit dem Ergebnis des
Produktes aus dem Inhalt von x und y gefüllt werden
   int* r; // r ist ein Zeiger auf ein Integer32
   // Achtung: Das produziert Unsinn!
   r = y + 1;  // r soll nun auf den Speicherplatz zeigen, der 4 Plätze hinter y
liegt
   Demo.Print(*r); // gebe den Inhalt aus, auf den r zeigt
  }
  public static void run()
```

```
  {
   int i = 5;
   unsafe
   {
    ZeigerTest(&i); // Rufe ZeigerTest mit einem Zeiger auf den Speicherplatz von
i auf
   }
   Demo.Print(i);
  }
```

Auch für unsafe-Blöcke hat Microsoft Verbesserungen in C# 7.3 eingebaut. Die Allokierung von Speicher auf dem Stack mit stackalloc war bisher nicht möglich in Verbindung mit einer prägnanten Array-Initialisierung. Erst seit C# 7.3 kann man schreiben:

```
   unsafe
   {
    var a2 = stackalloc int[3] { 45, 2, 57 }; // seit C# 7.3
    var a3 = stackalloc int[] { 45, 2, 57 }; // seit C# 7.3
    var a4 = stackalloc[] { 45, 2, 57 }; // seit C# 7.3
     }
```

Zuvor musste man die Array-Elemente mühsam einzeln initialisieren:

```
  unsafe
   {
    var a1 = stackalloc int[3];    // bisher schon erlaubt
    a1[0] = 45;
    a1[1] = 2;
    a1[2] = 57;
     }
```

Die verkürzte Array-Initialisierung kann nun auch außerhalb von unsafe-Blöcken in Verbindung mit den in C# 7.2 eingeführten Typ Span<T> [*msdn.microsoft.com/de-de/magazine/mt814808.aspx*] zum Einsatz kommen:

```
Span<int> a5 = stackalloc[] { 1, 2, 3 }; // seit C# 7.3
```

Schon seit der ersten Version von C# gibt es mit "fixed" deklarierte Variaben, die nicht vom Garbage Collector verschoben werden können und nur in Strukturen (struct { }), nicht in Klassen (class { }) vorkommen dürfen. Als "Indexing movable fixed Buffers" bezeichnet Microsoft die Möglichkeit, dass mit "fixed" deklarierte Variablen einfacher zu handhaben sind.

Das Befüllen und Auslesen eines Fixed Array erforderte in C# bis einschließlich Version 7.2 immer einen zusätzlichen fixierten Zeiger, wie das nächste Listing.

Listing: Alte Handhabung fixierter Arrays mit fixierten Zeigern (vor C# 7.3)

```
  unsafe struct Daten
  {
   public fixed int Zahlen[7];
  }

  /// <summary>
  /// vor C# 7.2
  /// </summary>
  class BerechnungAlt
  {
   static Daten s = new Daten();

   unsafe public void Berechnen()
   {
```

```
    fixed (int* ptr = s.Zahlen)
    {
     for (int i = 0; i < 7; i++)
     {
      ptr[i] = new System.Random().Next(1, 49);
     }

     int p1 = ptr[5];
     Console.WriteLine(p1);
    }
   }
  }
```

Seit C# 7.3 kann man darauf verzichten, siehe nächstes Listing.

Listing: Vereinfachte Handhabung fixierter Arrays seit C# 7.3

```
   /// <summary>
   /// Ab C#7.3
   /// </summary>
   class BerechnungNeu
   {
    static Daten s = new Daten();

    unsafe public void Berechnen()
    {
     for (int i = 0; i < 7; i++)
     {
      s.Zahlen[i] = new System.Random().Next(1, 49);  // geht nicht vor C# 7.3
     }

     int p2 = s.Zahlen[5];   // geht nicht vor C# 7.3
     Console.WriteLine(p2);
    }
   }
```

45.2 Zeigerprogrammierung mit ref (Managed Pointer)

Zeigerprogrammierung war in C# lange nur bei Methodenparametern und im Rahmen sogenannter unsafe-Blöcke möglich. Das bisher bei den Methodenparametern verwendete ref-Schlüsselwort dehnt Microsoft in C# 7.0 auch auf lokale Variablen und Methodenrückgabewerte aus. Dabei verwendet man das Schlüsselwort ref sowohl bei der Deklaration des Zeigers ref typ name (vgl. in C++: Typ*) als auch um einen Zeiger auf eine Variable zu erhalten: ref name (vgl. C++: & name). Im Untergrund arbeiten sogenannte Managed Pointer.

Das folgende Beispiel zeigt aber, dass im Gegensatz zu C++ in C# eine kontrollierte Variante der Zeigerprogrammierung zum Einsatz kommt. Während eine vergleichbare Befehlsfolge in C++ den Zeiger z im Speicher verschieben würde, wirkt das +=10 in C# 7.0 sich auf den Inhalt statt dem Zeiger aus. Die Variable z enthält danach einen Zeiger auf den Wert 42.

```
int i = 32;
ref int z = ref i;
z+=10;
```

Das nächste Listing zeigt den Einsatz von ref bei dem Rückgabewert einer Methode. Die Methode GetExperte() erhält ein Array und liefert ein Element als Zeiger zurück. Der Aufrufer ändert bei

der Verwertung des Rückgabewertes also das Array. Eine Methode kann aber nicht einen Zeiger auf eine lokale Variable innerhalb der Methode zurückgeben.

Hinweis: Solche Zeiger mit ref sind auch nicht anwendbar bei der Deklaration von Klassenattributen als Fields und Properties, in asynchronen und anonymen Methoden, Iteratoren, Lambda- und LINQ-Ausdrücken.

Listing: Einsatz von Zeigern als Rückgabewert einer Methode

```
  /// <summary>
  /// Diese Funktion liefert die Speicherstelle eines Array-Elements, nicht den
Wert!
  /// </summary>
  static public ref string GetReiseziel(string[] namen, int position)
  {
   if (namen.Length > 0) return ref namen[position];
   throw new IndexOutOfRangeException($"Experte #{nameof(position)} nicht
gefunden.");
  }


  /// <summary>
  /// nutzt die Funktion GetReiseziel()
  /// </summary>
  static public void DemoRefReturns2()
  {
   string[] orte = { "Rom", "Paris", "Oslo", "Istanbul", "Moskau" };
   ref string ort4 = ref GetReiseziel(orte, 3);
   Console.WriteLine("Ort vorher: {0}", ort4); // --> "Istanbul"
   // ändert das Array, da ref!
   ort4 = "Athen";
   Console.WriteLine("Ort nun: {0}", orte[3]); // --> "Athen"
  }
```

In C# 7.0 bis 7.2 ist es nicht möglich, einen Managed Pointer, der ja immer bei der Deklaration schon eine Zuweisung braucht, nachträglich auf eine andere Speicherstelle zu verschieben. Erst C# 7.3 unterstützt das Ref Local Reassignment.

```
int i = 32;
int k = 42;
ref int z = ref i;
Console.WriteLine("z=" + z);
z += 5;
Console.WriteLine("z=" + z);
```

Die Ausgabe ist erst z=32 und dann z=37, da z+=5 nicht den Zeiger verschiebt, sondern den Wert ändert. Bisher nicht erlaubt war, einen bestehenden Zeiger an eine andere Speicherstelle neu zuzuweisen. Folglich bemängelte der Compiler nachstehende Ergänzung in C# 7.0 bis 7.2:

```
z = ref k;
Console.WriteLine("z=" + z);
```

Das ist aber in Version 7.3 nun möglich unter dem Namen "Ref Local Reassignment", sodass die dritte Ausgabe z=42 lautet.

46 Abfrageausdrücke / Language Integrated Query (LINQ)

46.1 Einführung und Motivation

Language Integrated Query (LINQ) ist eine allgemeine Such- / Abfragesprache, die schon seit dem .NET Framework 3.5 in der .NET-Klassenbibliothek und der Sprachsyntax der Sprachen C# (seit Version 3.0) und Visual Basic .NET (seit Version 9.0) verankert ist.

Das Problem, das LINQ zu lösen versucht, lässt sich so beschreiben: Jede Art von Datenspeicher (z.B. Objektmengen im Hauptspeicher, Datenbanktabellen, XML-Dokumente, Verzeichnisdienste) besitzt eine Möglichkeit zur Suche nach Elementen. Bei Datenbanken ist dies in der Regel die Sprache Structured Query Language (SQL), bei XML-Dokumenten XPath oder XQuery und bei Verzeichnisdiensten LDAP. Für Objektmengen im Hauptspeicher gibt es keinen Standard oder De-Facto-Standard. Innerhalb der .NET-Klassenbibliothek findet man unterschiedliche Such- und Abfragemöglichkeiten, z.B. DataView-Objekte für DataTable-Objekte. Auch die Methoden Find() und FindAll(), mit denen man unter Angabe eines Prädikats in Objektmengen aus dem Namensraum System.Collections suchen kann, lassen sich dabei als eine Abfragesprache bezeichnen. Alle diese Abfragesprachen unterscheiden sich hinsichtlich ihrer Mächtigkeit und auch hinsichtlich ihrer Syntax, sodass man für diese verschiedenen Datenspeicher unterschiedliche Befehlssätze beherrschen muss. Erinnert sei an dieser Stelle auch noch daran, dass es zwar einen Standard für SQL gibt, aber es dennoch Unterschiede zwischen der SQL-Syntax verschiedener Datenbankmanagementsysteme gibt.

LINQ tritt an, eine allgemeine Such- und Abfragesyntax für alle Arten von Datenspeichern zu definieren. Unterhalb der LINQ-Abfrageebene werden die Abfragen durch LINQ-Provider in andere Sprachen (z.B. SQL, XPath oder LDAP) übersetzt oder direkt auf dem Datenspeicher ausgeführt.

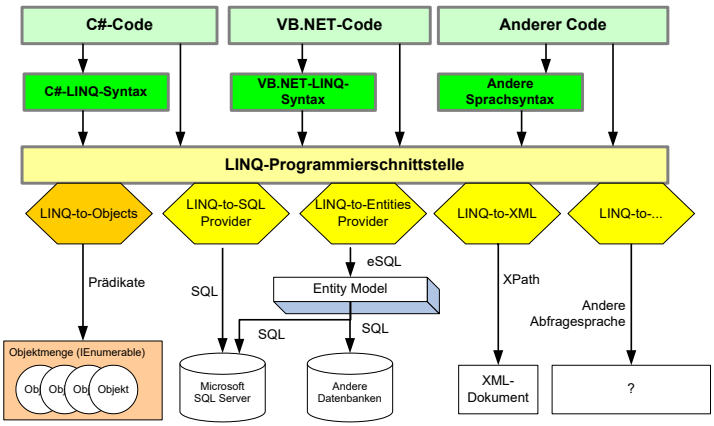


Abbildung: Architektur von LINQ

Neben der Vereinheitlichung der Sprachen bietet LINQ noch einen Vorteil: Während bisher Sprachen wie SQL, XPath und LDAP aus der Sicht des Sprachcompilers nur Zeichenkettenliterale waren, ist die Abfragesyntax nun in der Sprachsyntax bzw. Klassenbibliothek verankert. Der große Vorteil von LINQ ist, dass die Sprachcompiler die Syntax prüfen können und die Entwicklungsumgebung IntelliSense-Unterstützung anbieten kann. Dies ist mit »externen« Suchsprachen, die der Compiler nur als Zeichenkette sieht, nicht möglich.

46.2 LINQ-Provider

Dieser Abschnitt dokumentiert die zum Redaktionsschluss verfügbaren und dem Autor bekannten LINQ-Provider.

> **Hinweis:** LINQ-Provider haben meistens einen Namen, der mit LINQ to … beginnt (z.B. LINQ to XML). Einige wenige Provider verwenden noch die alte Benennungsweise mit einem vorangestellten Kürzel (z.B. hieß LINQ to XML früher XLINQ).

46.2.1 LINQ-Provider von Microsoft im .NET

Microsoft bietet seit .NET Framework 3.5 die Möglichkeit zur Abfrage von

- .NET-Objektmengen, die die Schnittstelle IEnumerable unterstützen (LINQ to Objects)
- Microsoft SQL Server-Datenbanken (LINQ to SQL, früher: DLINQ),
- ADO.NET-DataSets (LINQ to DataSet) und
- XML-Daten (LINQ to XML, früher: XLINQ).

Seit .NET Framework 3.5 Service Pack 1 liefert Microsoft noch zusätzlich:

- LINQ to Entities im Entity Framework: Abfrage von relationalen Datenbanken (nicht nur Microsoft SQL Server wie bei LINQ to SQL)
- LINQ to DataService: Steuerung von WCF-Datendiensten mit Open Data Protocol (OData)

- In Entity Framework Core liefert Microsoft ebenfalls einen LINQ-Provider mit.

In diesem Buch werden nur LINQ-to-Objects und Parallel-LINQ behandelt. Die anderen LINQ-Varianten setzen umfangreiche Kenntnisse zu den entsprechenden Klassenbibliotheken voraus, die außerhalb des Fokus dieses Buchs liegen.

46.2.2 Andere LINQ-Provider

Mittlerweise gibt es neben den im .NET Framework mitgelieferten Providern eine Reihe von Anbietern (kommerzielle und Open Source), so genannte LINQ-Provider für verschiedene Datenquellen.

46.2.3 Formen von LINQ

Es gibt zwei grundsätzliche Formen der LINQ-Unterstützung:

- Abfrage über Mengen, die IEnumerable unterstützen: Diese Abfragen fallen alle unter LINQ to Objects und werden von LINQ im RAM ausgeführt.
- Abfrage über Mengen, die IQueryable unterstützen: Diese Abfrage werden von einer datenquellenspezifischen LINQ-Implementierung ausgeführt. LINQ übergibt dieser Implementierung die Abfrage in Form eines Ausdrucksbaums (Expression Tree). Es ist der Implementierung überlassen, wie die Abfrage erfolgt (z.B. Umsetzung in SQL oder XPath oder Aufruf eines Webservices). Der Einsatz von IQueryable ist wesentlich komplexer als der Einsatz von IEnumerable, denn bei IQueryable werden die LINQ-Abfragen zunächst in einen Ausdrucksbaum (Expression Tree) umgewandelt. Dieser sprachneutrale Ausdrucksbaum wird dann an den LINQ-Provider übergeben, der diesen Baum in die jeweilige providerspezifische Anfragesyntax übersetzt.

46.2.4 Einführung in die LINQ-Syntax

Es gibt zwei Syntaxformen für LINQ: Die Abfragesyntax (Originalbezeichnung: Query Expression Syntax) und die Methodensyntax (Originalbezeichnung: Extension Method Syntax). Die Abfragesyntax ist eleganter, in der Praxis muss man in vielen Fällen beide Syntaxformen mischen, denn viele Befehle sind nur in der Methodensyntax verfügbar.

LINQ-Abfragesyntax

Die Grundstruktur eines LINQ-Befehls in der Abfragesyntax ist

```
from... where... orderby ... select...
```

Die Syntax von LINQ ist an die Datenbankabfragesprache SQL angelehnt, allerdings wird das from immer vorangestellt. Der Grund für diese Abweichung von SQL liegt darin, dass Entwicklungsumgebungen in der Lage sein sollen, dem Entwickler Hilfen bei der Eingabe (IntelliSense) zu geben. Dies kann eine Entwicklungsumgebung aber nur, wenn zu Beginn klar ist, auf welche Menge sich die Abfrage bezieht. Dies ist aber nicht die einzige Abweichung von der SQL-Syntax.

Die folgende Beschreibung liefert eine komplette formale Definition der LINQ-Abfragesyntax. Alle diese hier genannten Begriffe (außer den Platzhaltern id, expr, source, key, query, condition und ordering) sind Schlüsselwörter der Sprache C# (seit 3.0) bzw. Visual Basic (seit 9.0) und werden von der Entwicklungsumgebung Visual Studio (seit 2008) auch wie Sprachschlüsselwörter eingefärbt.

Listing: Syntaxbeschreibung für die LINQ-Abfragesyntax (C#)

```
from id in source
{ from id in source |
  join id in source on expr equals expr [ into id ] |
  let id = expr |
  where condition |
  orderby ordering, ordering, … }
  select expr | group expr by key
[ into id query ]
```

Listing: Syntaxbeschreibung für die LINQ-Abfragesyntax (Visual Basic .NET)

```
From id In source
{ from id In source |
  Join id in source On expr Equals expr [ Into id ] |
  Let id = expr |
  Where condition |
  Take x |
  Skip x |
  Order By ordering, ordering, … }
  Select expr | Group expr By key
  Aggregate x in source
[ Into id query ]
  Distinct
```

An den obigen Syntaxbeschreibungen wird deutlich, dass gar nicht alle Sprachelemente von SQL in der LINQ-Abfragesyntax (d. h. durch eigene Sprachelemente) unterstützt werden. Beispielsweise fehlen in C# DISTINCT und TOP. Dies bedeutet aber nicht, dass diese Funktionalität in LINQ-Abfragen nicht verfügbar wäre. Es bedeutet nur, dass sie in der LINQ-Abfragesyntax nicht verfügbar sind. Es gibt aber noch eine LINQ-Methodensyntax. In Visual Basic existieren mehr Befehle in der Abfragesyntax.

Beispiele

Vor der Diskussion der Methodensyntax sollen zunächst zwei Beispiele (jeweils in C# und Visual Basic) gezeigt werden.

Beispiel: Abfrage einer Menge von Zeichenketten

In diesem ersten Beispiel werden aus einer Liste von Monaten diejenigen Monate gefiltert, deren Namen vier Zeichen lang sind. Von den Monatsnamen werden nur die ersten drei Zeichen weiterverarbeitet. Die Liste wird lexikalisch aufsteigend sortiert. Das Ergebnis ist also Jul, Jun und Mär.

Listing: Filtern in einer Liste von Zeichenketten (C#)

```
  public static void Beispiel1()
  {
   // Datendefinition (=Datenquelle)
   string[] AlleMonate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
"Juli", "August", "September", "Oktober", "November", "Dezember" };

   // LINQ-Abfrage
   IEnumerable<string> Monate4 = from Monat in AlleMonate
                where Monat.Length == 4
                orderby Monat
                select Monat.Substring(0, 3);

   // Nutzung des Abfrageergebnisses
   foreach (string Monat in Monate4)
   {
    Console.WriteLine(Monat);
   }
  }
```

Listing: Filtern in einer Liste von Zeichenketten (Visual Basic .NET)

```
Public Sub Beispiel1()
  ' Datendefinition (=Datenquelle)
  Dim AlleMonate As String() = {"Januar", "Februar", "März", "April", "Mai",
"Juni", "Juli", "August",
                                "September", "Oktober", "November", "Dezember"}

  ' LINQ-Abfrage
  Dim Monate4 As IEnumerable(Of String) = From Monat In AlleMonate _
   Where Monat.Length = 4 _
   Order By Monat _
   Select Monat.Substring(0, 3)

  ' Nutzung des Abfrageergebnisses
  For Each Monat As String In Monate4
   Console.WriteLine(Monat)
  Next
 End Sub
```

Beispiel: Abfrage einer Menge von Objekten des Typs Process

Im zweiten Beispiel werden aus der Liste der laufenden Prozesse diejenigen herausgefiltert, die weniger als 700.000 Bytes Speicher benötigen. Die Datenmenge wird in diesem Fall von der statischen Methode GetProcesses() in der FCL-Klasse System.Diagnostics.Process geliefert. Von den gefilterten Prozessen wird der Name und die Speichermenge ausgegeben.

Listing: Filtern der Prozessliste (C#)

```
  public static void Beispiel2()
  {
   // LINQ-Abfrage
   var Prozesse =
   from p in System.Diagnostics.Process.GetProcesses()
   where p.WorkingSet64 < 700000
   select new { p.ProcessName, p.WorkingSet64 };

   // Nutzung des Abfrageergebnisses
   foreach (var Prozess in Prozesse)
   {
    Console.WriteLine(Prozess.ProcessName + ": " + Prozess.WorkingSet64);
   }
  }
```

Listing:Filtern der Prozessliste (Visual Basic .NET)

```
Public Sub Beispiel2()
  'LINQ-Abfrage
  Dim Prozesse = _
   From p In System.Diagnostics.Process.GetProcesses() _
   Where (p.WorkingSet64 < 700000) _
   Select New With {p.ProcessName, p.WorkingSet64}

  ' Nutzung des Abfrageergebnisses
  Dim Prozess
  For Each Prozess In Prozesse
   Console.WriteLine(Prozess.ProcessName & ": " & Prozess.WorkingSet64)
  Next
 End Sub
```

Hinweis: In dem zweiten Beispiel ist der Einsatz des Schlüsselwortes var anstelle eines konkreten Typnamens bzw. Dim ohne Datentyp zu beachten. Der Grund dafür ist, dass durch die Reduktion der Prozessliste auf die Attribute ProcessName und WorkingSet64 ein anonymer Typ entsteht.

Wichtig: Es gibt eine wichtige Voraussetzung, damit die LINQ-Abfragesyntax in MSIL (alias CIL) übersetzt werden kann: Der Namensraum System.Linq muss importiert sein, also in C#:

using System.Linq;

Häufig wird diese Bedingung übersehen. Dies erkennt man an der Fehlermeldung »Could not find an implementation of the query pattern for source type '…'«.

Da select, where, from, etc. ja Schlüsselwörter der Programmiersprachen C# und Visual Basic sind, stellt sich der kritische Leser sicherlich die Frage, warum dieser Import notwendig erfüllt sein muss. Vor .NET Framework 3.5 gab es keine Schlüsselwörter, die von Referenzen und Importanweisungen abhängig waren. Der Grund liegt in diesem Fall darin, dass der Compiler die LINQ-Abfragesyntax in einem ersten Übersetzungsschritt in LINQ-Methodensyntax übersetzt. Diese Methoden sind Erweiterungsmethoden für bestehende Typen. Wenn diese Erweiterungsmethoden aber nicht verfügbar sind, schlägt die Übersetzung fehl.

LINQ-Methodensyntax

Wie bereits im vorangegangenen Abschnitt erwähnt, sind alle LINQ-Anweisungen intern als Methodenaufrufe realisiert. So wird z.B. das Schlüsselwort where der Abfragesyntax auf die Erweiterungsmethode Where() abgebildet, orderby ist realisiert durch OrderBy() und select durch Select(). Durch die Aneinanderreihung der Methodenaufrufe können komplexe Abfragen definiert werden.

Abfragesyntax	Methodensyntax

// LINQ-Abfrage in Abfragesyntax
IEnumerable<string> Monate4 =
from Monat in AlleMonate
where Monat.Length == 4
orderby Monat
select Monat.Substring(0, 3);
	// LINQ-Abfrage in Methodensyntax
	IEnumerable<string> Monate4 =
	AlleMonate
	.Where(Monat => Monat.Length == 4)
	.OrderBy(Monat => Monat)
	.Select(Monat => Monat.Substring(0,3));

Tabelle: Vergleich von Abfragesyntax und Methodensyntax an einem Beispiel

Tatsächlich existiert nur für einen sehr kleinen Teil der Möglichkeiten von LINQ eine Repräsentation in der Abfragesyntax. Viele Möglichkeiten sind – insbesondere in C# – nur in der Methodensyntax verfügbar, z.B. Top(), Skip(), Distinct(), Min(), Average() etc.

Um die Monate 6 bis 8 in der Liste zu ermitteln, kann man mit Skip() die ersten fünf überspringen und dann mit Take() die nächsten drei auswählen.

Listing: Beispiel in Methodensyntax

```
// LINQ-Abfrage in Methodensyntax
IEnumerable<string> SommerMonate =
    AlleMonate
    .Select(Monat => Monat.Substring(0, 3))
    .Skip(5).Take(3);
```

Die Methodensyntax ist nicht so elegant wie die Abfragesyntax. Der Entwickler kann aber die beiden Syntaxformen miteinander kombinieren, indem er den Ausdruck in Abfragesyntax in runden Klammern einschließt und auf diesem Ausdruck dann die Erweiterungsmethoden anwendet.

Listing: Beispiel in gemischter Syntax

```
// LINQ-Abfrage in gemischter Syntax
IEnumerable<string> SommerMonate =
```

```
    (from Monat in AlleMonate
    select Monat.Substring(0, 3))
    .Skip(5).Take(3);
```

Hinweis: In Visual Basic ist die Abfragesyntax umfangreicher als in C#. In C# kann man aber auch alle LINQ-Befehle nutzen, zum Teil ist die Anwendung aber wesentlich uneleganter als in Visual Basic.

Es gibt zur Laufzeit keinen Unterschied zwischen den beiden Syntaxformen. Auch die Mischung der Syntaxformen hat keinen Nachteil, denn die Klammerung sorgt nicht dafür, dass der Teilausdruck vorher ausgewertet wird. LINQ-Ausdrücke werden immer erst bei ihrer Verwendung ausgeführt (verzögerte Ausführung). Eine Ausnahme bilden die Konvertierungsmethoden ToArray(), ToDictionary(), ToList() und ToLookup(). Diese vier Methoden sorgen allerdings dafür, dass der davorstehende LINQ-Befehl sofort ausgeführt wird.

Übersicht über die LINQ-Befehle

Die folgende Tabelle zeigt die Liste aller in .NET 3.5 / 4.0 verfügbaren LINQ-Befehle. LINQ-Befehle werden auch LINQ-Operatoren genannt.

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
Aggregate			Eigene Aggregatfunktionen	–
All		Aggregate … In … Into All()	Liefert *true*, wenn alle Elemente einer Menge die angegebene Bedingung erfüllen	–
Any		Aggregate … In … Into Any()	Liefert *true*, wenn mindestens ein Element der Menge die angegebene Bedingung erfüllt	EXISTS
Average			Mittelwert (arithmetischer Durchschnitt)	AVG
Cast	from Typ x in Menge	From … As …	Typumwandlung aller Elemente der Menge	–
Concat			Vereinigungsmenge zweier Mengen	UNION
Contains			Prüft, ob die Menge ein bestimmtes Element enthält	IN
Count		Aggregate … In … Into Count()	Liefert die Anzahl der Elemente in der Menge in Form einer 32-Bit-Ganzzahl (Typ Int32)	COUNT
Distinct		Distinct	Entfernt alle doppelten Elemente in der Liste	DISTINCT
ElementAt			Liefert das Element in der Menge an einer bestimmten Stelle (Index)	–

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
ElementAtOrDefault			Liefert das Element in der Menge an einer bestimmten Stelle (Index) oder einen Standardwert, wenn der Index negativ oder größer als die Anzahl der Elemente ist	–
Empty			Erstellt eine leere Menge vom angegebenen Typ	–
Except			Vergleicht zwei Mengen und liefert nur diejenigen Elemente, die in der ersten Menge (die Menge, auf die die Methode angewendet wird), aber nicht in der zweiten Menge (die Menge, die als Parameter angegeben wird) vorhanden sind	–
First			Das erste Element einer Menge. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen. Wenn es kein Element gibt, tritt ein Laufzeitfehler auf.	–

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
FirstOrDefault			Das erste Element einer Menge oder ein Standardwert (bei Referenztypen *null* bzw. *Nothing*), wenn die Menge leer ist. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen.	–
GroupBy	group … by … into …	Group … By … Into …	Gruppiert eine Menge nach dem angegebenen Kriterium	GROUP BY
GroupJoin	join … in … on … equals … into …	Group Join … In … On …	Verbindet zwei Mengen durch einen OUTER JOIN	JOIN
Intersect			Liefert die Schnittmenge zweier Mengen	–
Join	join … in … on … equals …	Join … In … On … Equals …	Verbindet zwei Mengen durch einen INNER JOIN	JOIN
Last			Liefert das letzte Element einer Menge	–
LastOrDefault			Liefert das letzte Element einer Menge oder einen Standardwert, wenn die Menge leer ist	–
LongCount		Aggregate … In … Into LongCount()	Liefert die Anzahl der Elemente in der Menge in Form einer 64-Bit Ganzzahl (Typ Int64)	COUNT

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
Max		Aggregate … In … Into Max()	Ermittelt den maximalen Wert einer Menge	MAX
Min		Aggregate … In … Into Min()	Ermittelt den minimalen Wert einer Menge	MIN
OfType			Liefert alle Elemente einer Menge, die Instanzen einer bestimmten Klasse sind	–
OrderBy	orderby	Order By	Sortiert eine Menge aufsteigend	ORDER BY
OrderByDescending	orderby … descending	Order By … Descending	Sortiert eine Menge absteigend	ORDER BY DESC
Range			Erzeugt eine Menge mit den numerischen Werten von *n* bis *m*	–
Repeat			Erzeugt eine Menge mit n-Mal dem gleichen Element	–
Reverse			Umkehren der Reihenfolge	
Select	select	Select	Bestimmt die Daten und bildet die Elemente, die aus einer Menge erstellt werden	SELECT
SelectMany			Durchläuft Mengen, die selbst Mitglieder anderer Mengen sind und liefert eine flache Liste	–

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
SequenceEqual			Prüft, ob zwei Mengen identisch sind hinsichtlich der Anzahl, Reihenfolge und Inhalt der Elemente	–
Single			Das erste Element einer Menge. Wenn es kein Element gibt oder wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.	–
SingleOrDefault			Das erste Element einer Menge. Wenn es kein Element gibt, wird der Standardwerte (bei Referenztypen *null* oder *Nothing*) geliefert. Wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.	–
Skip		Skip	Überspringt die ersten *n* Elemente einer Menge und liefert den Rest	–
SkipWhile		Skip While	Überspringt so lange Elemente, wie eine Bedingung erfüllt wird und liefert den Rest	–
Sum		Aggregate … In … Into Sum()	Summiert die Elemente einer Menge	SUM
Take		Take	Liefert die ersten *x* Elemente einer Menge	TOP

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
TakeWhile		Take While	Liefert so lange Elemente, wie eine Bedingung erfüllt wird	–
ThenBy	orderby …, …	Order By …, …	Angabe eines weiteren aufsteigenden Ordnungskriteriums bei einer Sortierung	ORDER BY
ThenByDescending	orderby …, … descending	Order By …, … Descending	Angabe eines weiteren absteigenden Ordnungskriteriums bei einer Sortierung	ORDER BY
ToArray			Konvertiert eine Menge zu einem Array	–
ToDictionary			Konvertiert eine Menge zu einer generischen Dictionary<K,T>-Menge	–
ToList			Konvertiert eine Menge zu einer generischen List<T>- Menge	–
ToLookup			Konvertiert eine Menge zu einer generischen Look-up<K,T>-Menge.	–
Union			Vereint zwei Mengen zu einer	UNION
Where	where	Where	Filtern der Eingabemenge	WHERE

Tabelle: LINQ-Befehle

Neben den LINQ-Befehlen kann man auch die Methoden der .NET-Klassenbibliothek in LINQ-Abfragen verwenden. Sinnvoll sind z.B. die Methoden der Klassen System.String (z.B. StartsWith()), System.DateTime (z.B. AddYears() und System.Math (z.B. Round()). Mit LINQ to

Objects kann man prinzipiell alle Methoden der .NET Klassenbibliothek und auch eigene Methoden in eigenen Geschäftsobjekten nutzen. Mit anderen LINQ-Providern ist dies nur dann möglich, wenn es für die Methode eine Entsprechung in der Basissyntax gibt. Dies gilt bei LINQ to SQL im Wesentlichen nur für einige Methoden der Klassen System.String, System.Math und System.DateTime. Andere Methoden und selbstdefinierte Methoden haben keine Entsprechung in SQL und können daher auch nicht in LINQ to SQL genutzt werden.

Achtung: Ob die Reihenfolge der Befehle entscheidend ist, hängt von dem LINQ-Provider ab. Bei LINQ to Objects ist

```
from x in Zahlen where x < 50 orderby x  select x
```

viel schneller als

```
from x in Zahlen orderby x where x < 50 select x
```

Bei LINQ to Entities gibt es keinen Unterschied, denn die zugrundeliegende Datenbank wird dies optimieren.

46.3 LINQ to Objects

Mit LINQ to Objects wird die Abfrage von Objektmengen im Hauptspeicher bezeichnet. Abgefragt werden können alle Objektmengen, die entweder die Schnittstelle IEnumerable oder ihr generisches Pendant IEnumerable<T> unterstützen. Dies sind also die Klassen in System.Collections (z.B. ArrayList, Hashtable, Queue und Stack), die Klassen in System.Collections.Generic (z.B. List<T>, SortedDictionary<T>, Queue<T> und Stack<T>), die Klasse System.Array sowie spezielle Mengen wie DataRowCollection, DataColumnCollection, DirectoryEntries und ManagementObjectCollection. Da IEnumerable bzw. IEnumerable<T> Voraussetzungen für das Funktionieren der foreach-Schleife sind, besitzt praktisch jede Menge in der .NET-Klassenbibliothek eine der beiden Schnittstellen. Für LINQ to Objects ist es unerheblich, ob die Menge vom .NET Framework erzeugt wird oder von eigenem Programmcode.

46.3.1 LINQ to Objects mit elementaren Datentypen

Am Beispiel einer Menge von Zahlen in Form eines Arrays vom Typ Int32 soll die Anwendung von LINQ-Befehlen auf elementaren Datentypen gezeigt werden.

Gegeben sind zwei Zahlenmengen:

Listing: Definition der Zahlenmenge

```
int[] Zahlen1 = { 15, 4, 11, 3, 19, 8, 16, 7, 12, 5, 9, 20, 1, 4, 8, 13, 14, 4, 1
};
int[] Zahlen2 = { 12, 5, 31, 24, 29, 20, 13, 31 };
```

Das folgende Listing enthält zahlreiche Fragestellungen in Bezug auf diese beiden Zahlenmengen und den Weg, die Lösung mit LINQ zu ermitteln. Das jeweilige Ergebnis wird aus Platzgründen hier nicht abgedruckt. Durch den Programmcode zu diesem Buch können Sie dies jedoch selbst ausprobieren.

Listing: Anwendungsbeispiele von LINQ to Objects auf Zahlenmengen

```
  private static void Demo_LTO_Zahlen()
  {
   int i;
   double d;

   string s = "Geben Sie die Zahlen aus, die kleiner als 10 sind.";
   var Ergebnis =
       from n in Zahlen1
       where n < 10
```

```
       select n;
   Print(Ergebnis, s);

   s = "Geben Sie die Zahlen, die kleiner als 10 sind, aufsteigend sortiert
aus.";
   Ergebnis =
       from n in Zahlen1
       where n < 10
       orderby n // optional
       select n;
   Print(Ergebnis, s);

   s = "Geben Sie die Zahlen, die kleiner als 10 sind, absteigend sortiert aus.";
   Ergebnis =
       from n in Zahlen1
       where n < 10
       orderby n descending
       select n;
   Print(Ergebnis, s);

   s = "Geben Sie die Zahlen, die kleiner als 10 sind, absteigend sortiert aus."
+
       "Eliminieren Sie alle Duplikate.";
   Ergebnis =
       (from n in Zahlen1
        where n < 10
        orderby n descending
        select n).Distinct();
   Print(Ergebnis, s);

   s = "Geben Sie die vierte bis achte Zahl aus.";
   Ergebnis =
       (from n in Zahlen1
        where n < 10
        select n).Skip(3).Take(4);
   Print(Ergebnis, s);

   s = "Geben Sie die erste Zahl aus!";
   i =
      (from n in Zahlen1
       select n).First();
   Print(i, s);

   s = "Geben Sie die letzte Zahl aus!";
   i =
      (from n in Zahlen1
       select n).Last();
   Print(i, s);

   s = "Geben Sie die 10. Zahl aus!";
   i =
      (from n in Zahlen1
       select n).ElementAt(9);
   Print(i, s);

   s = "Geben Sie die 50. Zahl aus! (Fangen Sie den Fehler ab!)";
   i =
      (from n in Zahlen1
       select n).ElementAtOrDefault(49);
   Print(i, s);

   s = "Geben Sie die Anzahl der Zahlen aus.";
   i =
      (from n in Zahlen1
```

```
      select n).Count();
   Print(i, s);

   s = "Geben Sie nur die niedrigste Zahl aus.";
   i =
      (from n in Zahlen1
       select n).Min();
   Print(i, s);

   s = "Geben Sie nur die höchste Zahl aus.";
   i =
     (from n in Zahlen1
      select n).Max();
   Print(i, s);

   s = "Geben Sie den Durchschnitt aus.";
   d =
     (from n in Zahlen1
      select n).Average();
   Print(d, s);

   s = "Geben Sie die Summe aus.";
   d =
     (from n in Zahlen1
      select n).Sum();
   Print(d, s);

   s = "Geben Sie das Produkt aller Werte aus.";
   d =
     (from n in Zahlen1
      select n).Aggregate((summe, wert) => summe *= wert);
   Print(d, s);

   s = "Gruppieren Sie die Werte.";
   IEnumerable<IGrouping<int, int>> GruppeErgebnis =
       (from n in Zahlen1
        group n by n);
   Print(GruppeErgebnis, s);

   s = "Geben Sie die Häufigkeit eines jeden Werts aus!";
   IDictionary<int, int> GruppeHaeufigkeit =
         (from n in Zahlen1
          group n by n into g
          select new { Wert = g.Key, Anzahl = g.Count() }
          ).ToDictionary(y => y.Wert, y => y.Anzahl);
   Print(GruppeHaeufigkeit, s);

   s = "Verbinden Sie die Zahlenmengen 1 und 2:";
   Ergebnis = (from n in Zahlen1 select n).Union(from n2 in Zahlen2 select n2);
   Print(Ergebnis, s);

   s = "Verbinden Sie die Zahlenmengen 1 und 2 und sortieren Sie das Ergebnis:";
   Ergebnis = (from n in Zahlen1 select n).Union(from n2 in Zahlen2 select
n2).OrderBy(n => n);
   Print(Ergebnis, s);

   s = "Bilden Sie die Schnittmenge aus den Zahlenmengen 1 und 2.";
   Ergebnis = (from n in Zahlen1 select n).Intersect(from n2 in Zahlen2 select
n2).OrderBy(n => n);
   Print(Ergebnis, s);

   s = "Schließen Sie die Zahlen aus Zahlenmengen 2 in Menge 1 aus.";
   Ergebnis = (from n in Zahlen1 select n).Except(from n2 in Zahlen2 select
n2).OrderBy(n => n);
```

```
   Print(Ergebnis, s);

   s = "Prüfen Sie, ob die Zahlenmenge 1 und 2 die gleichen Zahlen in der
gleichen Reihenfolge enthalten.";
   bool Erfuellt = (from n in Zahlen1 select n).SequenceEqual(from n2 in Zahlen2
select n2);
   Print(Erfuellt, s);

   s = "Prüfen Sie, ob die Zahl 20 in der Menge vorkommt.";
   Erfuellt =
       (from n in Zahlen1
        orderby n descending
        select n).Contains(20);
   Print(Erfuellt, s);

   s = "Prüfen Sie, ob Zahlen größer als 20 in der Menge vorkommen.";
   Erfuellt =
       (from n in Zahlen1
        orderby n descending
        select n).Any(n => n > 20);
   Print(Erfuellt, s);

   s = "Prüfen Sie, ob alle Zahlen kleiner 20 sind.";
   Erfuellt =
       (from n in Zahlen1
        orderby n descending
        select n).All(n => n < 20);
   Print(Erfuellt, s);

   s = "Filtern Sie alle Integer-Werte heraus!";
   Ergebnis =
   (from n in Zahlen1 select n).OfType<int>();

   Print(Ergebnis, s);

   s = "Wandeln Sie alle Zahlen in Byte-Werte um!";
   var kleineZahlen =
     (from n in Zahlen1 select n).Cast<byte>();
   foreach (var x in kleineZahlen)
   {
    Console.WriteLine(x);
   }
   Print(kleineZahlen, s);
  }
```

Das obige Listing nutzt zur Ausgabe die selbstdefinierte Methode Print(). Es muss aber mehrere Überladungen von Print() geben, da die LINQ-Abfragen unterschiedliche Ergebnisse liefern können:

- Viele der obigen LINQ-Abfragen liefern wieder eine Zahlenmenge zurück. Der konkrete Datentyp, der zurückgeliefert wird, ist von den eingesetzten Methoden abhängig. Alle diese Klassen besitzen jedoch die Schnittstelle IEnumerable<int>. Zum Durchlaufen des Ergebnisses ist eine einfache Schleife ausreichend.
- Durch das Gruppieren von Elementen ohne das Schlüsselwort into entstehen zwei verschachtelte Objektmengen des Typs IEnumerable<IGrouping<int, int>>. Die obere Menge repräsentiert dabei die Gruppen, die untergeordnete Menge die Elemente in jeder Gruppe. Zum Durchlaufen des Ergebnisses ist eine geschachtelte Schleife notwendig. Diese Form des Gruppierens bezeichnet man als hierarchisches Gruppieren.
- Durch das Gruppieren von Elementen mit dem Schlüsselwort into entsteht ein neuer anonymer Typ, der das Gruppierungskriterium und die zusammengefassten Daten anderer Mitglieder des

Ausgangstyps enthält. Das Ergebnis ist ein Dictionary-Objekt mit zwei Int32-Werten: IDictionary<int, int>. Diese Form des Gruppierens entspricht dem flachen Gruppieren aus SQL. Trotz der Verwendung von into kann man hierarchisches Gruppieren erreichen, wenn man in dem anonymen Typ auf die Gruppe selbst verweist, z.B. from p in System.Diagnostics.Process.GetProcesses group p by p.ProcessName into g select new { Name = g.Key, Anzahl = g.Count(), Max = g.Max(p => p.WorkingSet64), ProzesseInDieserGruppe = g };

Listing: Ausgaberoutinen für die Ergebnisse der LINQ-Abfragen (Auswahl)

```
priate static void Print(IEnumerable<int> Nums, string s)
  {
   HeadLine(s);
   foreach (int x in Nums)
   {
    Console.WriteLine(x);
   }
  }

  private static void Print(IDictionary<int, int> gruppe, string s)
  {
   HeadLine(s);
   foreach (var x in gruppe)
   {
    Console.WriteLine(x.Key + ": " + x.Value);
   }
  }

  private static void Print(IEnumerable<IGrouping<int, int>> Gruppen, string s)
  {
   HeadLine(s);
   foreach (IGrouping<int, int> x in Gruppen)
   {
    Console.WriteLine("---- " + x.Key);
    foreach (int i in x)
    {
     Console.WriteLine(i);
    }
   }
  }
```

46.3.2 LINQ to Objects mit komplexen Typen des .NET Framework

Die Anwendung von LINQ to Objects auf komplexe Datentypen unterscheidet sich von der Anwendung auf elementare Datentypen wie folgt:

- Bei LINQ to Objects mit elementaren Datentypen wurde die in dem from-Ausdruck deklarierte Laufvariable selbst für Bedingungen, Sortierungen und Berechnungen verwendet. Bei komplexen Datentypen muss mithilfe der Laufvariablen Bezug auf ein Mitglied des Objekts genommen werden.
- LINQ to Objects mit elementaren Datentypen liefert in der Regel eine Menge des Eingabetyps zurück. Bei komplexen Datentypen kann alternativ ein anonymer Typ zurückgegeben werden, der nur eine Teilmenge der Mitglieder des Ausgangstyps enthält. Dies nennt man eine Projektion.

Beispiel

In dem folgenden Beispiel werden LINQ-Befehle auf einer Menge von Objekten des Typs System.Diagnostics.Process angewendet. Die statische Methode GetProcesses() der Klasse

System.Diagnostics.Process liefert eine Liste der laufenden Prozesse auf einem System in Form eines Arrays mit Instanzen von System.Diagnostics.Process.

Listing: Anwendungsbeispiele von LINQ to Objects auf eine Menge von Objekten des Typs System.Diagnostics.Process

```
private static void Demo_LTO_Prozesse()
  {
   Process[] Prozesse = Process.GetProcesses();

   Process p;
   long i;
   double d;

   string s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes
Speicher verbrauchen.";
   var Ergebnis =
       from n in Prozesse
       where n.WorkingSet64 < 3000000
       select n;
   Print(Ergebnis, s);

   s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes Speicher
verbrauchen. Sortieren Sie die Liste aufsteigend nach Speicherverbrauch.";
   Ergebnis =
   from n in Prozesse
   where n.WorkingSet64 < 3000000
   orderby n.WorkingSet64 // optional
   select n;
   Print(Ergebnis, s);

   s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes Speicher
verbrauchen. Sortieren Sie die Liste absteigend nach Speicherverbrauch.";
   Ergebnis =
   from n in Prozesse
   where n.WorkingSet64 < 3000000
   orderby n.WorkingSet64 descending // optional
   select n;
   Print(Ergebnis, s);

   s = "Geben Sie die Prozesse aus. Eliminieren Sie alle Duplikate.";
   Ergebnis =
       (from n in Prozesse
        select n).Distinct();
   Print(Ergebnis, s);

   s = "Geben Sie den vierten bis achten Prozess aus in der nach
Speicherverbrauch aufsteigend sortierten Liste aller Prozesse, die mehr als
1.000.000 Bytes verbrauchen.";
   Ergebnis =
       (from n in Prozesse
        where n.WorkingSet64 > 1000000
        orderby n.WorkingSet64
        select n).Skip(3).Take(4);
   Print(Ergebnis, s);

   s = "Geben Sie den ersten Prozess aus in der nach Speicherverbrauch
aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes
verbrauchen.";
   p =
      (from n in Prozesse
       where n.WorkingSet64 > 1000000
       orderby n.WorkingSet64
       select n).First();
```

```
   Print(p, s);

   s = "Geben Sie den letzten Prozess aus in der nach Speicherverbrauch
aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes
verbrauchen.";
   p =
      (from n in Prozesse
       where n.WorkingSet64 > 1000000
       orderby n.WorkingSet64
       select n).Last();
   Print(p, s);

   s = "Geben Sie den 10. Prozess aus in der nach Speicherverbrauch aufsteigend
sortierten Liste
aller Prozesse, die mehr als 1.000.000 Bytes verbrauchen.";
   p =
      (from n in Prozesse
       where n.WorkingSet64 > 1000000
       orderby n.WorkingSet64
       select n).ElementAt(9);
   Print(p, s);

   s = "Geben Sie den 150. Prozess aus! (Fangen Sie den Fehler ab!)";
   p =
      (from n in Prozesse
       select n).ElementAtOrDefault(149);
   Print(p, s);

   s = "Geben Sie die Anzahl der Prozesse aus (mit einem LINQ-Statement!)";
   i =
      (from n in Prozesse
       select n).Count();
   Print(i, s);


   s = "Geben Sie nur den niedrigsten Speicherverbrauch aus";
   i =
      (from n in Prozesse
       select n).Min(n => n.WorkingSet64);
   Print(i, s);

   s = "Geben Sie nur den höchsten Speicherverbrauch aus";
   i =
      (from n in Prozesse
       select n).Max(n => n.WorkingSet64);
   Print(i, s);

   s = "Geben Sie den durchschnittlichen Speicherverbrauch aus";
   d =
      (from n in Prozesse
       select n).Average(n => n.WorkingSet64);
   Print(i, s);

   s = "Geben Sie die Summe des Speicherverbrauchs aus";
   i =
      (from n in Prozesse
       select n).Sum(n => n.WorkingSet64);
   Print(i, s);

   s = "Gruppieren Sie die Prozesse nach Namen.";
   IEnumerable<IGrouping<string, Process>> GruppeErgebnis =
       (from n in Prozesse
        group n by n.ProcessName);
   Print(GruppeErgebnis, s);
```

```
   s = "Geben Sie die Häufigkeit eines jeden Prozessnamens aus!";
   IDictionary<string, int> GruppeHaeufigkeit =
         (from n in Prozesse
          group n by n.ProcessName into g
          select new { Name = g.Key, AnzProzess = g.Count() }
          ).ToDictionary(y => y.Name, y => y.AnzProzess);
   Print(GruppeHaeufigkeit, s);

   s = "Starten Sie einen neuen Prozess (Notepad) und ermitteln Sie, durch einen
Vergleich der Prozessliste vorher und nachher, welche Prozesse neu hinzugekommen
sind. (Geben Sie die Process-ID und den Prozessnamen aus!)";
   Process neupro = Process.Start(@"C:\Windows\notepad.exe");
   neupro.WaitForInputIdle();
   Process[] Prozesse2 = Process.GetProcesses();

   //Print((from p1 in Prozesse where p1.ProcessName=="notepad" select p1),
"Test");
   //Print((from p2 in Prozesse2 where p2.ProcessName=="notepad" select p2),
"Test");
   IEnumerable<int> ProzessListe = (from n2 in Prozesse2 select
n2.Id).Except(from n in Prozesse select n.Id);
   Print(ProzessListe, s);

   //var ProzessListe2 = from p in System.Diagnostics.Process.GetProcesses()
select p.ProcessName;

   s = "Listen Sie die Prozesse mit ihren Threads auf.";
   var ProzesseMitThreads =
      (from n in Prozesse
       select new { n, n.Threads }
      );
   HeadLine(s);
   foreach (var x in ProzesseMitThreads)
   {
    Console.WriteLine(x.n);
    try
    {
     foreach (ProcessThread y in x.Threads)
     {
      Console.WriteLine(y.StartTime);
     }
    }
    catch (Exception)
    {
    }
   }

   s = "Geben Sie zu jedem Prozess die Anzahl der Threads aus!";
   var ProzesseMitThreadCount =
    (from n in Prozesse
     where n.Id > zehn
     select new { n, n.Threads.Count }
    );
   HeadLine(s);
   foreach (var m in ProzesseMitThreadCount)
   {
    Console.WriteLine(m.n + ":" + m.Count);
   }

   s = "Geben Sie die Prozesse aus, die mehr als 10 Threads haben!";
   Ergebnis =
      (from n in Prozesse
       where n.Threads.Count > 10
```

```
      select n);
   Print(Ergebnis, s);

   s = "Geben Sie den/die Prozess(e) aus, der/die die meisten Threads hat!";
   Ergebnis = (from n in Prozesse where n.Threads.Count == Prozesse.Max(x =>
x.Threads.Count) select n);
   Print(Ergebnis, s);
  }
```

46.3.3 LINQ to Objects mit eigenen Geschäftsobjekten

LINQ-Abfragen können auch über eigene (Geschäfts-)Objektmengen gestellt werden, egal ob diese direkt durch Implementierung von IEnumerable/IEnumerable<T> oder durch Ableiten von einer der vordefinierten Mengenklassen implementiert wurden. Das folgende Objektmodell zeigt drei Mengen (FlugMenge, PassagierMenge und BuchungsMenge), die jeweils durch Ableiten von der Klasse System.Collections.Generic.List<T> realisiert wurden.

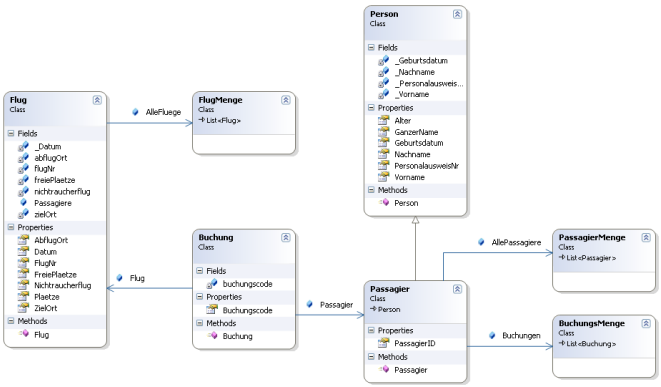


Abbildung: Objektmodell für die folgenden Beispiele

Beispiel

Das folgende Listing zeigt zahlreiche Beispiele zur Abfrage der Mengen in dem oben dargestellten Objektmodell. Das Listing setzt voraus, dass die Mengen vorher mit Daten gefüllt wurden. Diese Befüllung wird hier aus Platzgründen nicht abgedruckt, ist jedoch in den Codebeispielen zu diesem Buch enthalten.

Listing: Anwendungsbeispiele von LINQ to Objects auf verschiedene selbstdefinierte Geschäftsobjektmengen

```
private static void Demo_LTO_Objektmodell()
  {
   // Initialisiere das Objektmodell
   BO_Init.Init();
   string s;
```

```
   long i;
   Flug flug;
   double d;

   s = "Geben Sie alle Flüge von Rom abgehend aus!";
   var Ergebnis =
    from f in Flug.AlleFluege
    where f.AbflugOrt == "Rom"
    select f;
   Print(Ergebnis, s);

   s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben.";
   Ergebnis =
      from n in Flug.AlleFluege
      where n.FreiePlaetze < 100
      select n;
   Print(Ergebnis, s);

   s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben.
Sortieren Sie die Liste aufsteigend nach Platzanzahl.";
   Ergebnis =
   from n in Flug.AlleFluege
   where n.FreiePlaetze < 100
   orderby n.FreiePlaetze
   select n;
   Print(Ergebnis, s);

   s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben.
Sortieren Sie die Liste absteigend nach Platzanzahl.";
   Ergebnis =
   from n in Flug.AlleFluege
   where n.FreiePlaetze < 100
   orderby n.FreiePlaetze descending
   select n;
   Print(Ergebnis, s);

   s = "Geben Sie Flug 101 aus.";
   flug = (from f in Flug.AlleFluege
           where f.FlugNr == 101
           select f).SingleOrDefault();
   Print(flug, s);

   s = "Geben Sie die Flüge aus, aber jede Strecke nur einmal!";
   var Strecken =
        (from n in Flug.AlleFluege
         select new { n.AbflugOrt, n.ZielOrt }).Distinct();
   HeadLine(s);

   foreach (var f in Strecken)
   {
    Console.WriteLine(f.AbflugOrt + " -> " + f.ZielOrt);
   }

   s = "Geben Sie alle Ziele aus, die von Rom aus erreichbar sind.";
   var Ziele =
        (from n in Flug.AlleFluege
         where n.AbflugOrt == "Rom"
         select n.ZielOrt).Distinct();
   HeadLine(s);

   foreach (string f in Ziele)
   {
    Console.WriteLine(f);
   }
```

```
   s = "Geben Sie den vierten bis achten Flug aus in der nach freien Plätzen
aufsteigend sortierten Liste aller Flüge, die in Berlin landen.";
   Ergebnis =
       (from n in Flug.AlleFluege
        where n.ZielOrt == "Berlin"
        orderby n.FreiePlaetze
        select n).Skip(3).Take(4);
   Print(Ergebnis, s);

   s = "Geben Sie den ersten Flug aus in der nach freien Plätzen aufsteigend
sortierten Liste aller Flüge, die in Berlin landen.";
   flug =
      (from n in Flug.AlleFluege
       where n.ZielOrt == "Berlin"
       orderby n.FreiePlaetze
       select n).First();
   Print(flug, s);

   s = "Geben Sie den letzten Flug aus in der nach freien Plätzen aufsteigend
sortierten Liste aller Flüge, die in Berlin landen.";
   flug =
      (from n in Flug.AlleFluege
       where n.ZielOrt == "Berlin"
       orderby n.FreiePlaetze
       select n).Last();
   Print(flug, s);

   s = "Geben Sie den 10. Flug aus in der nach freien Plätzen aufsteigend
sortierten Liste aller Flüge, die in Berlin landen.";
   flug =
      (from n in Flug.AlleFluege
       where n.ZielOrt == "Berlin"
       orderby n.FreiePlaetze
       select n).ElementAt(9);
   Print(flug, s);

   s = "Geben Sie den 150. Flug aus in der nach freien Plätzen aufsteigend
sortierten Liste aller Flüge, die in Berlin landen.";
   flug =
      (from n in Flug.AlleFluege
       where n.ZielOrt == "Berlin"
       orderby n.FreiePlaetze
       select n).ElementAtOrDefault(149);
   Print(flug, s);

   s = "Geben Sie Anzahl der Flüge aus (mit einem LINQ-Statement!)";
   i =
      (from n in Flug.AlleFluege
       select n).Count();
   Print(i, s);

   s = "Geben Sie die geringste freie Platzanzahl aus.";
   i =
      (from n in Flug.AlleFluege
       select n).Min(n => n.FreiePlaetze);
   Print(i, s);

   s = "Geben Sie die höchste freie Platzanzahl aus.";
   i =
      (from n in Flug.AlleFluege
       select n).Min(n => n.FreiePlaetze);
   Print(i, s);
```

```
   s = "Geben Sie die durchschnittliche freie Platzanzahl aus.";
   d =
      (from n in Flug.AlleFluege
       select n).Average(n => n.FreiePlaetze);
   Print(d, s);

   s = "Geben Sie Summe aller freien Plätze aus.";
   i =
      (from n in Flug.AlleFluege
       select n).Sum(n => n.FreiePlaetze);
   Print(i, s);

   s = "Gruppieren Sie die Flüge nach Abflugorten.";
   IEnumerable<IGrouping<string, Flug>> GruppeErgebnis =
       (from n in Flug.AlleFluege
        group n by n.AbflugOrt);
   Print(GruppeErgebnis, s);

   s = "Geben Sie die Häufigkeit eines jeden Abflugortes aus!";
   IDictionary<string, int> GruppeHaeufigkeit =
         (from n in Flug.AlleFluege
          group n by n.AbflugOrt into g
          select new { Name = g.Key, AnzFlug = g.Count() }
          ).ToDictionary(y => y.Name, y => y.AnzFlug);
   Print(GruppeHaeufigkeit, s);

   s = "Erstellen Sie eine gruppierte Liste aller Passagiere mit ihren
Buchungen!";
   var pass2 = from p in Passagier.AllePassagiere
               orderby p.GanzerName
               select new { p.GanzerName, p.Buchungen };
   foreach (var p in pass2)
   {
    Console.WriteLine(p.GanzerName);
    foreach (Buchung b in p.Buchungen)
     Console.WriteLine("\t" + b.Buchungscode);
   }

   s = "Erstellen Sie die Liste der zehn Passagiere mit den meisten Buchungen.";
   var pass = (from p in Passagier.AllePassagiere
               orderby p.Buchungen.Count descending
               select p).Take(10);
   Print(pass, s);

   s = "Erstellen Sie die Liste des/der Passagier(e) mit den meisten Buchungen.";
   pass = (from n in Passagier.AllePassagiere where n.Buchungen.Count ==
Passagier.AllePassagiere.Max(x => x.Buchungen.Count) select n);
   Print(pass, s);

   s = "Finden Sie alle Passagiere, die nach Rom fliegen.";
   pass = (from p in Passagier.AllePassagiere
           where p.Buchungen.Any(b => b.Flug.ZielOrt == "Rom")
           select p);
   Print(pass, s);

   s = "Finden Sie alle Passagiere, die genauso viele Buchungen haben wie ein
Flug freie Plätze.";
   var joinpass = (from p in Passagier.AllePassagiere
                   join f in Flug.AlleFluege
                   on p.Buchungen.Count equals f.FreiePlaetze
                   select new { p.GanzerName, f.FlugNr, p.Buchungen.Count,
f.FreiePlaetze });
   HeadLine(s);
   foreach (var j in joinpass)
```

```
   {
    Console.WriteLine(j.GanzerName + " und Flug " + j.FlugNr + " haben die
gleiche Zahl: " + j.Count + " / " + j.FreiePlaetze);
   }

   s = "Geben Sie alle Passagiere aus und optional dazu einen Flug, der
genausoviele freie Plätze hat wie der Passagier Buchungen hat.";
   var joinpass2 = (from p in Passagier.AllePassagiere
                    join f in Flug.AlleFluege
                    on p.Buchungen.Count equals f.FreiePlaetze
                    into Fluege
                    select new { p.GanzerName, Fluege });
   HeadLine(s);
   foreach (var j in joinpass2)
   {
    Console.WriteLine(j.GanzerName + " hat " + j.Fluege.Count() + "
korrespondierende Flüge!");
   }

   s = "Geben Sie alle Passagiere aus, die älter als 50 Jahre sind!";
   pass = (from p in Passagier.AllePassagiere
           where p.Geburtsdatum.AddYears(50) < DateTime.Now
           select p);
   Print(pass, s);

   s = "Geben Sie alle Flüge aus, mit Passagieren älter als 50 Jahre !";
   Ergebnis = (from p in Passagier.AllePassagiere
               where p.Geburtsdatum.AddYears(50) < DateTime.Now
               from b in p.Buchungen
               select b.Flug).Distinct();
   Print(Ergebnis, s);
 }
```

46.4 Parallel LINQ (PLINQ)

Parallel LINQ (PLINQ, früher auch LINQ to Parallel) ist neu ab .NET 4.0. Es ermöglicht die Parallelisierung von LINQ to Objects-Abfragen auf mehrere Prozessoren / Prozessorkerne. Dadurch kann (!) sich eine Beschleunigung ergeben.

PLINQ ist realisiert in Form der Erweiterungsmethode AsParallel(), die auf einfache Weise in LINQ to Objects-Abfragen integriert werden kann.

Das folgende Beispiel zeigt eine einfache Abfrage mit Filtern (where) und Sortieren (orderby) über eine Zahlenreihe mit Einsatz von AsParallel().

Listing: Eine Abfrage ohne und mit PLINQ

```
  /// <summary>
  /// Massendaten filtern und sortieren mit PLINQ
  /// </summary>
  public static void LTOMassendaten_mit_PLINQ()
  {

   long AnzZahlen = 1000000;
   System.Random rnd = new Random(DateTime.Now.Year);
   List<long> Zahlen = new List<long>();
   for (int i = 1; i <= AnzZahlen; i++) Zahlen.Add(rnd.Next(100));

   long Summe = 0;
   Stopwatch t = new Stopwatch();
   t.Start();
   for (int w = 1; w <= 20; w++)
   {
```

```
    var q = (from x in Zahlen.AsParallel() where x < 50 orderby x select x).ToLis
t();
    Summe += q.Count();
   }
   t.Stop();
   Console.WriteLine("Summe: " + Summe);
   Console.WriteLine("Mit PLINQ  = " + t.ElapsedMilliseconds);

  }
```

Die folgende Tabelle zeigt Messergebnisse, auch im Vergleich, wenn man AsParallel() weglassen würde.

Anzahl Zahlen	Ohne PLINQ – ohne AsParallel()	Mit PLINQ – mit AsParallel()
10000	50 Millisekunden	76 Millisekunden
100000	441 Millisekunden	190 Millisekunden
1000000	5132 Millisekunden	1532 Millisekunden

Tabelle: Ausführungsdauer von LINQ to Objects ohne und mit PLINQ, jeweils auf dem gleichen Rechner mit Intel Core I7 mit acht Prozessorkernen

Achtung: Man sieht: Erst bei größeren Grundmengen lohnt der mit der Parallelisierung verbundene Zusatzaufwand!

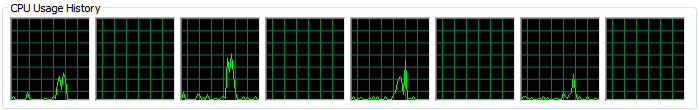


Abbildung: Auslastung von acht Kernen bei einer Abfrage ohne PLINQ

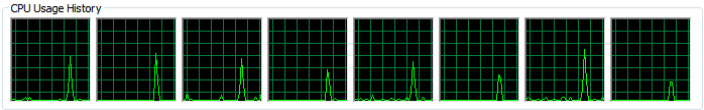


Abbildung: Auslastung von acht Kernen bei einer Abfrage mit PLINQ

Achtung: PLINQ bessert auch Reihenfolgefehler aus. Dort ist

```
from x in Zahlen orderby x where x < 50 select x
```

genauso schnell wie

```
from x in Zahlen where x < 50 orderby x  select x
```

Ohne PLINQ dauert die erste LINQ to Objects-Abfrage bei 10000 Zahlen etwa doppelt so lange wie die zweite!

Tipp: Bei Bedarf kann das Verhalten von PLINQ durch den Einsatz weiterer Erweiterungsmethoden beeinflusst werden. Wird zum Beispiel mit AsOrdered() festgelegt, dass die Sortierreihenfolge aus der Quelle erhalten bleiben soll, bringt dies im Zuge einer parallelen Abfrage etwas Mehraufwand mit sich und muss deswegen mit dieser Methode bei Bedarf angefordert werden. Mittels WithCancellation() wird darüber hinaus ein CancellationToken an die Abfrage übergeben, sodass deren Ausführung später abgebrochen werden kann.

WithDegreeOfParallelism() gibt an, wie viele Tasks maximal für diese Anfrage verwendet werden dürfen. Standardmäßig werden so viele Tasks wie Kerne verwendet, die dann im Idealfall alle genutzt werden können. Kommt PLINQ zur Entscheidung, dass das Parallelisieren einer Abfrage nicht sinnvoll ist, so wird diese sequenziell ausgeführt. Dieses Verhalten kann allerdings mittels WithExecutionMode() beeinflusst werden. Im betrachteten Listing wird damit beispielsweise eine Parallelisierung erzwungen. Die letzte der verwendeten Optionen, WithMergeOptions(), legt fest, wie die Ergebnisse der unterschiedlichen Tasks kombiniert werden sollen. Mit FullyBuffered wird zum Beispiel erreicht, dass jeder Task sämtliche Ergebnisse in einen eigenen Buffer ablegt, wobei diese erst zum Schluss zur Ergebnismenge zusammengefügt werden.

Lesen Sie unbedingt »When To Use Parallel.ForEach and When to Use PLINQ?« [*download.microsoft.com/download/B/C/F/BCFD4868-1354-45E3-B71B-B851CD78733D/WhenToUseParallelForEachOrPLINQ.pdf*].

47 Source-Generatoren

Eine weitere größere Neuerung seit C# 9.0 sind Source Generators (anfangs auch Source Code Generators genannt), mit denen ein Entwickler zusätzlichen Programmcode zur Kompilierungszeit erzeugen kann, der zusammen mit dem eigentlichen Programmcode kompiliert wird (siehe Abbildung). Damit kann man z.B. Annotationen eine Bedeutung geben im Sinne aspektorientierter Programmierung (AOP). Für Microsoft sollen die neuen Generatoren den Weg zu einem allgemeinen Ahead-of-Time-Compiler ebnen, der in sowohl in .NET 5.0 als auch .NET 6.0 noch fehlt.

Hinweis: Ein Source Generator kann zusätzlichen Programmcode erzeugen, nicht aber wie Werkzeuge zum "IL Enhancement" (z.B. PostSharp) bestehenden Programmcode verändern.

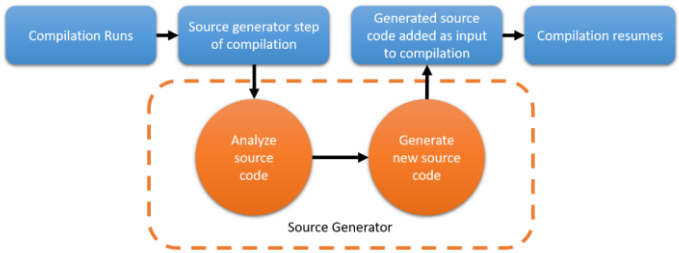


Abbildung: Funktion eines Source Code Generators (Quelle: Microsoft)

47.1 Aufbau eines Source-Generators

Ein Source-Generator ist eine .NET-Klasse, die die Schnittstelle Microsoft.CodeAnalysis.ISourceGenerator mit diesen beiden Methoden realisiert:

- void Initialize(GeneratorInitializationContext context)
- void Execute(GeneratorExecutionContext context)

Das nächste Listing zeigt einen Generator, der eine Klasse HelloWorld erzeugt.

Listing: Ein ganz einfacher Source-Generator [CSharpSourceCodeGenerators/HelloWorldGenerator.cs]

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Text;

namespace SourceGeneratorSamples
{

 [Generator]
 public class HelloWorldGenerator : ISourceGenerator
 {
```

```
  public void Execute(GeneratorExecutionContext context)
  {
   var source = @"
using System;
namespace HelloWorldGenerated
{
    public static class HelloWorld
    {
        public static void SayHello()
        {
            Console.WriteLine(""Hallo aus der Assembly "" + System.Reflection.Ass
embly.GetExecutingAssembly().GetName().Name);
        }
    }
}";

   // Code wird injiziert
   context.AddSource("helloWorldGenerator", SourceText.From(source, Encoding.UTF8
));
  }

  /// <param name="context"></param>
  public void Initialize(GeneratorInitializationContext context)
  {
  }
 }
}
```

Diese Generator-Klasse muss in eine DLL-Assembly kompiliert werden und benötigt Verweise auf die NuGet-Pakete "Microsoft.CodeAnalysis.Csharp" und "Microsoft.CodeAnalysis.Analyzers".

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

 <PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <LangVersion>13.0</LangVersion>
 </PropertyGroup>

 <ItemGroup>
  <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="3.8.0-
3.final" PrivateAssets="all" />
  <PackageReference Include="Microsoft.CodeAnalysis.Analyzers" Version="3.0.0" Pr
ivateAssets="all" />
 </ItemGroup>

</Project>
```

Diese DLL kann der Entwickler dann in einem anderen Projekt wie einen Rosyln-Analyzer einbinden. Notwendig bei der <ProjectReference> sind die die Zusatzattribute OutputItemType="Analyzer" ReferenceOutputAssembly="false".

```
<Project Sdk="Microsoft.NET.Sdk">
  …
  <ItemGroup>
   <ProjectReference
```

```
    Include="..\CSharpSourceCodeGenerators\CSharpSourceCodeGenerators.csproj"
    OutputItemType="Analyzer"
    ReferenceOutputAssembly="false" />
  </ItemGroup>
 …
</Project>
```

Man sieht den generierten Quellcode dann in Visual Studio im Ast "Dependencies/Analyzers" unter dem Namen der Assembly. Der Quellcode wird also nicht in dem Projekt, der den Generator implementiert, sondern in dem nutzenden Projekt erzeugt.

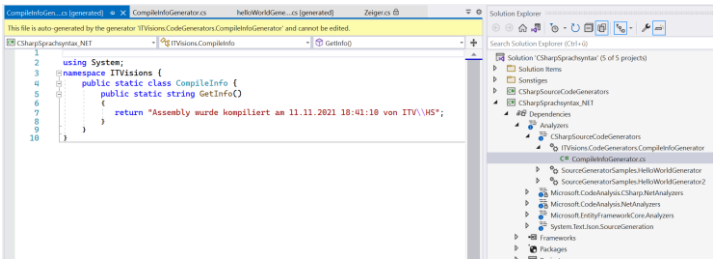


Abbildung: Generierter Quellcode von einem Source-Generator im nutzenden Projekt

Danach steht innerhalb dieses Projekts die Klasse HelloWorld zur Verfügung:

```
HelloWorldGenerated.HelloWorld.SayHello();
```

Mit der Taste F12 ("Go to Definition") kann man den generierten Code direkt anspringen.

Verweis: Da dies ein sehr umfangreiches Thema ist, sei hier auf die Einträge "Introducing C# Source Generators" [*devblogs.microsoft.com/dotnet/introducing-c-source-generators*] und "New C# Source Generator Samples" [*devblogs.microsoft.com/dotnet/new-c-source-generator-samples*] im .NET-Blog sowie die Dokumentation auf GitHub [*github.com/dotnet/roslyn/blob/master/docs/features/source-generators.md*] verwiesen.

47.2 Praxisbeispiel

Ein Praxisbeispiel für den Einsatz eines Source Code-Generators ist die Generierung einer C#-Funktion, die Informationen liefert, zu welchem Zeitpunkt und von wem eine Assembly übersetzt wurde. Dies müsste der Entwickler normalerweise jeweils händisch im Code hinterlegen oder man müsste über einen Pre-Build-Schritt den Quellcode modifizieren. Hier hilft ein Source Code-Generator, der jeweils beim Übersetzen eine entsprechende Funktion erzeugt.

Listing: [CSharpSourceCodeGenerators/CompileInfoGenerator.cs]

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Text;
using System;
using System.Text;

namespace ITVisions.CodeGenerators
{
 [Generator]
 public class CompileInfoGenerator : ISourceGenerator
```

```
 {

  public void Execute(GeneratorExecutionContext context)
  {
   System.Diagnostics.Trace.WriteLine("=== HelloWorldGenerator.Execute");

   // Source wird erzeugt
   var source = @"
using System;
namespace ITVisions {
    public static class CompileInfo {
        public static string GetInfo()
        {
           return ""Assembly wurde kompiliert am [NOW] von [USER]"";
        }
    }
}";
   source = source.Replace("[NOW]", DateTime.Now.ToString());
   // hier vier Backslash notwendig, da im generierten Code \\ stehen muss!
   source = source.Replace("[USER]", System.Environment.UserDomainName + "\\\\" +
System.Environment.UserName);

   // neuer Code wird injeziert
   context.AddSource("CompileInfoGenerator", SourceText.From(source,
Encoding.UTF8));
  }

  /// <summary>
  /// In diesem Fall ohne Funktion
  /// </summary>
  public void Initialize(GeneratorInitializationContext context)
  {
   System.Diagnostics.Trace.WriteLine("=== HelloWorldGeneratorInitialize");
  }
 }
}
```

Nach der Einbindung der generierten Assembly als <ProjectReference> kann man im Quellcode den generierten Code verwenden, z.B.

```
Console.WriteLine(ITVisions.CompileInfo.GetInfo());
```

48 Performanceoptimierungen

Dieses Kapitel erörtert Themen zur Leistungssteigerung von C#-basierten Anwendungen. Dabei geht es hier – wie im gesamtem Buch – ausschließlich um Leistungssteigerungen auf Ebene des Compilers. Leistungstipps in Bezug auf die Verwendungen von .NET-Klassen sind nicht Thema des Buchs.

48.1 x64 versus x86

Grundsätzlich sollte man .NET-Anwendungen immer im Modus "AnyCPU" kompilieren. Damit laufen diese sowohl auf 32-Bit-Betriebssystemen als auch auf 64-Bit-Betriebssystemen jeweils in der entsprechenden Bit-Anzahl.

Wenn man eine .NET-Anwendung im "x86"-Modus kompiliert, läuft sie auch auf einem 64-Bit-Betriebssystem, dort aber im Emulator (unter Windows heißt dieser Windows on Windows 64, kurz WOW64). Die Performanz ist dann schlechter als wenn diese als 64-Bit-Anwendung laufen würde.

Gründe für die bessere Performance von 64-Bit-Anwendungen sind:

- Durch die längeren CPU-Register kann die Verarbeitung von Zahlen schneller erfolgen (32-Bit-Anwendungen brauchen 2 Register, wo 64-Bit-Anwendungen mit einem auskommen)
- Die 64-Bit-.NET-CLR ist optimiert gegenüber der 32-Bit-.NET-CLR.
- Der Emulator entfällt.

Hinweis: Eine im "x64"-Modus kompilierte Anwendung läuft nur auf 64-Bit-Betriebssystemen.

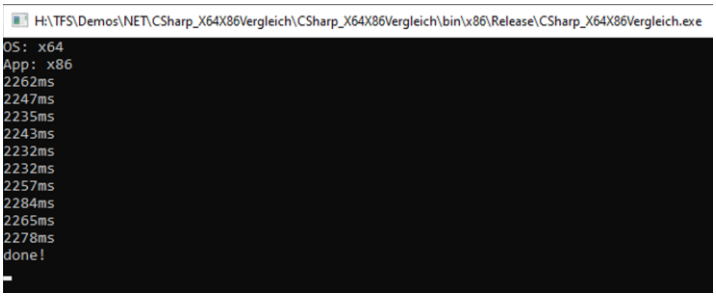


Abbildung: Eine Anwendung im x86-Modus auf einem 64-Bit-Windows

Abbildung: Die gleiche Anwendung im x64-Modus auf einem 64-Bit-Windows ist schneller

48.2 Debug versus Release

Eine .NET-Anwendung, die im Release-Modus kompiliert wurde, läuft schneller als nach dem Kompilieren im Debug-Modus. Zur Betriebszeit sollte eine Anwendung daher als "Release" kompiliert werden.

Abbildung: Eine Anwendung im Debug-Modus kompiliert

Abbildung: Die gleiche Anwendung im Release-Modus kompiliert ist schneller

48.3 Vermeidung von Laufzeitfehlern (Exceptions)

Vermeiden Sie das Auslösen und Abfangen von Laufzeitfehlern (Exceptions), wo immer möglich, durch aktive Prüfung und Fallunterscheidungen. Das Auslösen von Exceptions ist in der .NET-Laufzeitumgebung eine recht "teure" Operation.

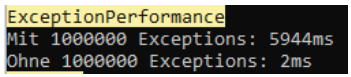


Abbildung: Verlangssamung durch Exceptions

Listing: Programmcode zu obiger Ausgabe

```
using ITVisions;
using System;
using System.Diagnostics;

namespace CSharpSyntaxNET5
{
 public class Performance
 {
  public void ExceptionPerformance()
```

```
  {
   CUI.H1(nameof(ExceptionPerformance));

   int anz = 1000000;
   int x = 1;
   int y = 0;

   #region Abfangen einer Exception
   var sw1 = new Stopwatch();
   sw1.Start();
   for (int i = 0; i < anz; i++)
   {
    int z;
    try
    {
     z = x / y;
    }
    catch (Exception)
    {
     z = Int32.MinValue;
    }
   }
   sw1.Stop();
   Console.WriteLine("Mit " + anz + " Exception: " + sw1.ElapsedMilliseconds +
"ms");
   #endregion

   #region Vermeidung einer Exception
   var sw2 = new Stopwatch();
   sw2.Start();
   for (int i = 0; i < anz; i++)
   {
    int z;

    if (y == 0) z = Int32.MinValue;
    else z = x / y;
   }
   sw2.Stop();
   Console.WriteLine("Ohne " + anz + " Exception: " + sw2.ElapsedMilliseconds +
"ms");
   #endregion
  }
 }
}
```

48.4 Ahead-of-Timer-Compiler (Native AOT)

Schon seit dem Jahr 2016 arbeitet Microsoft als Alternative zu dem Just-in-Time-Compiler (JIT) an einem Ahead-of-Timer-Compiler (AOT), der direkt Machinencode erzeugt und damit .NET-Anwendungen schneller starten lässt. Dieses Projekt lief zunächst als "CoreRT" [*github.com/dotnet/corert*] und sollte schon in .NET 5.0, dann später in .NET 6.0 erscheinen. Stattdessen aber wurden die Arbeiten in das Projekt "Native AOT"

[*github.com/dotnet/runtimelab/tree/feature/NativeAOT*] überführt und .NET 6.0 lieferte AOT lediglich für Blazor WebAssembly.

In .NET 7.0 hat Microsoft den Native AOT-Code in das offizielle GitHub-Repository "dotnet/runtime" [*https://github.com/dotnet/runtime*] überführt und bot eine erste Version von Native AOT in .NET 7.0. In .NET 8.0 gab es eine Erweiterung.

Native AOT verkleinert das Deployment-Paket erheblich (es findet ein Application Trimming, alias Tree Shaking, statt) und beschleunigt den Anwendungsstart.

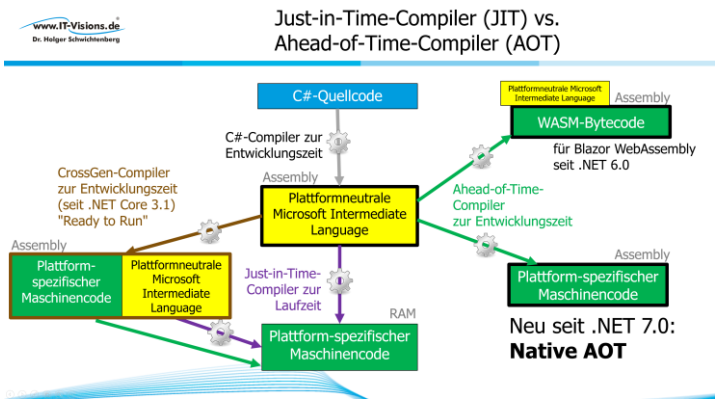


Abbildung: Der Native AOT-Compiler im Vergleich zu anderen Compile-Verfahren in .NET

48.4.1 Native AOT in .NET 7.0

Die erste Version von Native AOT war sehr eingeschränkt, denn sie funktioniert nur für Konsolenanwendungen und auch dort gibt es Einschränkungen. Nicht möglich mit dem AOT-Compiler sind:

- Funktionen, die auf Laufzeitcodegenerierung basieren (Reflection Emit)
- Dynamisches Nachladen von Assemblies (Add-Ins/Plug-Ins)
- Funktioniert nicht mit C++/CLI
- Die Nutzung von Komponenten des Component Object Models (COM), z.B. zur Steuerung von Microsoft Office, ist nicht möglich
- Auch die Programmierschnittstellen der Windows Runtime Library (WinRT) sind nicht nutzbar

Den AOT-Compiler aktiviert man durch eine Einstellung in der Projektdatei:

```
<!--für AOT -->
<PropertyGroup>
 <PublishAot>true</PublishAot>
</PropertyGroup>
```

Hinweis: Seit .NET 8.0 kann man diese Einstellung direkt beim Anlegen eines Projekts vorsehen durch das Häkchen "Enable native AOT publish" bzw. an der Kommandozeile via Option --aot, z.B. dotnet new console –aot

In .NET 8.0 ist die AOT-Kompilierung möglich für folgende Projekttypen: console, api, grpc und worker.

Die Einstellung wirkt aber nur, wenn man die Anwendung veröffentlicht, also z.B. per Kommandozeile

```
dotnet publish -r win-x64 -c Release
```

Dann sieht man den Kompilierungsvorgang, der ab der Ausgabe "Generating native code" einige Zeit braucht. Das Ergebnis eine einzige ausführbare Datei im Projektunterordner /bin/Release/[DOTNETVERSION]/win-x64/native. Das Ergebnis ist ein "Self-Contained Executable", läuft also ohne dass vorher eine .NET Runtime installiert werden musste. Die notwendigen Teile der Runtime sind schon enthalten! So gesehen ist diese Datei sehr klein!

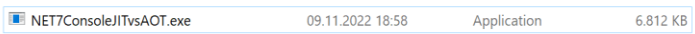


Abbildung: Ergebnis der AOT-Kompilierung

Die folgenden Abbildungen zeigt eine Konsolenanwendung, die mit JIT und AOT kompiliert wurden.

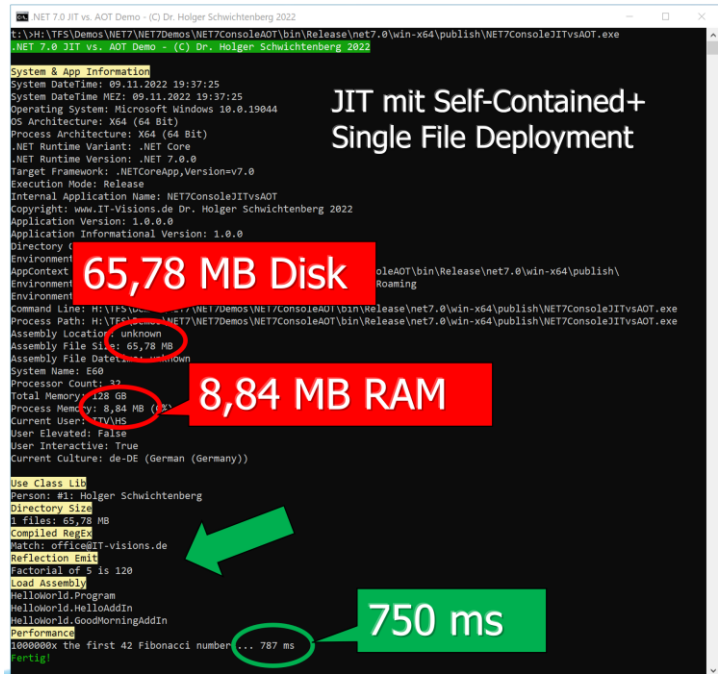


Abbildung: JIT-Kompilierte Konsolenanwendungen

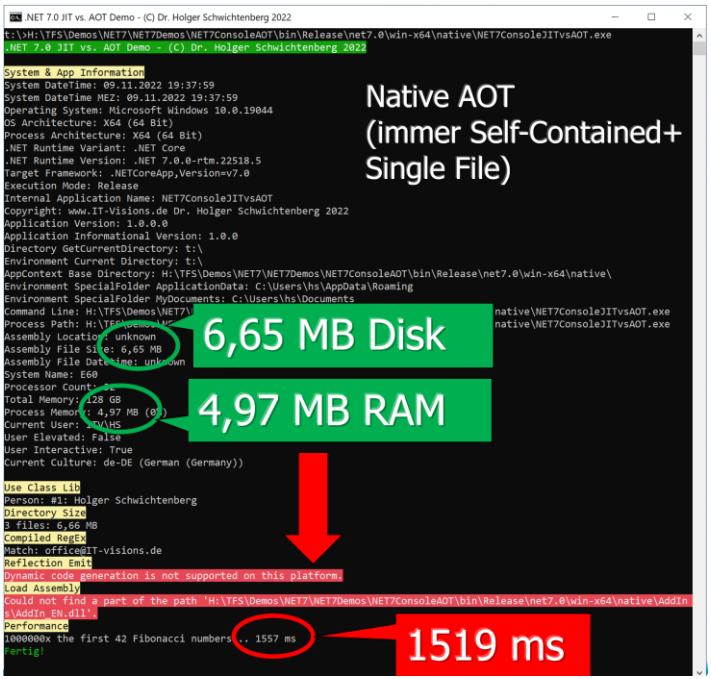


Abbildung: AOT-kompilierte Konsolenanwendungen

Die wesentlichen Erkenntnisse aus den beiden Bildern sind:

- Wie oben erwähnt, funktioniert die Laufzeitcodegenerierung mit Reflection Emit und das dynamische Nachladen von Assemblies nicht bei AOT.
- Das Deploymentpaket bei AOT ist sehr wesentlich kleiner: 6,65 MB vs. 65,78 MB.
- Entsprechend ist der RAM-Bedarf bei Anwendungsstart bei AOT geringer: 4,97 MB vs. 8,84 MB.
- Aber: die Rechenzeit für eine Zahlenreihe von 42 Millionen zahlen dauert bei Native-AOT doppelt so lange: 787 ms vs. 1519 ms.

Die offizielle Dokumentationsseite zu Native AOT verschweigt [*https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/*], dass es Einstellungen für den AOT-Compiler gibt. Erste beim Wühlen in GitHub findet man eine weitere Dokumentationsseite [*https://github.com/dotnet/runtime/blob/main/src/coreclr/nativeaot/docs/optimizing.md*] mit der

Zusatzoption <IlcOptimizationPreference>Speed</IlcOptimizationPreference>. Damit kommt ein AOT-Kompilat heraus, dass nur wenige Kilobyte größer ist (6.84 MB), aber die Berechnung mit 612 Millisekunden noch schneller als der Just-in-Time-Compiler ausführt!

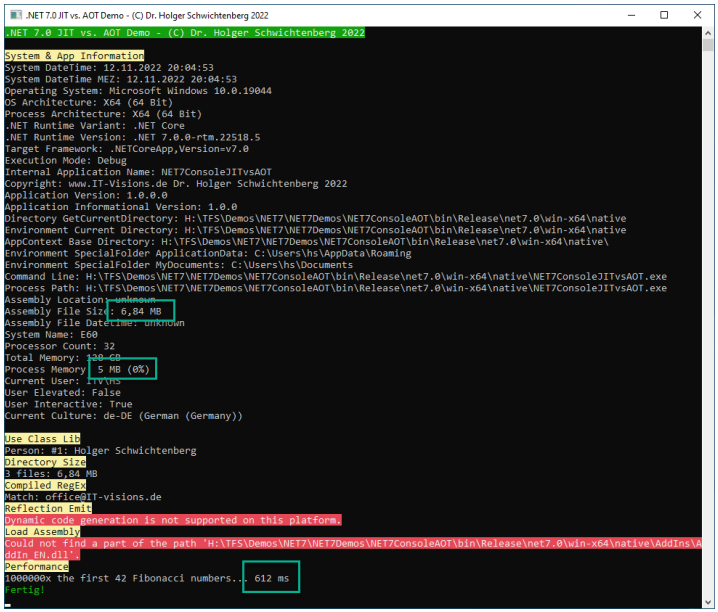


Abbildung: AOT-kompilierte Konsolenanwendungen mit Optimierung auf Leistung

Zu Windows Forms mit Native AOT gibt es von Microsoft Stand .NET 9.0 diese Aussagen:

- "you could run a Windows Forms application under native AOT"
- "This work is still highly experimental, and some scenarios are rough and require manual work."

Quelle: *https://devblogs.microsoft.com/dotnet/winforms-enhancements-in-dotnet-7*

48.4.2 Native AOT in .NET 8.0

Der "Native AOT" genannte Compiler konnte in .NET 7.0 nur Konsolenanwendungen übersetzen.

Seit .NET 8.0 sind nun zusätzlich auch folgende Anwendungsarten beim AOT-Compiler möglich:

- Hintergrunddienste (Worker Services)
- gRPC-Dienste

- WebAPIs mit Einschränkungen: Bei den WebAPIs ist lediglich das "Minimal WebAPI" genannte Modell möglich mit JSON-Serialisierung via System.Text.Json im Source Generator-Modus. Weitere Einschränkungen siehe folgende Abbildung.

Hinweis: Native AOT funktioniert also weiterhin dort nicht, wo es am nötigsten wäre, die Startzeit und den RAM-Bedarf zu verringern: Windows Forms und WPF.

Den Source Generator in System.Text.Json hat Microsoft dazu ausgebaut, sodass er nun fast alle Konfigurationsoptionen wie der Reflection-basierte Modus kennt. Zudem funktioniert der Source Generator jetzt zusammen mit den Init-Only-Properties aus C# 9.0 und den Required Properties aus C# 11.0. Den alten Reflection-Modus kann man durch eine Projekteinstellung komplett deaktivieren. Den Modus prüfen Entwicklerinnen und Entwickler mit der Bedingung if (JsonSerializer.IsReflectionEnabledByDefault) { … }.

Feature	Fully Supported	Partially Supported	Not Supported
gRPC	✅Fully supported		
Minimal APIs		✅Partially supported	
MVC			❌Not supported
Blazor			❌Not supported
SignalR			❌Not supported
Authentication			❌Not supported (JWT soon)
CORS	✅Fully supported		
Health checks	✅Fully supported		
Http logging	✅Fully supported		
Localization	✅Fully supported		
Output caching	✅Fully supported		
Rate limiting	✅Fully supported		
Request decompression	✅Fully supported		
Response caching	✅Fully supported		
Response compression	✅Fully supported		
Rewrite	✅Fully supported		
Session			❌Not supported
SPA			❌Not supported
Static files	✅Fully supported		
WebSockets	✅Fully supported		

Abbildung: Unterstützte ASP.NET Core-Features in Native AOT 8.0 (Bildquelle: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/native-aot?view=aspnetcore-8.0&tabs=netcore-cli)

48.4.3 Native AOT in .NET 9.0

In .NET 9.0 sind folgende Erweiterungen des AOT-Compilers enthalten:

- ASP.NET Core SignalR-Hubs können mit Native AOT kompiliert werden
- .NET für iOS / macOS: Trimming bei Native AOT
- Swashbuckle.AspNetCore (für OAS) funktioniert mit NativeAOT seit Version 6.6
- AOT für WinUI 3-Oberflächen seit Windows-App-SDK 1.6

  Das Entwicklungsteam von WinUI hat auf der Microsoft BUILD-Konferenz 2024 in einem Vortrag *https://build.microsoft.com/en-US/sessions/11626139-a9d0-4f8c-b664-*

3f3436cea50a angekündigt, dass man ab Version 1.6 des Windows App SDK, die im September 2024 erscheinen soll, die Native AOT-Kompilierung von WinUI 3-XAML und Code-Behind-Code für WinUI 3-Anwendungen erlauben wird. Zunächst wird man den Native AOT-Compiler aus .NET 8.0 unterstützen, später dann .NET 9.0. Die WinUI 3-Bibliotheken selbst sind ja bereits Native Code, müssen also nicht erneut übersetzt werden. Das Team verspricht 45% kleinere Deployment-Pakete und einen um die Hälfte der Zeit beschleunigten Anwendungsstart. Eine gut funktionierende AOT-Kompilierung mit den genannten Vorteilen wäre ein wesentliches Argument für den Einsatz von WinUI 3 statt WPF.

- Entity Framework Core soll leider nur experimentell in .NET 9.0 laufen auf Basis von statischer Codeanalyse und C#-Interceptoren. Es wird nur für statische LINQ-Abfragen funktionieren.

Hinweis: Native AOT funktioniert also weiterhin dort nicht, wo es am nötigsten wäre, die Startzeit und den RAM-Bedarf zu verringern: Windows Forms und WPF.

48.4.4 Neue Native AOT-Option in Projektvorlagen

Neu seit .NET 8.0 ist, dass es bei einigen Projektvorlagen nun direkt eine Möglichkeit gibt, den AOT-Compiler zu aktivieren mit der Kommandozeilenoption --aot bzw. mit einem Häkchen in Visual Studio:

- Konsolenanwendung
  ```
  dotnet new console --aot
  ```
- Worker Service
  ```
  dotnet new worker --aot
  ```
- gRPC
  ```
  dotnet new grpc --aot
  ```

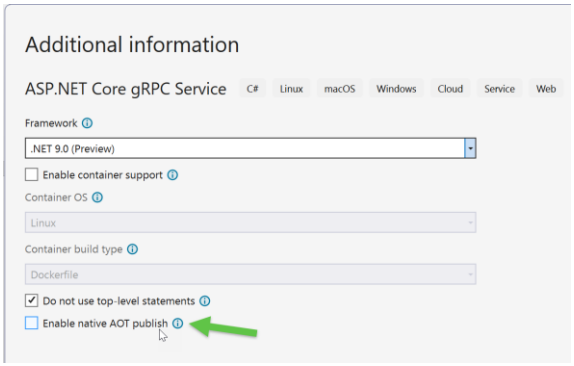


Abbildung: Native AOT-Option bei der Projektvorlage für gRPC-Dienste in Visual Studio

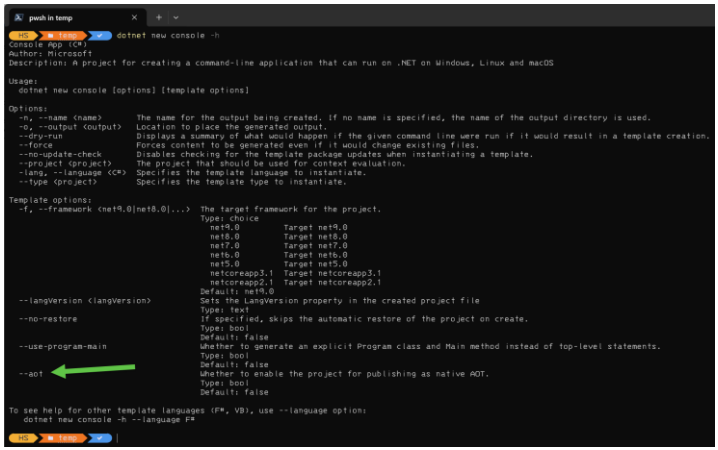


Abbildung: Option --aot bei dotnet new

Bei der Projektvorlage für ASP.NET Core WebAPIs (Kurzname: "webapi") gibt es keine Option --aot und kein Häkchen in Visual Studio. Hier hat sich Microsoft entschlossen, eine eigene Projektvorlage zu bauen "ASP.NET Core WebAPI (native AOT)" mit Kurznamen "webapiaot". Diese verwendet auch nicht das bisher in der WebAPI-Projektvorlage übliche Beispiel von Wetterdaten, sondern eine Aufgabenliste.

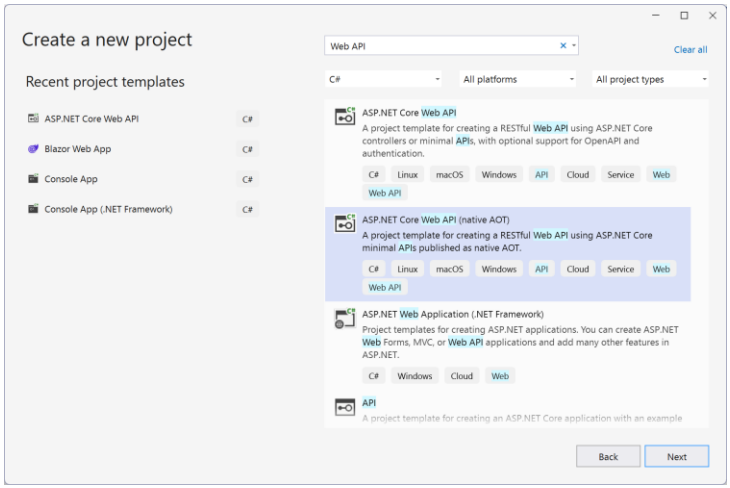


Abbildung: WebAPI-Projektvorlagen in Visual Studio

48.4.5 Warnungen bei nicht kompatiblem Code

Entwickler(innen), die den AOT-Compiler für ein ASP.NET Core-Projekt aktivieren, erhalten nun Warnungen, falls sie Methoden aufrufen, die nicht kompatibel mit dem AOT-Compiler sind (siehe Abbildung).

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddControllers();

var app = builder.Build();

app.Run();
```

IL2026: Using member 'Microsoft.Extensions.DependencyInjection.MvcServiceCollectionExtensions.AddControllers(IServiceCollection)' which has 'RequiresUnreferencedCodeAttribute' can break functionality when trimming application code. MVC does not currently support native AOT. https://aka.ms/aspnet/nativeaot

Abbildung: Warnung, dass der Aufruf AddControllers() zum Aktivieren des Model-View-Controller-Frameworks nicht beim Ahead-of-Timer-Compiler möglich ist.

48.4.6 Mögliche und nicht mögliche Operationen bei Native AOT

Datenbankzugriffe sind beim AOT-Compiler allerdings weiterhin nicht mit dem Objekt-Relationalen Mapper Entity Framework Core möglich, da dieser immer noch Laufzeitkompilierung verwendet. Gleiches gilt für den zweitwichtigsten OR-Mapper der .NET-Welt, den Micro-ORM Dapper *https://github.com/DapperLib/Dapper*. In AOT-kompilierten Anwendungen können Entwicklerinnen und Entwickler derzeit nur DataReader, DataSet und

Command-Objekte aus ADO.NET oder das GitHub-Projekt NanORM *https://github.com/DamianEdwards/Nanorm* verwenden.

Das ist mit Native AOT auch in .NET 8.0 nicht möglich, selbst wenn man eine der o.g. Anwendungsarten erstellt:

- Laufzeitcodegenerierung (Reflection Emit)
- dynamisches Nachladen von Assemblies (Add-Ins/Plug-Ins)
- COM-Interop
- WinRT-APIs
- Windows Management Instrumentation
- Zugriff auf Active Directory Services
- C++/CLI
- AOT mit WebAPIs im IIS
- Entity Framework Core
- Dapper
- JSON-Serialisierung mit JSON.NET (Newtonsoft JSON)
- AutoMapper und viele andere Drittanbieterbibliotheken

Beispiel, was möglich ist, sind:

- Reguläre Ausdrücke
- Dateisystemzugriffe
- JSON-Serialisierung mit System.Text.Json
- ADO.NET
- NanORM
- Dependency Injection mit Microsoft Dependency Injection-Container (Microsoft.Extensions.DependencyInjection) und AutoFac

48.4.7 Performance bei Native AOT

Für .NET 8.0 hat Microsoft Zahlen herausgegeben, weil Auswirkungen der Native AOT-Compiler auf WebAPIs hat. Man sieht in der folgenden Grafik:

- Größe des Kompilats, RAM-Bedarf (insbes. auf Linux) und Startdauer werden wesentlich geringer.
- Die Ausführungsgeschwindigkeit sinkt aber leider auch etwas, denn der Native-AOT-kompilierte Code schafft weniger "Requests per Second" (RPS).

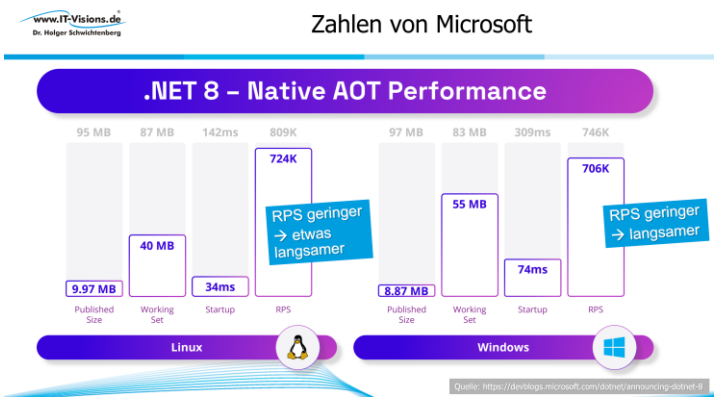


Abbildung: Quelle: Microsoft

49 Anhang: Syntaxreferenz: C# versus Visual Basic .NET

Für Umsteiger von Visual Basic .NET zu C# stellen nachfolgende Tabellen die wichtigsten syntaktischen Konstrukte direkt gegenüber.

Typdefinitionen	C#	Visual Basic
Namensraumfestlegung für einen Block	namespace de.ITVisions { … }	Namespace de.ITVisions … End Namespace
Namensraumfestlegung auf Dateiebene	namespace de.ITVisions	---
Namensraumimport auf Dateiebene	using de.ITVisions;	Imports de.ITVisions
Namensraumimport mit Alias	using ITV = de.ITVisions;	Imports ITV = de.ITVisions
Namensraumimport global für Projekt im Code	global using de.ITVisions;	--
Namensraumimport global für Projekt in Projektdatei	<ItemGroup> <Using Include="de.ITVisions"/>	<ItemGroup> <Import Include="de.ITVisions" />
Implizite Namensräume	<PropertyGroup> <ImplicitUsings>enable</ImplicitUsings>	---

Typdefinitionen	C#	Visual Basic .NET
Klasse (Referenztyp)	class Klasse { … }	Class Klasse … End Class
Struktur (Wertetyp)	struct Strukturname { … }	Structure Strukturname … End Structure
Struktur (Wertetyp), die nur auf dem Stack lebt	ref struct Strukturname { … }	---
Klasse mit Wertesemantik	record Name { … } oder record class Name { … }	---
Struktur mit Record-Eigenschaften	record struct Name { … }	---
Öffentliche Klasse	public class Klasse { … }	Public Class Klasse … End Class
Klasse nur innerhalb der Assembly sichtbar	internal class Klasse { … }	Friend Class Klasse … End Class
Partielle Klasse	partial class Klasse { … }	Partial Class Klasse … End Class

Typdefinitionen	C#	Visual Basic .NET
Statische Klasse (nur statische Mitglieder)	static class Klasse { … }	Module Klasse … End Module
Generische Klasse	public class Klasse<T1, T2>	Public Class Klasse(Of T1, T1)
Implementierungsvererbung	class C1 : C2	Inherits
Abstrakte Klasse	abstract	MustInherit
Finale Klasse	sealed	NotInheritable
Deklaration einer Schnittstelle	interface IXyz	Interface IXyz
Schnittstellenvererbung	class C2 : C1	Class C2 Implements C1
Anonymer Typ	var obj = new { Name = "World Wide Wings", Gruendungsdatum = new DateTime(2005, 01, 01), Vorstand = Vorstandsmitglieder };	Dim obj = New With {.Name = "World Wide Wings", .Gruendungsdatum = New DateTime(2005, 1, 1), .Vorstand = Vorstandsmitglieder}
Tupel	var dozent = (ID: 1, Name: "Holger Schwichtenberg", DOTNETExperte: true);	Dim dozent = (ID:=1, Name:="Holger Schwichtenberg", DOTNETExperte:=True)
Array	byte[] x;	Dim x as Byte()
Array-Größenveränderung	Array.Resize()	ReDim Preserve
Array initialisieren	string[] WebSites1 = new string[] { "www.dotnet-doktor.de", "www.dotnet-lexikon.de" }; oder ab C# 12.0: string[] WebSites2 = ["www.dotnet-doktor.de", "www.dotnet-lexikon.de"];	Dim WebSites As String() = {"www.dotnet-doktor.de", "www.dotnet-lexikon.de"}
Enumeration	enum name { a, b, c } enum name { a = 10, b = 20, c }	Enum name a b End Enum

Variablen und Literale	C#	Visual Basic .NET
Wertlose Werttypen	Typ? Oder Nullable<Typ>	Nullable(Of Typ)
Variablendeklaration/ Attributdefinition als Field	Typ x	Dim x as Typ
Implizit typisierte Variable	var x = Wert	Dim x = Wert
Zeichenketten mit Escape-Sequenz	"Er sagte:\r\n\"Hallo Welt!\"";	"Er sagte:" & vbCrLf & """Hallo Welt!"""
Zeichenketten ohne Escape-Sequenz	@"c:\temp\daten.txt"	"c:\temp\daten.txt"
Einzelne Zeichen	char Wichtigkeit = 'A';	Dim Wichtigkeit As Char = "A"
String Interpolation	$"Er sagte am {Zeitpunkt:d}:\n\r	$"Er sagte am {Zeitpunkt:d}:{vbLf}

Variablen und Literale	C#	Visual Basic .NET
	{seineAussage}";	{seineAussage}!"

Zahlenliterale	byte z1 = 123;
	short z2 = 123;
	int z3 = 123;
	long z4 = 123;
	float z5 = 123.45f;
	double z6 = 123.45d;
	decimal z7 = 123.45m;
		Dim z1 As Byte = 123
		Dim z2 As Short = 123
		Dim z3 As Integer = 123
		Dim z4 As Long = 123
		Dim z5 As Single = 123.45
		Dim z6 As Double = 123.45
		Dim z7 As Decimal = 123.45

Datumsliterale	new DateTime(2014,12,24)	#12/24/2014#
XML-Literale	---	Dim x As XElement = _ <Flug ID="347"> <Abflugort>Madrid</Abflugort> <Zielort>Paris</Zielort> </Flug>
Zeilenumbruch (Zeilenvorschubzeichen ASCII-Code 10 & Wagenrücklauf ASCII-Code 13)	"\n\r"	vbCrLf
Zeigerprogrammierung (unsafe)	unsafe, &x, *x	---
Zeigerprogrammierung (safe)	ref int z = ref i;	---

Typmitglieder	C#	Visual Basic .NET

Attributdefinition als Property mit expliziten Field	private string x;
	public string X
	{
	get { return x; }
	set { x = value; }
	}
		Private _X as String
		Property X() As String
		Get
		Return _X
		End Get
		Set(ByVal value As String)
		_X = value
		End Set
		End Property

Attributdefinition als automatisches Property (Automatic Properties/Auto-Implemeted Properties)	public Type Name { get; set; }	Public Property X As String

Methode ohne Parameter und ohne Rückgabetyp	void f()
	{
	…
	}
		Sub f()
		…
		End Sub

Typmitglieder	C#	Visual Basic .NET

Methode mit Parametern aber ohne Rückgabetyp	void f(string s, int i)
	{
	…
	}
		Sub f(ByVal s As String, ByVal i As Integer)
		…
		End Sub

Methode mit Parametern und mit Rückgabewert	Typ f(string s, int i)
	{
	Typ t = new Typ();
	…
	return t;
	}
		Function f(ByVal s As String, ByVal i As Integer) as Typ
		Dim t as Typ
		…
		Return t
		End Function

Überladene Methode	keine Zusatzangabe	Overloads
Methode verlassen	return	Return
Methode verlassen und beim nächsten Aufruf danach fortsetzen	yield	Yield
Bezug auf Basisklasse	base	MyBase
Bezug auf aktuelle Klasse	Name der Klasse	MyClass
Bezug auf das aktuelle Objekt	this	Me
Konstantes Mitglied	const	Const
Methoden ohne Rückgabewert	void	Sub
Statisches Mitglied	static	Shared
Überschreiben einer Methode	override	Overrides
Abstrakte Methode	abstract	MustOverride
Versiegelte Methode	sealed	NotOverridable
Überschreibbare Methode	virtual	Overridable
Verdeckendes Mitglied	keine Zusatzangabe	Shadows
Konstruktor	public Klassenname() { … }	Sub New() … End Sub

Primärkonstruktor	public Klassenname(int a, string b)
	{
	public int A { get; init; } = a;
	public string B { get; set; } = b;
	…
	}
		--- 	

Desktruktor/Finalizer	~Person() { … }	Sub Finalize() … End Sub
Referenz auf eine Methode	delegate	Delegate
Mitglied mit Ereignissen	---	WithEvents
Bindung einer Ereignisbehandlungsroutine	+= -=	Handles AddHandler RemoveHandler
Partielle Methode (Deklaration)	public partial void f();	Partial Public Sub f() End Sub

Typmitglieder	C#	Visual Basic .NET

Partielle Methode (Implementierung)	public partial void f()
	{
	…
	}
		Partial Public Sub f()
		….
		End Sub

Partielles Property (Deklaration)	public partial int x { get; set; }	---

Partielles Property (Implementierung)	public partial int x
	{
	get
	{
	…
	}
	set
	{
	…
	}
	}
		--- 	

Typen verwenden	C#	Visual Basic .NET
Programm-Einsprungpunkt	static void Main(string[] args)	Sub Main(ByVal args() As String)
Klasse instanzieren	new Klasse()	New Klasse
Generische Klasse instanzieren	new Klasse<Typ>()	New Klasse(of Typ)
Anonyme Methoden	+= delegate(){ … }	---
LINQ-Abfrageausdruck	(from m in Menge where m.Feld < 1000 select m).Skip(1200).Take(10)	From m In Menge Where m.Feld < 1000 Select m Skip 1200 Take 10;
Lambda-Ausdruck	Func<string, int> f3 = s => s.Length;	Dim f3 As Func(Of String, Integer) = Function(s) s.Length
Blockbildung für Objekte	---	With obj … End With

Datentyp	C#	Visual Basic .NET
Ganzzahl / 1 Byte	byte	Byte
Ganzzahl / Boolean	bool	Boolean
Ganzzahl / 2 Bytes	short	Short
Ganzzahl / 4 Bytes	int	Integer
Ganzzahl / 8 Bytes	long	Long
Zahl / 4 Bytes	float	Single
Zahl / 8 Bytes	double	Double
Zahl / 12 Bytes	decimal	Decimal
Zeichen / 1 Byte oder 2 Bytes	char	Char
Zeichenkette	string	String
Datum/Uhrzeit	DateTime	Date

Operatoren Zeichenketten	C#	Visual Basic .NET
Zeichenkettenverbindung	+	&

Operatoren Mathematik	C#	Visual Basic .NET
Addition	+	+
Subtraktion	–	–
Multiplikation	*	*
Division	/	/
Ganzzahldivision	/	\
Modulus	%	Mod
Potenz	Math.Pow(x,y)	^
Negation	~	Not
Inkrement	++	---
Dekrement	- -	---

Operatoren Zuweisung	C#	Visual Basic .NET
Einfache Zuweisung	=	=
Addition	+ =	+ =
Subtraktion	- =	- =
Multiplikation	*=	*=
Division	/=	/=
Ganzzahl-Division	/=	\=
Zeichenkettenverbindung	+=	&=
Modulo (Divisionsrest)	%=	---
Bit-Verschiebung nach links	<< =	<< =
Bit-Verschiebung nach rechts	>> =	>> =
Bit-weises UND	&=	---
Bit-weises XOR	^ =	---
Bit-weises OR	| =	---

Operatoren Vergleich	C#	Visual Basic .NET
Kleiner	<	<
Kleiner gleich	< =	< =
Größer	>	>
Größer gleich	> =	> =
Gleich	= =	=
Nicht gleich	!=	< >
Objektvergleich	= =	Is

Objektvergleich (negativ)	!=	IsNot
Objekttypvergleich	x is Klasse	TypeOf x Is Klasse
Zeichenkettenvergleich	==	=
Zeichenkettenverbindung	+	&

Operatoren Logik	**C#**	**Visual Basic .NET**
UND	&&	And
ODER	||	Or
NICHT	!	Not
Short-circuited UND	&&	AndAlso
Short-circuited ODER	||	OrElse

Operatoren Bit	**C#**	**Visual Basic .NET**
Bit-weises UND	&	And
Bit-weises XOR	^	Xor
Bit-weises OR	|	Or
Bit-Verschiebung nach links	<<	<<
Bit-Verschiebung nach rechts	>>	>>

Bedingungs-operatoren	**C#**	**Visual Basic .NET**
Bedingungsoperator	Bedingung ? wert1 : wert2	IIF-Funktion und If-Operator
NULL-Sammeloperator	Objekt ?? wert1 : wert2	---
NULL-Bedingungsoperator	obj?.mitglied	obj?.mitglied

Typoperatoren	**C#**	**Visual Basic .NET**
Typermittlung	typeof(obj) obj.GetType()	obj.GetType()
Typvergleich	k1 is Kunde	TypeOf k1 Is Kunde
Typkonvertierung	x as Klasse oder ((Klasse) x)	CType(x,Klasse)
Namensoperator	nameof(x)	NameOf(x)

Bedingungen	**C#**	**Visual Basic .NET**
Einfache Bedingung	if (Bedingung) {…} else {…}	If Bedingung Then … Else … End If
Mehrfachverzweigung	switch (a) { case 1: … break; case 2: … break; case 3: … break;	Select Case a Case 1: … Case 2: … Case 3: … Case Else: …

Bedingungen	C#	Visual Basic .NET
	default: … break; }	End Select

Schleifen	C#	Visual Basic .NET
Kopfgeprüfte bedingte Schleife	while (c < 10) { c++; }	While c < 10 c += 1 End While
Fußgeprüfte bedingte Schleife	do { d++; } while (d < 10);	Do d += 1 Loop While d < 10
Zählschleifen	for (int a = 1; a <= 10; a++) { … }	For a As Integer = 1 To 10 Step 1 … Next

Schleifen über Mengen	foreach (int e in zahlen) {
	…
	}
		For Each x As Integer In y
		…
		Next

50 Anhang: Neuerungen in früheren Versionen

Diese Kapitel bleibt auch in der C# 12.0-Version des Buchs als Anhang erhalten, weil viele Unternehmen erst jetzt vom klassischen .NET Framework mit C# 7.3 auf die moderne .NET-Welt mit C# 12.0 umsteigen, auf offiziellen Support von Microsoft Wert legen und die neueren Versionen erst damit nutzen können.

50.1 Neuerungen in C# 8.0

Die fertige Version von C# 8.0 ist am 23.09.2019 im Rahmen von .NET Core 3.0 und Visual Studio 2019 v16.3 erschienen.

Die wichtigsten Neuerungen in C# 8.0 sind:

- Nullable Reference Types string? und Null-Forgiveness-Operator !.
  → Kapitel "Behandlung von null/Null-Referenz-Prüfung / Nullable Reference Types"
- Standardimplementierungen in Schnittstellen (*)
  → Kapitel "Schnittstellen/Standardimplementierungen in Schnittstellen"
- Index ^ und Range .. (*)
  → Kapitel "Operatoren/Index und Range"
- Switch Expressions
  → Kapitel "Verzweigungen/Switch Expressions"

Weitere Neuerungen in C# 8.0 sind:

- Null Coalescing Assignment ??=
  → Kapitel "Operatoren/Null Coalescing Assignment"
- Alternative für Verbatim Interpolated Strings: @$ zusätzlich zu $@
  → Kapitel "Datentypen/Konsolenausgabenformatierung mit ANSI-Codes

Mit den uralten VT100/ANSI-Codes (siehe *https://en.wikipedia.org/wiki/ANSI_escape_code*) kann man auch heute noch in Konsolenanwendungen zahlreiche Formatierungen auslösen, z.B. 24-Bit-Farben, Fettschrift, Unterstreichen, Durchstreichen, Blinken usw. Die VT100/ANSI-Codes werden durch das ESCAPE-Zeichen (ASCII-Zeichen 27, hexadezimal: 0x1b) eingeleitet.

Vor C# 13.0 konnte man dieses ESCAPE-ASCII-Zeichen 27 in .NET-Konsolenanwendungen bei Console.WriteLine() nur umständlich ausdrücken über \u001b, \U0000001b oder \x1b, wobei letzteres nicht empfohlen ist: "Wenn Sie die Escapesequenz \x verwenden, weniger als vier Hexadezimalziffern angeben und es sich bei den Zeichen, die der Escapesequenz unmittelbar folgen, um gültige Hexadezimalziffern handelt (z. B. 0–9, A–F und a–f), werden diese als Teil der Escapesequenz interpretiert. \xA1 erzeugt beispielsweise "¡" (entspricht dem Codepunkt U+00A1). Wenn das nächste Zeichen jedoch "A" oder "a" ist, wird die Escapesequenz stattdessen als \xA1A interpretiert und der Codepunkt "ਚ" erzeugt (entspricht dem Codepunkt U+0A1A). In solchen Fällen können Fehlinterpretationen vermieden werden, indem Sie alle vier Hexadezimalziffern (z. B. \x00A1) angeben." [*https://learn.microsoft.com/de-de/dotnet/csharp/programming-guide/strings/*].

Hinweis: ਚ ist ein Panjabi-Schriftzeichen. Panjabi ist eine in Pakistan und Indien gesprochene Sprache.

Typischerweise sahen Ausgaben mit VT100/ANSI-Escape-Codes dann aus wie im nächsten Listing.

Listing: Bisherige VT100/ANSI-Escape-Codes

```
Console.WriteLine("This is a regular text");
Console.WriteLine("\u001b[1mThis is a bold text\u001b[0m");
Console.WriteLine("\u001b[2mThis is a dimmed text\u001b[0m");
Console.WriteLine("\u001b[3mThis is an italic text\u001b[0m");
Console.WriteLine("\u001b[4mThis is an underlined text\u001b[0m");
Console.WriteLine("\u001b[5mThis is a blinking text\u001b[0m");
Console.WriteLine("\u001b[6mThis is a fast blinking text\u001b[0m");
Console.WriteLine("\u001b[7mThis is an inverted text\u001b[0m");
Console.WriteLine("\u001b[8mThis is a hidden text\u001b[0m");
Console.WriteLine("\u001b[9mThis is a crossed-out text\u001b[0m");
Console.WriteLine("\u001b[21mThis is a double-underlined text\u001b[0m");
Console.WriteLine("\u001b[38;2;255;0;0mThis is a red text\u001b[0m");
Console.WriteLine("\u001b[48;2;255;0;0mThis is a red background\u001b[0m");
Console.WriteLine("\u001b[38;2;0;0;255;48;2;255;255;0mThis is a blue text with a
yellow background\u001b[0m");
```

Seit C# 13.0 gibt es nun \e als Kurzform für das ESCAPE-Zeichen ASCII 27 ein, sodass die Zeichenfolgen deutlich kompakter und übersichtlicher werden (siehe nächstes Listings).

Listing: Etwas übersichtlichere VT100/ANSI-Escape-Codes mit der neuen Abkürzung \e in C# 13.0

```
Console.WriteLine("This is a regular text");
Console.WriteLine("\e[1mThis is a bold text\e[0m");
Console.WriteLine("\e[2mThis is a dimmed text\e[0m");
Console.WriteLine("\e[3mThis is an italic text\e[0m");
Console.WriteLine("\e[4mThis is an underlined text\e[0m");
Console.WriteLine("\e[5mThis is a blinking text\e[0m");
Console.WriteLine("\e[6mThis is a fast blinking text\e[0m");
Console.WriteLine("\e[7mThis is an inverted text\e[0m");
Console.WriteLine("\e[8mThis is a hidden text\e[0m");
Console.WriteLine("\e[9mThis is a crossed-out text\e[0m");
Console.WriteLine("\e[21mThis is a double-underlined text\e[0m");
Console.WriteLine("\e[38;2;255;0;0mThis is a red text\e[0m");
Console.WriteLine("\e[48;2;255;0;0mThis is a red background\e[0m");
Console.WriteLine("\e[38;2;0;0;255;48;2;255;255;0mThis is a blue text with a yell
ow background\e[0m");
```

Die Abbildung zeigt das Ergebnis, das sowohl beide Listings produziert.

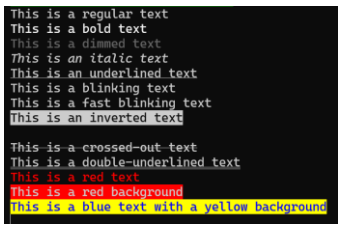


Abbildung: Die Ausgabe der beiden vorherigen Listings sieht gleich aus.

So gibt man ein Farbraster mit den neuen Escape-Codes aus (das war mit den alten Escape-Codes natürlich auch schon möglich, es ist jetzt nur prägnanter):

```
Console.WriteLine("\n\nFarbraster:");
for (int i = 0; i < 16; i++)
{
 for (int j = 0; j < 16; j++)
 {
  Console.Write("\e[48;5;" + (i * 16 + j) + "m" + (i * 16 + j).ToString().PadLeft
(4));
 }
 Console.WriteLine("\e[0m");
}
```

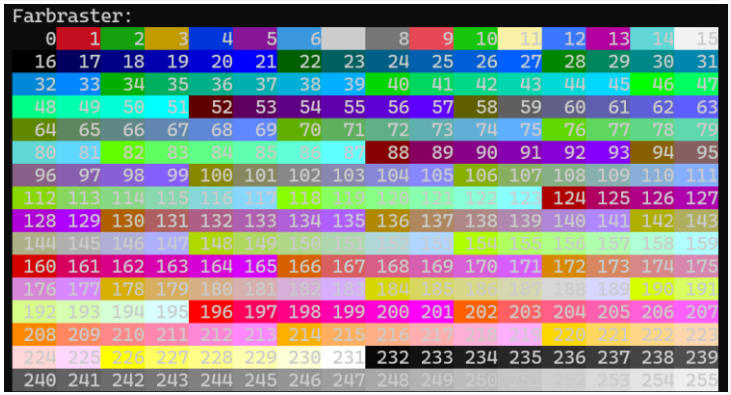


Abbildung: Farbraster in der Konsole mit ANSI-Codes

- String Interpolation"
- Asynchrone Streams und await foreach (*)
  → Kapitel "Iteratoren/Asynchrone Streams"
- Static Local Functions
  → Kapitel "Methoden/Statische lokale Funktionen (seit C# 8.0) "

- Using Declarations ohne Blöcke
  → "IDisposable/Vereinfachte Using-Deklarationen "
- Unmanaged Constructed Types
- Readonly-Mitglieder in einer Struktur
  → Kapitel "Strukturen/Readonly für einzelne Mitglieder einer Struktur "
- Dispose() für ref structs (Strukturen auf dem Stack)
  → Kapitel "IDisposable/IDispose für Strukturen auf dem Stack"

(*) Die mit Stern markierten Sprachfeatures erfordert .NET Standard 2.1, d.h. nur für .NET Core, Xamarin, Mono und Unity. Diese Sprachefeatures sind also im klassischen .NET Framework nicht verfügbar und Microsoft plant auch nicht, diese dort noch einzubauen.

50.2 Neuerungen in C# 9.0

Die fertige Version von C# 9.0 ist am 10.11.2020 im Rahmen von .NET 5.0 und Visual Studio 2019 v16.8 erschienen.

Hinweise: C# 9.0 wird offiziell von Microsoft nur ab .NET 5.0 unterstützt ("C# 9.0 is supported only on .NET 5 and newer versions." [*learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version*]. Man kann allerdings die meisten (aber nicht alle!) C# 9.0-Sprachfeatures auch in .NET Core, .NET Framework und Xamarin nutzen. Dazu muss man die <LangVersion> in der Projektdatei erhöhen. Dies wird im Kapitel "Erste C#-Schritte/Festlegen der Compilerversion" beschrieben.

Notwendige Visual Studio-Version für C# 9.0 ist Visual Studio 2019 v16.8 oder höher.

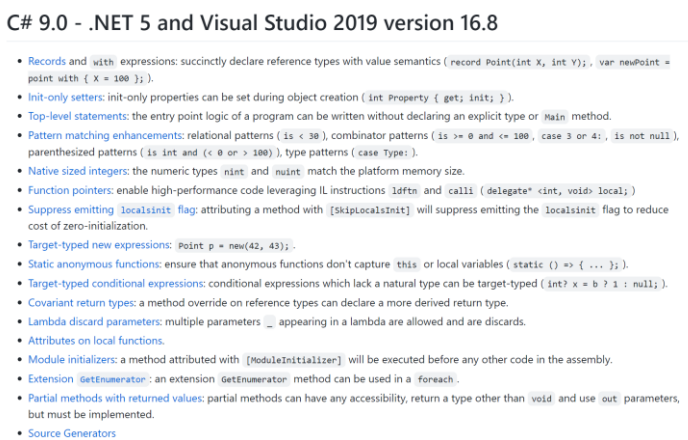


Abbildung: Übersicht über die Neuerungen in C# 9.0
Quelle: Microsoft
[github.com/dotnet/csharplang/blob/main/Language-Version-History.md]

Die wichtigsten Neuerungen in C# 9.0 sind:

- Record-Typen → siehe Kapitel "Record-Typen"
- Programme ohne Main() → Siehe Kapitel "Top-Level Statements"
- Properties, die nach Initialisierung unveränderlich sind (Init Only Properties mit Init Only Setters) → Siehe Kapitel "Attribute/Properties, die nach Initialisierung unveränderlich sind"
- Verwendung des Operators *new* ohne Typangabe (Target-Typed New Expression) → Siehe Kapitel "Klassendefinition/Instanzierung mit dem Operator new")
- Aufhebung der Restriktionen für partielle Methoden → Siehe Kapitel "Partielle Methoden"
- Statische anonyme Funktionen und Discard-Variablen in Lambdas → Siehe Kapitel "Lambda-Ausdrücke"
- Annotationen auf lokale Funktionen → Siehe Kapitel "Lokale Funktion"
- Erweiterung des Pattern Matching → Siehe Kapitel "Verzweigungen/Pattern Matching"
- Modul-Initialisierer → Siehe Kapitel "Modul-Initialisierer".
- Source Code-Generatoren: Mit diesen neuen Code-Generatoren kann ein Entwickler zusätzlichen Programmcode zur Kompilierungszeit erzeugen, der zusammen mit dem eigentlichen Programmcode kompiliert wird. Damit kann man z.B. Annotationen eine Bedeutung geben. → Siehe Kapitel "Source Code-Generatoren".

50.3 Neuerungen in C# 10.0

C# 10.0 ist zusammen mit Visual Studio 2022 und .NET 6.0 am 8.11.2021 erschienen.

Hinweise: C# 10.0 wird offiziell von Microsoft erst ab .NET 6.0 unterstützt ("C# 10.0 is supported only on .NET 6 and newer versions." [*learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version*]. Man kann allerdings die meisten (aber nicht alle!) C# 11.0-Sprachfeatures auch in älteren .NET-Versionen einschließlich .NET Framework, .NET Core und Xamarin nutzen. Dazu muss man die <LangVersion> in der Projektdatei auf "10.0" erhöhen. Dies wird im Kapitel "Erste C#-Schritte/Festlegen der Compilerversion" beschrieben.

Notwendige Visual Studio-Version für C# 10.0 ist Visual Studio 2022 v17.0 oder höher. Eine Verwendung von C# 10.0 sowohl mit Visual Studio for Mac 2022 als auch einer aktuellen Version von Visual Studio Code und anderen OmniSharp-kompatiblen Editoren [*www.omnisharp.net*] ist möglich.

C# 10.0 - .NET 6 and Visual Studio 2022 version 17.0

- Record structs and `with` expressions on structs (`record struct Point(int X, int Y);` , `var newPoint = point with { X = 100 };`).
- Global using directives: `global using` directives avoid repeating the same `using` directives across many files in your program.
- Improved definite assignment: definite assignment and nullability analysis better handle common patterns such as `dictionary?.TryGetValue(key, out value) == true` .
- Constant interpolated strings: interpolated strings composed of constants are themselves constants.
- Extended property patterns: property patterns allow accessing nested members (`if (e is MethodCallExpression { Method.Name: "MethodName" })`).
- Sealed record ToString: a record can inherit a base record with a sealed `ToString` .
- Incremental source generators: improve the source generation experience in large projects by breaking down the source generation pipeline and caching intermediate results.
- Mixed deconstructions: deconstruction-assignments and deconstruction-declarations can be blended together (`(existingLocal, var declaredLocal) = expression`).
- Method-level AsyncMethodBuilder: the AsyncMethodBuilder used to compile an `async` method can be overridden locally.
- #line span directive: allow source generators like Razor fine-grained control of the line mapping with `#line` directives that specify the destination span (`#line (startLine, startChar) - (endLine, endChar) charOffset "fileName"`).
- Lambda improvements: attributes and return types are allowed on lambdas; lambdas and method groups have a natural delegate type (`var f = short () => 1;`).
- Interpolated string handlers: interpolated string handler types allow efficient formatting of interpolated strings in assignments and invocations.
- File-scoped namespaces: files with a single namespace don't need extra braces or indentation (`namespace X.Y.Z;`).
- Parameterless struct constructors: support parameterless constructors and instance field initializers for struct types.
- CallerArgumentExpression: this attribute allows capturing the expressions passed to a method as strings.

Abbildung: Übersicht über die Neuerungen in C# 10.0 | Quelle: Microsoft [github.com/dotnet/csharplang/blob/main/Language-Version-History.md]

Das folgende Bild realisiert das kleine Kunststück, fast alle neuen C# 10.0-Sprachfeatures in zwei überschaubare und kommentierte Listings unterzubringen, die zusammen auch noch Sinn machen. Verstehen Sie dies als Kurzreferenz. Natürlich finden Sie eine ausführliche Beschreibung in den verschiedenen Kapiteln dieses Buchs.

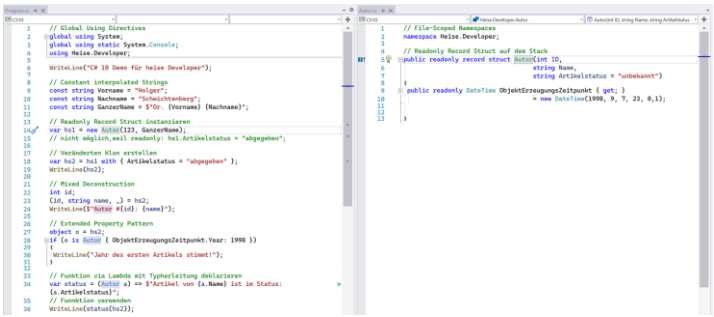


Abbildung: Fast alle neuen C# 10.0-Features auf einen Blick.

Sie finden in diesem Buch:

- Kapitel "Datentypen": Neuerungen zu Interpolated Strings
- Kapitel "Verzweigungen/ Pattern Matching": Neuerungen zum Pattern Matching
- Kapitel "Methoden": Caller Argument Expressions

- Kapitel "Namensräume": Alle Neuerungen zu den Namensräumen (File-Scoped Namespaces, Global Using Directives, Implicit Using Directives)
- Kapitel "Record-Typen": Alle Neuerungen zu Record-Typen (record class, record struct, sealed ToString())
- Kapitel "Strukturen/With-Ausdrücke": Einsatz von Klonen mit with bei Strukturen und anonymen Typen.
- Kapitel "Strukturen/Strukturen mit parameterlosem Konstruktor": Strukturen mit parameterlosem Konstruktor
- Kapitel "Tupel": Mixed Deconstruction
- Kapitel "Funktionale Programmierung/Lambda-Ausdrücke": Typherleitung, explizite Rückgabetypen und Annotationen/Attribute für Lambda-Ausdrücke

50.4 Neuerungen in C# 11.0

C# 11.0 ist zusammen mit Visual Studio 2022 Version 17.4 und .NET 7.0 am 8.11.2022 erschienen.

Wie schon bei .NET 6.0/C# 10.0 verwendet Microsoft bei .NET 7.0/C# 11.0 an vielen, aber nicht allen Stellen die Versionsnummer ohne ".0". Hier wird einheitlich die Schreibweise mit ".0" verwendet. Anders als .NET 6.0 besitzt die 7.0-Version keinen "Long-Term-Support", sondern nur "Standard Support" (früher "Current Version", zwischenzeitlich auch "Short-Termin-Support (STS)" genannt). Dafür gibt es also Unterstützung und Updates für 18 Monate, also von November 2022 bis Mai 2023.

Hinweise: C# 11.0 wird offiziell von Microsoft erst ab .NET 7.0 unterstützt ("C# 11.0 is supported only on .NET 7 and newer versions." [*learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version*]. Man kann allerdings die meisten (aber nicht alle!) C# 11.0-Sprachfeatures auch in älteren .NET-Versionen einschließlich .NET Framework, .NET Core und Xamarin nutzen. Dazu muss man die <LangVersion> in der Projektdatei auf "11.0" erhöhen. Dies wird im Kapitel "Erste C#-Schritte/Festlegen der Compilerversion" beschrieben.

Notwendige Visual Studio-Version für C# 11.0 ist Visual Studio 2022 v17.4 oder höher. Eine Verwendung von C# 11.0 ist sowohl mit Visual Studio for Mac 2022 als auch einer aktuellen Version von Visual Studio Code und anderen OmniSharp-kompatiblen Editoren [*www.omnisharp.net*] ist möglich.

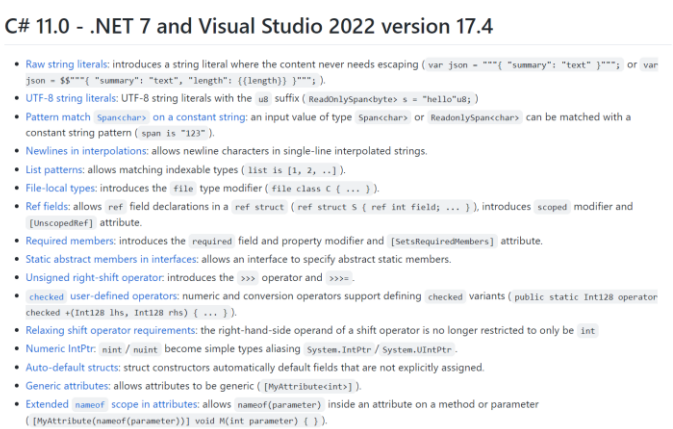


Abbildung: Übersicht über die Neuerungen in C# 11.0 | Quelle: Microsoft [github.com/dotnet/csharplang/blob/main/Language-Version-History.md]

Sie finden in diesem Buch:

- Kapitel "Grundkonzepte": Warnungen bei Typnamen komplett in Kleinbuchstaben
- Kapitel "Datentypen": Datentypen nint und nuint, Zeilenumbrüche innerhalb von Interpolationsausdrücken, Raw Literal Strings und UTF-8-Zeichenkettenliterale
- Kapitel "Operatoren": Erweiterte Einsatzgebiete von nameof()
- Kapitel "Verzweigungen": Pattern Matching für Listen und Teilmengen (List Pattern und Slice Pattern)
- Kapitel "Klassendefinition": File-local Types
- Kapitel "Datenmitglieder / Attribute": Pflichtmitglieder (Required Members)
- Kapitel "Schnittstellen": Statische abstrakte Properties und Methoden in Schnittstellen
- Kapitel "Annotationen (.NET-Attribute)": Annotationen mit Typparametern
- Kapitel "Generische Klassen": Generische Mathematik
- Kapitel "Strukturen": Auto-Defaults Structs
- Kapitel "Operatorüberladungen": Operatorüberladungen in Schnittstellen mit Hilfe von statischen abstrakten Methoden
- Kapitel "Performanceoptimierungen": Ahead-of-Timer-Compiler (Native AOT)

51 Anhang: Quellen im Internet

Neuerungen in C# 13.0

https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-13

Breaking Changes in C# 13.0

https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/breaking-changes/compiler%20breaking%20changes%20-%20dotnet%209

Projekt für das Design der Programmiersprache C#

https://github.com/dotnet/csharplang

Projekt für die Implementierung des neuen C#-Compilers

https://github.com/dotnet/roslyn

Versionsgeschichte der C#-Sprachsyntax

https://github.com/dotnet/csharplang/blob/master/Language-Version-History.md

Versionsgeschichte des neuen C#-Compilers

https://github.com/dotnet/roslyn/blob/master/docs/wiki/NuGet-packages.md

Language Feature Status

https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md

C# ECMA Standard

https://www.ecma-international.org/publications-and-standards/standards/ecma-334/

Weiterentwicklung des C# ECMA Standards

https://github.com/dotnet/csharpstandard

NuGet-Paket des C#-Compilers

https://www.nuget.org/packages/Microsoft.Net.Compilers

.NET-Entwickler-Lexikon

https://www.dotnet-lexikon.de

Website zu .NET 9.0

https://www.dotnet9.de

52 Anhang: Versionsgeschichte dieses Buchs

Die folgende Tabelle zeigt die Versionen, die von diesem Fachbuch erschienen sind, sowie die darin besprochenen Blazor-Versionen.

Hinweis: Diese Tabelle ist eine wichtige Referenz für die Leser, die sich aktuelle Versionen des Buchs beschaffen (z.B. über das PDF-Abo) und wissen wollen, was sich geändert hat. Wenn Sie das Buch erstmalig lesen, können Sie dieses Kapitel überspringen.

Die Behandlung einer neuen Versionsnummer des Produkts und die daraus resultierende Änderung des Buchtitels erfordert gemäß Amazon-Richtlinien ein neues Buchprojekt. In diesem Fall wird die Versionsnummer des Buchs an der ersten Stelle hochgezählt (z.B. 1.4 auf 2.0).

Eine Änderung der Versionsnummer an der zweiten Stelle (z.B. 1.3 auf 1.4) sind Aktualisierungen oder Erweiterungen, die keine Titeländerung erfordern.

Ergänzungen der Versionsnummer an der dritten Stelle (z.B. 1.2.2 auf 1.2.3) sind kleine Korrekturen im Buch, die nicht explizit in dieser Versionstabelle erscheinen. Das Erscheinungsdatum auf der Titelseite entspricht dem Erscheinungsdatum der Unterversion, kann also von dem in der Tabelle genannten Erscheinungsdatum der übergeordneten Version abweichen.

Leider sind Preiserhöhungen mit steigendem Buchumfang notwendig, da der Arbeitsaufwand der ständigen Aktualisierungen dieses Buchs sehr hoch ist.

Buchversion Datum	Umfang	C#-Version	Bemerkung
13.0 01.11.2024	421 Seiten	13.0	▪ Basisversion des Buchs

53 Stichwortverzeichnis (Index)

Es sind hier jeweils nur die zentralen Stellen im Buch verlinkt. Um alle Vorkommnisse eines Begriffs zu finden, nutzen Sie bitte die Volltextsuche im PDF, das Sie als Käufer des gedruckten Buchs kostenfrei bekommen (siehe Kapitel "Über dieses Fachbuch").

54 Werbung in eigener Sache ☺

54.1 Dienstleistungen

54.2 Aktion "Buch für Buchrezension"

Ich möchte Sie animieren, eine Rezension dieses Fachbuchs bei Amazon.de zu schreiben. Als Dank dafür erhalten Sie kostenlos ein weiteres E-Book (PDF) aus meiner Buchreihe (wenn Sie dieses Buch als gedrucktes Buch gekauft haben, können Sie auch das PDF des selben Buchs erhalten!).

So geht es:

- Sie schreiben bei Amazon.de eine Rezension zu diesem Fachbuch.
- Nach dem Erscheinen der Rezension besuchen Sie die Webadresse *www.IT-Visions.de/Buchrezension*
- Füllen Sie bitte das Formular aus. Geben Sie dabei in den Details insbesondere den Buchwunsch und Ihren Rezensionstext an, damit wir dies auf Amazon.de überprüfen können. **Sie müssen nicht Ihr Amazon-Konto angeben!**
- Das www.IT-Visions.de-Kundenteam sendet Ihnen nach der Überprüfung das E-Book (PDF-Format) des gewünschten Buchs per E-Mail.

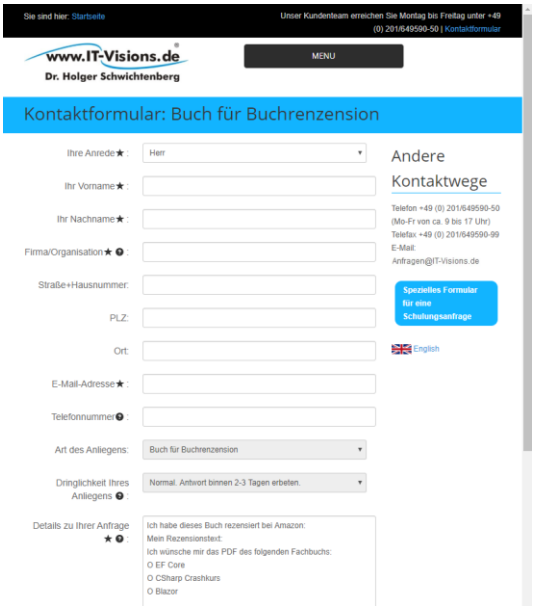


Abbildung: Webformular für die Aktion "Buch für Buchrezension"

54.3 Angebot "PDF-Buch-Abo"

Sie zahlen einen einmaligen Preis (ab 99 € zzgl. 7% MwSt) und erhalten für die Dauer des Abos:

- alle meine aktuellen .NET- und Web-Bücher
- in der jeweils aktuellen Version
- inklusive Zugriff auf alle früheren Ausgaben
- als PDF-E-Book zum Download
- alle 1-3 Monate die neusten Auflagen mit inhaltlichen Updates
- inklusive Neuausgaben, die im Abozeitraum erscheinen werden
- Zahlung auf Rechnung ohne Risiko
- ohne automatische Verlängerung!
  (Sie entscheiden selbst nach Laufzeitende, ob Sie das Abo fortsetzen wollen)

Enthalten sind folgende aktuelle Fachbücher:

- .NET 9.0 Update (~175 Seiten, Wert ~14,99 €)
- C# 13.0 Crashkurs (~420 Seiten, Wert ~29,99 €)
- Moderne Datenzugriffslösungen mit Entity Framework Core 9.0 (~824 Seiten, Wert: 49,99 €)
- Moderne Datenzugriffslösungen mit Entity Framework 6.x (287 Seiten, Wert: 24,99 €)
- Blazor 9.0 (~824 Seiten, Wert 49,99 €)
- Vue.js 3 (~260 Seiten, Wert ~19,99 €)

Die .NET-Bücher werden im Abstand von einigen Wochen aktualisiert, jeweils bis zum Erscheinen des Nachfolgebuchs. Die Nachfolgebücher, die im November 2025 erschienen werden, sind ebenfalls enthalten, sofern das Ihr Abo dann noch aktiv ist:

- .NET 10.0 Update
- C# 14.0 Crashkurs
- ASP.NET Core Blazor 10.0
- Moderne Datenzugriffslösungen mit Entity Framework Core 10.0

Ebenfalls im Buch-Abo enthalten sind alle vorherige Ausgaben zu C# 8.0 bis C# 12.0, ASP.NET Core Blazor 3.1 bis 8.0 sowie Entity Framework Core 3.1 bis 8.0.

Preise (jeweils zzgl. 7% Mehrwertsteuer):

- Einzelperson, 1 Jahr: 129 €
- Einzelperson, 2 Jahre: 218 € umgerechnet 109 € pro Jahr (15% Ersparnis)
- Einzelperson, 3 Jahre: 297 € umgerechnet 99 € pro Jahr (23% Ersparnis)
- Firmenlizenz bis zu 15 Personen, 1 Jahr: 399 €
- Firmenlizenz bis zu 50 Personen, 1 Jahr: 699 €
- Firmenlizenz bis zu 1000 Personen, 1 Jahr: 999 €

Weitere Informationen und Bestellung:

www.IT-Visions.de/BuchAbo