



## **Vielen Dank für Ihr Vertrauen in uns!**

Hallo Frau/Herr

Aalen,

Sie haben sich für ein Fachbuch vom BMU Verlag entschieden, und dafür möchten wir uns bei Ihnen recht herzlich bedanken.

Sollten Sie Fragen oder Probleme haben können Sie sich jederzeit gerne an uns oder den Autor wenden. Und nun wünschen wir Ihnen viel Erfolg beim Lernen mit diesem Buch.

Ihr Team vom BMU Verlag

C#

# KOMPENDIUM

PROFESSIONELL C# PROGRAMMIEREN LERNEN



## Eine umfassende Einführung in C#!

- ▶ Alle Grundlagen der Programmierung verständlich erklärt
- ▶ OOP, LINQ, Datenbanken, ASP.NET, WPF, MS-Test, u.v.m.
- ▶ Übungsaufgaben mit Musterlösungen nach jedem Kapitel



Inklusive eBook zum Download

# C# Kompendium

*Professionell C# Programmieren Lernen*

Robert Schiefele

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;  
detaillierte bibliografische Informationen sind im Internet über <http://dnb.d-nb.de> abrufbar.

©2021 BMU Media GmbH  
[www.bmu-verlag.de](http://www.bmu-verlag.de)  
info@bmu-verlag.de

Lektorat: Lektormeister  
Einbandgestaltung: Pro ebookcovers Angie  
Druck und Bindung: Wydawnictwo Poligraf sp. zo.o. (Polen)

Taschenbuch: 978-3-96645-076-8  
Hardcover: 978-3-96645-077-5  
E-Book: 978-3-96645-075-1

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte (Übersetzung, Nachdruck und Vervielfältigung) vorbehalten. Kein Teil des Werks darf ohne schriftliche Genehmigung des Verlags in irgendeiner Form – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert, verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit größter Sorgfalt erstellt, ungeachtet dessen können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären.

# C# Kompendium

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>7</b>
1.1 Über den Autor.....	7
1.2 Warum C#? .....	8
1.3 Die Historie von C#.....	9
1.4 Der .NET-Framework, CIL, die CLI und die CLR.....	12
<b>2. Visual Studio: die professionelle Entwicklungsumgebung für C#</b>	<b>15</b>
2.1 Die Installation von Visual Studio .....	15
2.2 Ein Projekt in Visual Studio erstellen .....	19
<b>3. Das erste Programm in C#</b>	<b>26</b>
3.1 Der Aufbau einer Konsolenanwendung.....	26
3.2 Aufrufargumente verarbeiten.....	28
3.3 Kommentare erleichtern das Leben .....	30
3.4 Übungsaufgabe: Eine Erweiterung für „Hello World!“ .....	32
<b>4. Variablen und ihre Typen</b>	<b>35</b>
4.1 Variablen in der Programmierung: Fast so wie in der Mathematik .....	35
4.2 Variablen, Typen und Operatoren .....	36
4.3 Variablen in einem C#-Programm verwenden .....	46
4.4 Benutzereingaben in Variablen speichern.....	48
4.5 Konvertieren von Variablen .....	50
4.6 Übungsaufgabe: Rechnen mit Variablen.....	52
<b>5. Verzweigungen zur Steuerung des Programmablaufs</b>	<b>55</b>
5.1 Die einfache, bedingte Verzweigung .....	55
5.2 Vergleichsoperatoren und logische Operatoren .....	56
5.3 Geschachtelte bedingte Verzweigungen.....	59
5.4 Switch-Case: Die Mehrfachverzweigung .....	60
5.5 Die bedingte Zuweisung .....	67
5.6 Übungsaufgabe: Programmieren mit Verzweigungen.....	70
<b>6. Programmteile wiederholen mit Schleifen</b>	<b>76</b>
6.1 Die while-Schleife: Erst prüfen, dann arbeiten .....	76
6.2 Die do-while-Schleife: Erst arbeiten, dann prüfen.....	78
6.3 Die for-Schleife: Eine feste Anzahl von Wiederholungen .....	82
6.4 Die foreach-Schleife läuft über alles .....	84
6.5 Feintuning von Schleifen mit break und continue .....	87
6.6 Übungsaufgabe: Programmsteuerung mit Schleifen .....	93
<b>7. Strukturierte Daten in C#</b>	<b>101</b>
7.1 Arrays: Eine Variable für viele Werte.....	101
7.2 Mehrdimensionale Arrays.....	103
7.3 Typisierte Listen: Arrays mit Komfort.....	104
7.4 Das Dictionary, die elegante Listenverwaltung .....	107
7.5 Das Tuple: Mehrere verschiedene Variablen in einer Struktur.....	109
7.6 Übungsaufgaben: Arbeiten mit Arrays, Listen, Dictionaries und Tuples.....	111

<b>8. Methoden schaffen Ordnung</b>	<b>119</b>
8.1 Wozu benötigt man Methoden? .....	119
8.2 Einfache Methoden.....	119
8.3 Methoden mit Übergabeparametern.....	120
8.4 Methoden mit Rückgabewerten.....	123
8.5 Methoden überladen .....	128
8.6 Übungsaufgaben: Programmieren mit Methoden .....	129
<b>9. Grundlagen der Objektorientierten Programmierung</b>	<b>151</b>
9.1 Was ist Objektorientierte Programmierung? .....	152
9.2 Eine Klasse erstellen .....	155
9.3 Klassen mit Konstruktoren.....	158
9.4 Eine Instanz einer Klasse erzeugen und verwenden .....	160
9.5 Zugriffsmodifizierer für Klassen, Eigenschaften und Methoden.....	164
9.6 Properties statt Variablen verwenden .....	165
9.7 Operatoren überladen.....	174
9.8 Übungsaufgabe: Mit Objekten programmieren.....	178
<b>10. Objektorientierung für Fortgeschrittene</b>	<b>203</b>
10.1 Vererbung: abgeleitete Klassen .....	203
10.2 Das Interface: Eine definierte Schnittstelle .....	210
10.3 Der Garbage Collector: Eine automatische Speicherverwaltung .....	217
10.4 Polymorphie: Virtuelle Methoden und Properties .....	229
10.5 Abstrakte und versiegelte Klassen .....	240
10.6 Erweiterungsmethoden .....	243
10.7 Generische Klassen.....	247
10.8 Generische Methoden.....	272
10.9 Generische Properties.....	275
10.10 Methoden und Variablen: Delegaten verwischen die Unterschiede .....	277
10.11 Übungsaufgabe: Fortgeschrittene Programmierung mit Objekten .....	282
<b>11. Objekte verarbeiten mit „Linq to Objects“</b>	<b>296</b>
11.1 Was ist Linq? .....	296
11.2 Ein erstes Beispiel für eine Linq-Abfrage .....	297
11.3 Daten abfragen mit Linq .....	300
11.4 Daten konvertieren mit Linq.....	306
11.5 Linq-Methoden verketten .....	313
11.6 Verschachtelte Linq-Ausdrücke .....	317
11.7 Wann wird eine Linq Abfrage ausgeführt? .....	320
11.8 Parallelе Verarbeitung mit Linq.....	322
11.9 Übungsaufgabe: Linq verwenden.....	326
<b>12. Fehlerbehandlung mit Exceptions</b>	<b>334</b>
12.1 Was ist eine Exception?.....	334
12.2 Exceptions abfangen mit try – catch.....	336
12.3 Exceptions kontrolliert auslösen.....	340
12.4 Eigene Exceptions definieren .....	342
12.5 Übungsaufgabe: Ein Programm mit Fehlerbehandlung erstellen .....	346
<b>13. Visual Studio reloaded: Funktionalitäten für Fortgeschrittene</b>	<b>352</b>
13.1 Ein weiteres Projekt zur Projektmappe hinzufügen .....	352
13.2 Eine Klassenbibliothek erstellen .....	361
13.3 Die eigene Klassenbibliothek verwenden.....	364
13.4 Codenavigation.....	370

**Inhaltsverzeichnis**

13.5 Fehler finden mit dem Debugger .....	384
13.6 Übungsaufgabe: Eine eigene Klassenbibliothek erstellen .....	391
<b>14. Dateizugriff mit C#</b>	<b>400</b>
14.1 Textdateien lesen und schreiben .....	400
14.2 XML-Dateien verarbeiten .....	411
14.3 Objekte serialisieren und deserialisieren .....	416
14.4 Verzeichnisse erzeugen und durchsuchen .....	423
14.5 Übungsaufgaben: Arbeiten mit Dateien .....	426
<b>15. Datenbankzugriff mit dem Microsoft SQL-Server</b>	<b>435</b>
15.1 Microsoft SQL-Server installieren .....	435
15.2 Eine Datenbank erstellen .....	451
15.3 Den Entity Framework zum Projekt hinzufügen .....	453
15.4 Ein Entity-Modell erstellen .....	455
15.5 Lesen, ändern, hinzufügen und löschen von Daten .....	470
15.6 Übungsaufgaben: Programmierung mit Datenbanken .....	487
<b>16. ASP.NET MVC – Anwendungen fürs Web</b>	<b>502</b>
16.1 Eine neue ASP.NET MVC-Anwendung erstellen .....	502
16.2 Einen Controller und eine View hinzufügen .....	519
16.3 Daten mit dem Controller bereitstellen .....	525
16.4 Daten mit der View anzeigen .....	531
16.5 Eine eigene View, um Daten zu ändern .....	533
16.6 Übungsaufgabe: Die Webanwendung erweitern .....	539
<b>17. Grafische Benutzeroberflächen mit WPF</b>	<b>548</b>
17.1 Eine neue WPF-Anwendung erstellen .....	548
17.2 Was ist XAML? .....	552
17.3 Wir programmieren einen einfachen Textdateibetrachter .....	556
17.4 Organisation der Benutzeroberfläche mit dem Grid-Steuerelement .....	563
17.5 Auswahl einer Datei mit einem OpenFileDialog-Objekt .....	567
17.6 Auflistung der Dateien mit einem ListBox-Steuerelement .....	570
17.7 Styling mit Ressourcen .....	577
17.8 Das MVVM-Entwurfsmuster .....	587
17.9 Übungsaufgabe: Eine WPF-Anwendung auf MVVM umstellen .....	596
<b>18. Testautomatisierung mit MS-Test</b>	<b>608</b>
18.1 Ein Testprojekt anlegen .....	608
18.2 Erstellen eines Komponententests .....	610
18.3 Verwendung des Test-Explorers .....	613
18.4 Abhängigkeiten reduzieren mit Mocks .....	614
18.5 Übungsaufgabe: Komponententests unter Verwendung von Mocks .....	620
<b>19. Schlusswort</b>	<b>627</b>
<b>20. Glossar</b>	<b>628</b>
<b>21. Index</b>	<b>633</b>

# Kapitel 1

## Einleitung

Dieses Buch beschäftigt sich mit der Programmiersprachsprache C# und dem .NET-Framework. Zudem werden die folgenden Programmierschwerpunkte umfassend behandelt:

- ▶ Erstellen von Konsolen-Apps
- ▶ Objekte verarbeiten mit Linq to Objects
- ▶ Fehlerbehandlung mit Exceptions
- ▶ Dateizugriff mit C#

Wie der Titel schon sagt, ist der Hauptschwerpunkt dieses Buchs die Programmiersprache C# selbst. Diese möchte ich möglichst umfassend mit all ihren Facetten behandeln. Des Weiteren möchte ich Ihnen noch ein paar Programmierthemen vorstellen, die für die Programmierung mit C# und dem .NET-Framework von großer Bedeutung sind. Allerdings sind diese Themen so umfangreich, dass eine vollständige Behandlung jedes einzelnen Aspekts mit all seinen Details ein eigenes Buch rechtfertigen würde. Daher werde ich die folgenden Themen nur jeweils mit einem eigenen Kapitel anreißen.

- ▶ Datenbankzugriff am Beispiel des Microsoft SQL-Servers
- ▶ ASP.NET MVC-Anwendungen fürs Web
- ▶ Grafische Benutzeroberflächen mit WPF
- ▶ Testautomatisierung mit MS-Test

### 1.1 Über den Autor

Der Autor Robert Schiefele hat 1981 im Alter von 15 Jahren damit begonnen, die Programmierung im Selbststudium zu erlernen. Die ersten Schritte machte er mit der Programmiersprache Basic, welche damals noch in der Form eines Kommandozeilen-Interpreters vorlag. Auch die Grundlagen der Assembler-Programmierung brachte er sich als Schüler selbst bei. Während seines Informatik-Studiums erlernte er dann die Programmiersprache Pascal. In studentischen Nebentätigkeiten kam er mit Turbo Pascal zum ersten Mal mit der Objektorientierten Programmierung in Berührung. Während dieser Nebentätigkeiten beschäftigte er sich mit Hilfe von Microsoft Access und dem Microsoft SQL-Server auch das erste Mal mit relationalen Datenbanken. Nach dem Studium erweiterte er seine Programmierkenntnisse als festangestellter

## 1 Einleitung

Datenbankentwickler um Microsoft Visual Basic und die Programmierung von Oracle-Datenbanken. Später nahm er eine Position als Berater für DevOps, was zu dieser Zeit noch ALS hieß, an. Dort lernte er auch die Programmiersprache Java. Ab der Jahrtausendwende arbeitete er als Berater für DevOps und Business Process Management. Während dieser Tätigkeit lernte er auch den Microsoft .NET Framework kennen. Zunächst arbeitete er mit VB.NET, stieg dann aber wegen der syntaktischen Ähnlichkeit mit Java auf die Programmiersprache C# um, welche er seitdem als seine bevorzugte Programmiersprache betrachtet.

Seit November 2016 arbeitet er als freiberuflicher Software-Entwickler ausschließlich in der .NET Programmierung – sowohl im Bereich der Web-Applikationen mit ASP.NET MVC als auch im Bereich der Desktopapplikationen mit WPF. Im Juli 2018 gründete er die Firma Contigo UG & Co. IT-Consulting KG, welche sich die Software-Entwicklung mit .NET auf die Fahnen geschrieben hat.

Privat lebt der Autor im Großraum München, ist verheiratet, hat zwei Kinder und beschäftigt sich in seiner Freizeit am liebsten mit seinem Garten.

### 1.2 Warum C#?

Wikipedia bezeichnet C# als eine streng typisierte, objektorientierte Allzweck-Programmiersprache. Diese Bezeichnung liefert uns schon mal drei gute Gründe, warum Sie mit C#-Kenntnissen bestens gerüstet sind, egal ob Sie in Ihrer Freizeit oder in Ihrem Beruf in Teil- oder in Vollzeit Software entwickeln.

Streng typisiert bedeutet, alle Konstrukte der Programmiersprache, mit denen man in irgendeiner Form Daten verarbeiten kann, haben einen fest zugeordneten Typ. Das heißt vereinfacht ausgedrückt: Ein Text ist ein Text und keine Zahl und zu einem Text können Sie auch nicht die Zahl 5 addieren. Strenge Typisierung ist ein wichtiges Hilfsmittel, wenn es darum geht, den sogenannten „Spaghetti-Code“ zu vermeiden. „Spaghetti-Code“ ist ein Ausdruck, der sagen möchte, dass ein Programm logisch so durcheinander ist wie ein Teller Spaghetti. Wenn Sie an solchen Programmen Veränderungen, Korrekturen oder Erweiterungen vornehmen müssen, ist das eine sehr mühsame und zeitraubende Angelegenheit.

Der zweite gute Grund: C# ist objektorientiert. Heutzutage kann man die Objektorientierung als State of the Art in der Programmierung bezeichnen. Sämtliche Programmiersprachen, die heute von Bedeutung sind, wie C++, Java, Python, TypeScript, JavaScript, VB.NET und natürlich C#, sind objektorientiert. Gerade wenn Sie sich beruflich mit der Programmierung beschäftigen wollen oder müssen, kommen Sie um dieses Konzept nicht herum. Die Objektorientierung ist zur Vermeidung von Spaghetti-Codes das wichtigste Hilfsmittel. Wie Objektorientierung genau funktioniert, werden wir in diesem Buch noch ausführlich erörtern.

Der dritte gute Grund: C# ist eine Allzweck-Programmiersprache. Mit C# können Sie alles Mögliche programmieren. Folgende Arten von Programmen können mit C# realisiert werden:

- ▶ Konsolen-Apps für Windows, Linux und MacOS
- ▶ Apps mit grafischer Benutzeroberfläche für Windows, Android und iOS
- ▶ Web-Applikationen
- ▶ Dienste, die im Hintergrund laufen für Windows und Linux
- ▶ Spiele für Windows, Android, iOS, Xbox, PS4, Nintendo Switch, Steam und Web

Ein weiterer Grund für C# hat in den letzten Jahren zunehmend an Bedeutung gewonnen. C# ist unabhängig von Plattformen. Damit ein C#-Programm auf einem Computer laufen kann, benötigt es lediglich ein .NET-Framework, das für das jeweilige Betriebssystem dieses Computers geschrieben wurde. Für die Betriebssysteme Windows, Linux, Android, MacOS und iOS gibt es bereits angepasste Versionen des .NET-Frameworks. Mit C# lassen sich auch Programme schreiben, die auf kleinen, günstigen Einplatinencomputern wie dem Raspberry Pi laufen. Das ist möglich, da es für den Raspberry Pi speziell angepasste Linux- und Windows-Versionen gibt. Allerdings gibt es für Linux-Betriebssysteme eine Einschränkung für den Einsatz von C#: Desktop-Programme, die eine grafische Benutzeroberfläche verwenden, können zum Zeitpunkt der Erstellung dieses Buches noch nicht für Linux in C# erstellt werden, da noch kein .NET-Framework für Linux zur Verfügung steht, das die grafischen Fähigkeiten von Linux unterstützt. Eventuell ändert sich das im November 2021, wenn die Version 6.0 des .NET-Framework erscheint.

### 1.3 Die Historie von C#

Die Geschichte von C# ist sehr eng mit der Geschichte der konkurrierenden Programmiersprache Java verknüpft. Als die Plattform-unabhängige Programmiersprache Java aus dem Hause Sun Microsystems in den 90er Jahren des letzten Jahrtausends ihren Siegeszug antrat, erkannte auch Microsoft das Potential der neuen Programmiersprache und erwarb 1995 eine Lizenz für Java von Sun Microsystems und integrierte Java in seine Entwicklungsumgebung Developer Studio unter dem Namen J++. Als Microsoft aber spezielle Erweiterungen, die nur unter Windows lauffähig waren, auf den Markt brachte, sah Sun Microsystems die Plattform-Unabhängigkeit von Java gefährdet und intervenierte 1998 mit einem Gerichtsverfahren, das zugunsten von Sun Microsystems ausging. Aufgrund dieses Verfahrens musste Microsoft sämtliche Java-Technologien in seinen Produkten kompatibel zu Sun-Java halten. Microsoft brachte 1999 eine neue Version von J++ heraus, die alle gerichtlichen Auflagen erfüllte. Im Jahr 2000 gab Microsoft bekannt, dass es seine Java-Lizenz die 2001 auslief, nicht mehr verlängern wird und J++ in der kommen Version seiner Entwicklungsum-

## 1 Einleitung

gebung, die künftig unter dem Namen Visual Studio erscheinen wird, nicht mehr enthalten sein wird. Microsoft entwickelte dann seine Java-artige Programmiersprache für Visual Studio von Grund auf neu.

Dieses Projekt bekam den Codenamen Cool und wurde zur Markteinführung in C# (gesprochen C-Sharp) umbenannt. Sharp ist die englische Bezeichnung des Doppelkreuz-Zeichens. Das Doppelkreuz hat verschiedene symbolische Bedeutungen.

Zum einen wird es in der Notenschrift der Musik nach einer Note geschrieben, um anzudeuten, dass die Note um einen Halbton erhöht ist. Somit drückt der Name aus, dass C# eine erhöhte Variante der Programmiersprache C ist, von der es einige grundlegende syntaktische Regeln geerbt hat.

Zum anderen kann das Doppelkreuz auch als Kombination von vier +-Zeichen, die in einem Quadrat angeordnet sind, interpretiert werden. Damit würde der Name C# bedeuten, dass man dem Namen C++ noch zwei weitere +-Zeichen hinzugefügt hat und C# somit eine Weiterentwicklung von C++ ist.

C# wurde ursprünglich von Anders Hejlsberg im Auftrag von Microsoft entwickelt. Als es im Jahre 2002 mit Visual Studio 2002 und dem .NET-Framework 1.0 als C# 1.0 am Markt erschien, wurde der neuen Programmiersprache keine große Zukunft vorgehersagt.

„Es ist wie Java, kostet aber Geld, ist von Microsoft und läuft nur unter Windows.“ Dieser bissige Spruch kursierte damals unter Programmierern und entsprach zu diesem Zeitpunkt auch der Wahrheit. Aber das sollte sich im Laufe der Zeit Stück für Stück ändern.

Eine 2003 hastig nachgeschobene Version 1.1 des .NET-Frameworks mit einem Visual Studio 2003 brachte keine Verbesserung der Sprache C# an sich und auch nur eine geringe Verbesserung der Akzeptanz des Gesamtsystems.

Im Jahre 2005 erschien Visual Studio 2005 mit dem .NET-Framework 2.0 und mit C# in der Version 2.0. Diese Version zog nicht nur mit neuen, kurz zuvor releaseden Sprachfeatures von Java gleich, sondern konnte auch noch ein paar zusätzliche Features bieten, die für Java bis heute nicht zur Verfügung stehen. Als Microsoft 2006 auch noch die kostenlosen Expressvarianten von Visual Studio 2005 veröffentlichte, konnten sich - bis auf die Plattform-Unabhängigkeit - C# und Java zum ersten Mal auf Augenhöhe begegnen.

Ebenfalls 2006 erschien die Version 3.0 des .NET-Frameworks, welche zwar weitere .NET Programmiermodelle brachte, aber keine Neuerungen in der Sprache C#.

2008 erschien Visual Studio 2008 und brachte den .NET-Framework 3.5 und C#3.0 mit. Das brachte weitere neue Sprachfeatures, die die Objektabfrage-Bibliothek Linq ermöglichten. Linq erfreut sich bis heute bei C#-Programmierern größter Beliebtheit. In diesem Buch habe ich Linq ein eigenes Kapitel gewidmet. Java-Entwickler mussten auf eine mit Linq vergleichbare Technik noch bis zum Erscheinen von Java 8 im Jahr 2014 warten. Ab C# 3.0 war C# als Programmiersprache bereits das mit Abstand bessere Java. Lediglich bei der Plattform-Unabhängigkeit konnte Java noch punkten.

Visual Studio 2010 brachte das .NET Framework 4.0 und C# 4.0. Neben ein paar kleineren Erweiterungen der Sprache C# konzentrierten sich die Microsoftentwickler hauptsächlich auf die Verbesserung von WPF, einer Technologie zum Erstellen grafischer Benutzeroberflächen.

Mit Visual Studio 2012 erhöhte sich die .NET-Framework Version auf 4.5 und die C#-Version auf 5.0. In C# kamen die sogenannten Async Features hinzu, die den Programmierer bei der Programmierung asynchroner Abläufe unterstützten. Diese Technik wird für Programme verwendet, die mehrere Dinge gleichzeitig tun.

Visual Studio 2013 erhöhte lediglich die Version des .NET-Frameworks auf 4.5.1, brachte aber signifikante Verbesserungen im Bereich Application Life Cycle Management.

Visual Studio 2015 brachte den .NET-Framework 4.6, die C#-Version 6.0 und zusätzliche Erweiterungen von C# und weitere Verbesserungen im Bereich Application Life Cycle Management. Zudem wurde für Visual Studio 2015 auch .NET Core 1.0 veröffentlicht. Das .NET Framework war von Anfang an auf Plattform-Unabhängigkeit ausgelegt, allerdings hat sich Microsoft vorerst nicht darum gekümmert, einen .NET-Framework für andere Plattformen außer Windows zur Verfügung zu stellen. Mit .NET Core sollte sich das nun ändern. Mit .NET Core 1.0 konnten in C# Konsolen-Apps und Webapplikationen erstellt werden, die auch auf Linux-basierten Computern liefen.

Die Visual Studio Version 2017 kam mit .NET Core 2.0 und der C#-Version 7.0. Neben weiteren Verbesserungen der Programmiersprache C# und der Bereinigung einiger Kinderkrankheiten von .NET Core brachte Visual Studio 2017 vor allem Xamarin-Tools. Nachdem Microsoft den Hersteller Xamarin übernommen hatte, integrierte es dessen Tools in Visual Studio. Mit den Xamarin-Tools konnte man nun Applikationen für Android und iOS mit Visual Studio entwickeln.

Die beim Erstellen dieses Buchs aktuelle Version Visual Studio 2019 umfasst .NET Core 3.1 und C# 8.0. Auch hier wurde C# wieder verbessert. Aber die wichtigste Neuerung war .NET Core 3.1, das es ermöglicht, grafische Desktop-Applikationen als .NET Core-Applikationen zu entwickeln. Desktop-Applikationen für .NET Core zeichnen sich durch ein schnelleres Laufzeitverhalten aus und können als eine einzige ausführbare

## 1 Einleitung

Datei an den Benutzer geliefert werden, ohne dass dieser ein installiertes .NET-Framework benötigt.

Noch während der Entwicklung von .NET Core bis zur aktuellen Version 3.1 wurde parallel der normale .NET-Framework bis zur Version 4.8 weiterentwickelt.

Im November 2020 vereinte Microsoft die beiden Frameworks und brachte .NET 5.0 nur noch ein einziges aktuelle Framework auf den Markt, das für Windows, Linux, Android, MacOS und iOS zur Verfügung steht. Allerdings werden grafische Desktop-Anwendung für andere Plattformen als Windows immer noch nicht unterstützt.

Eventuell gibt es diese Unterstützung im November 2021, wenn .NET 6.0 erscheint.

### 1.4 Der .NET-Framework, CIL, die CLI und die CLR

Früher unterschied man Programmiersprachen grundsätzlich in zwei verschiedene Arten: Compiler-Sprachen und Interpreter-Sprachen. Bei beiden schreibt man das Programm in der jeweiligen höheren Programmiersprache. Bei Compiler-Sprachen wird das Programm mit einem sogenannten Compiler in die Maschinensprache des Computers umgesetzt. Und dieses kompilierte Maschinenprogramm kann dann auf dem Zielcomputer ausgeführt werden. Bei Interpreter-Sprachen gibt es auf dem Zielrechner ein sogenanntes Interpreter-Programm, das das Programm, ohne es vorher umzusetzen, direkt in der höheren Programmiersprache abarbeitet. Der Vorteil von Compiler-Sprachen ist, dass die kompilierten Programme wesentlich schneller laufen, da der Compiler bei der Erstellung des Maschinencodes zahlreiche Optimierungen vornehmen kann. Der Vorteil von Interpreter-Sprachen ist, dass sie plattformunabhängig sind. Es genügt ein Interpreter-Programm für die jeweilige Zielpfaltform zu entwickeln und schon sind alle Programme in der zugehörigen Hochsprache auf dieser Zielpfaltform lauffähig.

Bei der Entwicklung des .NET-Frameworks ist Microsoft einen Mittelweg gegangen und hat das Beste aus beiden Welten vereinigt. Daher gibt es einen Compiler, der C# oder eine andere .NET-Sprache in einen sogenannten CIL-Code umsetzt. CIL steht für Common Intermediate Language. Den CIL-Code muss man sich als Maschinencode für einen virtuellen (gedachten) Computer vorstellen, den es als echte physische Maschine gar nicht gibt. Wenn man jetzt auf einem echten physischen Computer so eine Art Simulationsprogramm hätte, das diesen virtuellen Computer simuliert, könnte dieses Simulationsprogramm ein CIL-Programm ausführen. Das klingt zwar etwas kompliziert und umständlich, hat aber den Charme, dass ein Programm nur einmal entwickelt und in ein CIL-Programm umgesetzt werden muss, und dann auf allen Plattformen lauffähig ist, für die es so ein Simulationsprogramm gibt. Bis jetzt haben wir mit unserem neuen Verfahren lediglich die Vorteile eines Interpreters erreicht und haben uns zusätzlich den Nachteil eingefangen, dass wir

unser Programm kompilieren müssen, was bei herkömmlichen Interpretern ja entfallen könnte. Aber durch das Kompilieren erhalten wir auch Vorteile. Ein CIL-Code hat Ähnlichkeit mit einem echten Maschinencode und da er mit einem Compiler erzeugt wird, kann dieser Compiler wieder Optimierungen einbauen. Das heißt, unser CIL-Code ist zwar langsamer als ein echter Maschinencode, aber schneller als interpretierter Code und genauso Plattform-unabhängig wie der interpretierte Code. Zudem hat sich Microsoft einen kleinen Trick einfallen lassen und das ist der JIT-Compiler oder einfach nur JIT. Das steht für Just in Time-Compiler. Der JIT kompiliert den CIL-Code in den Maschinencode der Zielpлатzform. Damit ist ein Simulationsprogramm, das CIL-Codes ausführt, überflüssig. Die Laufzeit-Umgebung für den CIL-Code, die auch den jeweiligen JIT enthält, wird allgemein als CLI (Common Language Infrastructure) bezeichnet. Eine konkrete Implementierung für eine bestimmte Zielpunktform dagegen wird als CLR (Common Language Runtime) bezeichnet. In anderer Literatur oder in manchen Internetbeiträgen wird die CIL manchmal auch als MSIL bezeichnet, das steht für Microsoft Intermediate Language, bezeichnet aber dieselbe Sache. Mit der folgenden Grafik möchte ich den Weg von einem in C# geschriebenen Programm zu einem ausführbaren Computerprogramm noch einmal veranschaulichen.

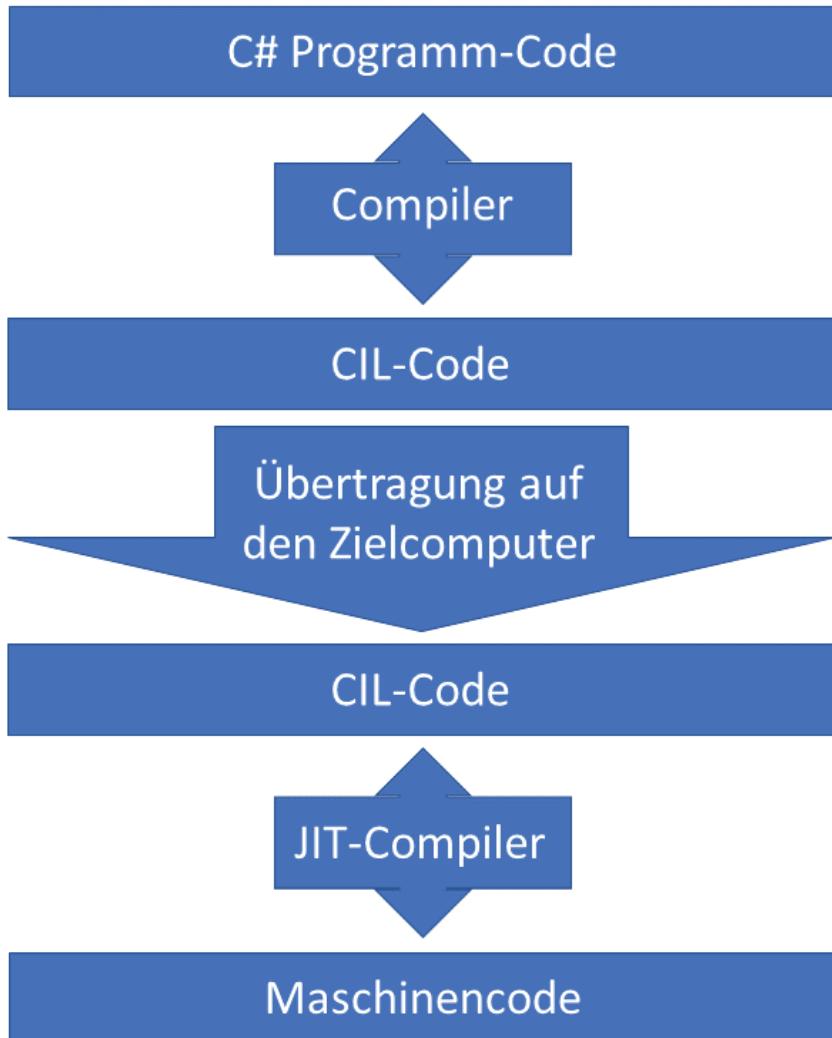


Abb.1.3.1 Kompilierprozess des .NET-Frameworks

Der Entwickler schreibt ein Programm in C#, kompiliert es auf seinem Computer in CIL Code. Er kann es dort auch ausführen und testen. Wenn das Programm in Ordnung ist, wird es auf den Zielcomputer beziehungsweise den Computer des Benutzers kopiert. Auf dem Computer des Benutzers befindet sich ein installiertes .NET-Framework und damit eine CLR. Die CLR enthält einen JIT-Compiler, der den CIL-Code in Maschinencodes kompiliert, der dann auf dem Zielcomputer ausgeführt wird.

## Kapitel 2

# Visual Studio: die professionelle Entwicklungsumgebung für C#

2

Eine Entwicklungsumgebung ist ein Softwarepaket, das uns hilft, möglichst komfortabel Software zu entwickeln. Eine Entwicklungsumgebung enthält einen Editor, mit dem ein Programmcode geschrieben werden kann und der vor allem auf die verwendete Programmiersprache zugeschnitten ist. Ein guter Editor sollte „Source Code Highlighting“ und „Intellisense“ beherrschen. „Source Code Highlighting“ bedeutet, dass der Editor verschiedene Komponenten der jeweiligen Programmiersprache verschiedenfarbig darstellen kann, was die Lesbarkeit enorm erhöht. „Intellisense“ ist eine Technologie, die Microsoft bereits in den 90er Jahren erfunden hat. Java-Programmierer, die die Entwicklungsumgebung Eclipse verwenden, kennen „Intellisense“ bereits unter dem Namen „Code Complete“. Ein Editor, der diese Technologie beherrscht, macht dem Programmierer Vorschläge, wie sein Programmcode weitergehen könnte, statt weiter zu tippen, kann der Programmierer aus einer Liste auswählen. Des Weiteren enthält eine Entwicklungsumgebung einen oder mehrere Compiler und verschiedene Projektvorlagen. Je nach Projekt, das Sie verwirklichen wollen, müssen Sie das Verzeichnis, in dem Sie Ihre Projektdateien anlegen, anders strukturieren. Eine geeignete Projektvorlage erledigt das automatisch für Sie.

Die professionelle Entwicklungsumgebung für die C#-Programmierung ist Visual Studio von Microsoft. Derzeit liegt sie in der Version Visual Studio 2019 vor. Visual Studio gibt es in verschiedenen Editionen, die je nach Leistungsumfang auch sehr stark im Preis variieren. Zudem gibt es eine kostenlose Community Edition. Die Community Edition dürfen Sie generell zur Entwicklung von Software, mit der Sie kein Geld verdienen, verwenden. Für kommerzielle Zwecke darf Sie nur verwendet werden, wenn im Unternehmen nicht mehr als 5 Personen Visual Studio nutzen und das Unternehmen nicht mehr als 1.000.000\$ umsetzt und nicht mehr als 250 PCs betreibt. Das heißt, wenn Sie sich im Eigenstudium zu Hause C# beibringen wollen, können Sie auf jeden Fall die kostenlose Community Edition verwenden.

### 2.1 Die Installation von Visual Studio

Um Visual Studio 2019 Community Edition zu installieren, öffnen Sie einen Web-Browser und geben folgende URL ein:

<https://visualstudio.microsoft.com/de/>

## 2 Visual Studio: die professionelle Entwicklungsumgebung für C#



Abb. 2.1.1 Download Seite für Visual Studio

Klicken Sie auf „Visual Studio herunterladen/Community 2019“ und der Download des Installationsprogramms erscheint.

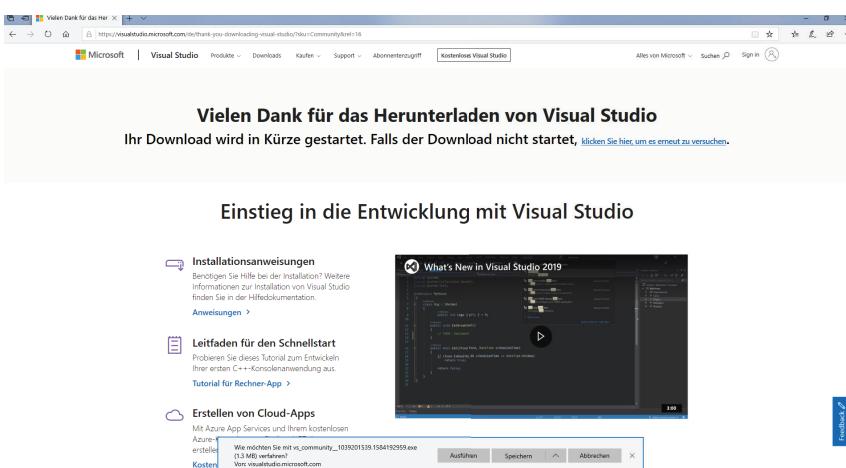
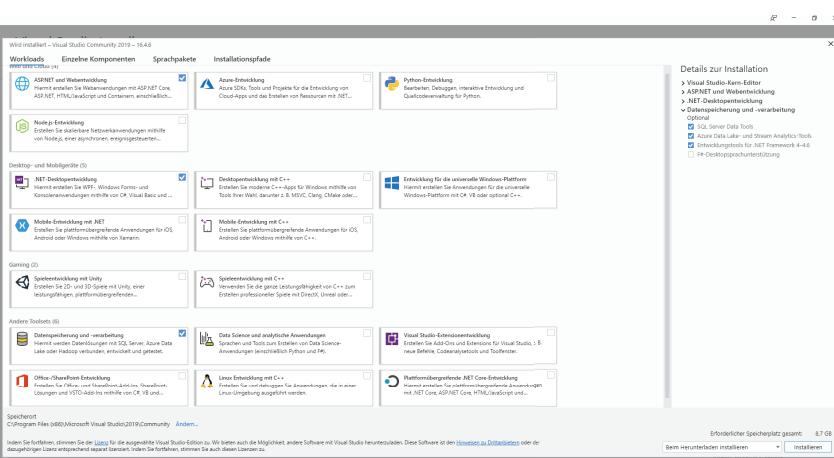


Abb. 2.1.2 Download des Visual Studio Installers

Je nachdem, welchen Browser Sie mit welchen Einstellungen verwenden, bietet Ihr Browser Ihnen das direkte Starten des Installationsprogramms an oder nicht. Wenn möglich, klicken Sie auf ausführen, wenn nicht, laden Sie die Installationsdatei herunter und starten Sie das Programm.

Windows fragt Sie nun, ob Sie dieses Programm wirklich ausführen wollen. Bestätigen Sie dies mit einem Klick auf die Schaltfläche „Ja“. Beim Begrüßungsdialog klicken Sie auf die Schaltfläche „fortfahren“. Danach wird das Visual Studio Installationsprogramm installiert und gestartet.



**Abb. 2.1.3** Auswahl der Pakete für die Installation von Visual Studio

Für die Beispielprogramme dieses Buches benötigen Sie folgende Pakete:

- ▶ ASP.NET und Webentwicklung
- ▶ .NET Desktopentwicklung
- ▶ Datenspeicherung und -verarbeitung

Es steht Ihnen natürlich frei, weitere Pakete, die Sie interessieren, zu installieren. Um den Installationsprozess zu starten, klicken Sie auf die Schaltfläche „Installieren“ rechts unten. Die Installation beginnt. Je nach Anzahl der ausgewählten Pakete und je nach Bandbreite Ihrer Internetverbindung kann die Installation zwischen ein paar Minuten und zwei Stunden dauern. Nach der Installation startet Visual Studio mit dem ersten Begrüßungsdialog.



Abb. 2.1.4 Visual Studio Anmeldung

Wenn Sie bereits über ein Visual Studio Konto verfügen, klicken Sie auf die Schaltfläche „Anmelden“. Wenn Sie noch nicht über ein Visual Studio Konto verfügen, können Sie ein Konto erstellen, in dem Sie auf „Erstellen Sie ein Konto!“ klicken. Falls Sie kein Konto erstellen wollen, klicken Sie auf: „Jetzt nicht, vielleicht später“. Nach der Anmeldung erscheint der Standard-Begrüßungsdialog von Visual Studio. Hier können Sie ein neues Projekt erstellen, was wir dann im nächsten Kapitel tun werden.

## 2.2 Ein Projekt in Visual Studio erstellen

Nachdem wir die Visual Studio Community Edition erfolgreich installiert haben, wollen wir uns ansehen, wie man in Visual Studio ein Projekt erstellt, bevor wir unser erstes Programm in C# schreiben.

2

Starten Sie Visual Studio. Das Startfenster von Visual Studio erscheint am Bildschirm.

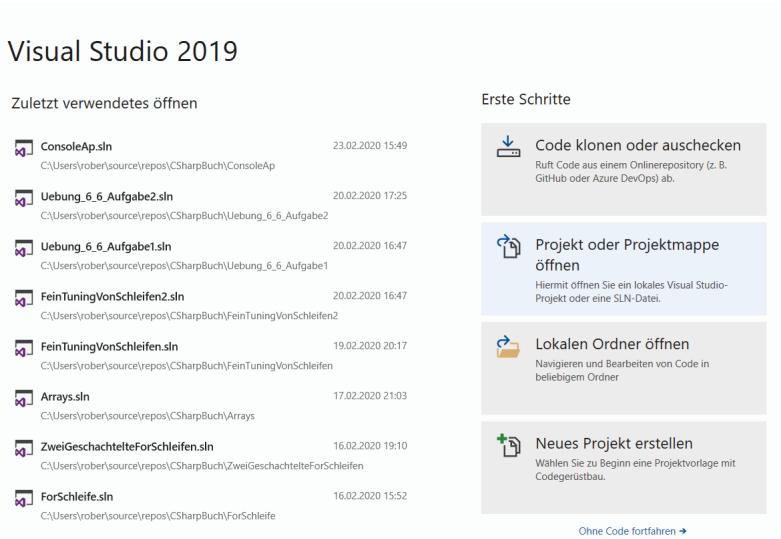


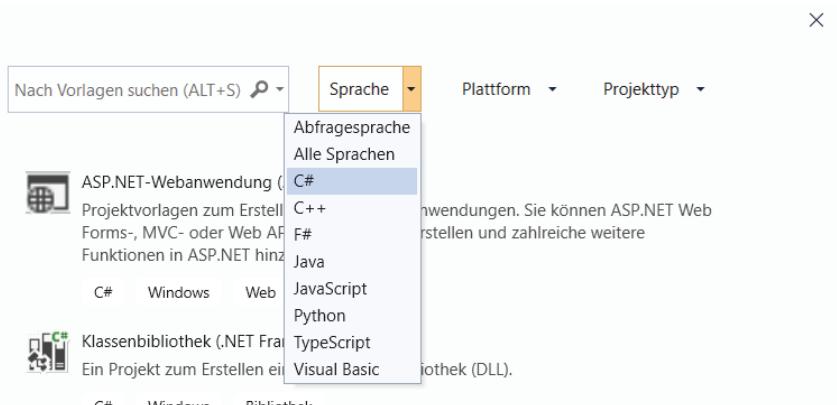
Abb. 2.2.1 Der Willkommensdialog von Visual Studio

Klicken Sie auf die Schaltfläche „Neues Projekt erstellen“.

## 2 Visual Studio: die professionelle Entwicklungsumgebung für C#

**Abb. 2.2.2** Ein neues Projekt erstellen

Auf der linken Seite sehen Sie die die von Ihnen zuletzt verwendeten Projektvorlagen und auf der rechten Seite eine lange Liste von allen Projektvorlagen in Visual Studio.

**Abb. 2.2.3** Die Auswahl der Programmiersprache

In der Dropdownbox-Sprache können Sie eine Programmiersprache auswählen, um nur Projektvorlagen für diese Programmiersprache anzuzeigen. Hier wählen Sie natürlich C# aus.

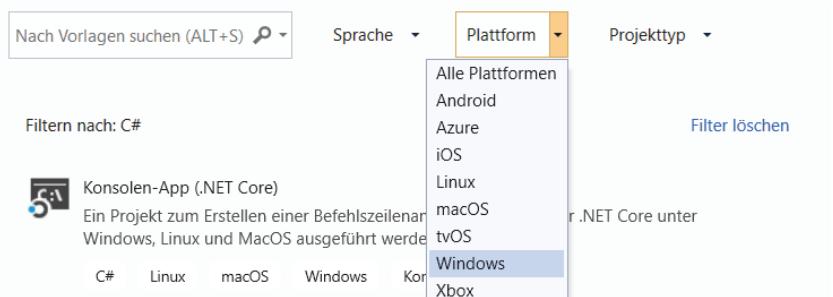


Abb. 2.2.4 Die Auswahl der Plattform

In der Dropdownbox Plattform können Sie eine Plattform beziehungsweise ein Betriebssystem auswählen, für das Sie ein Projekt erstellen wollen. Jetzt sehen wir nur noch Projektvorlagen für Windows und C# in der Liste.

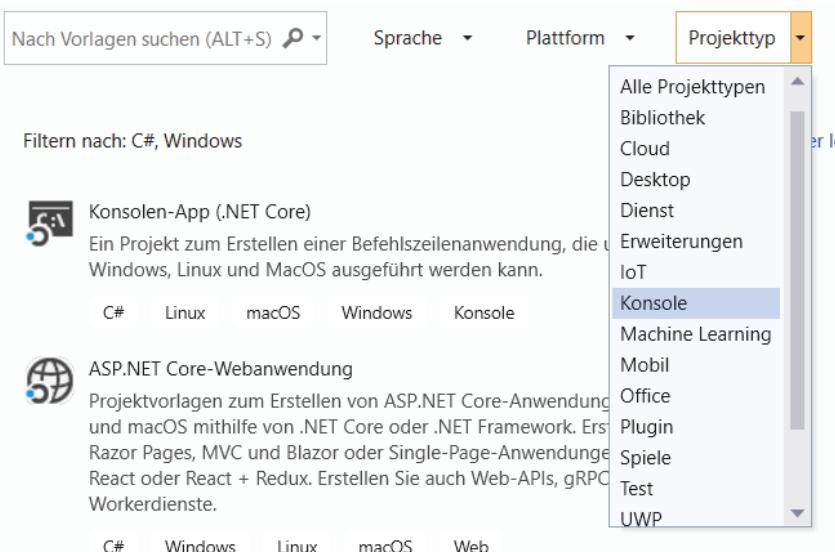
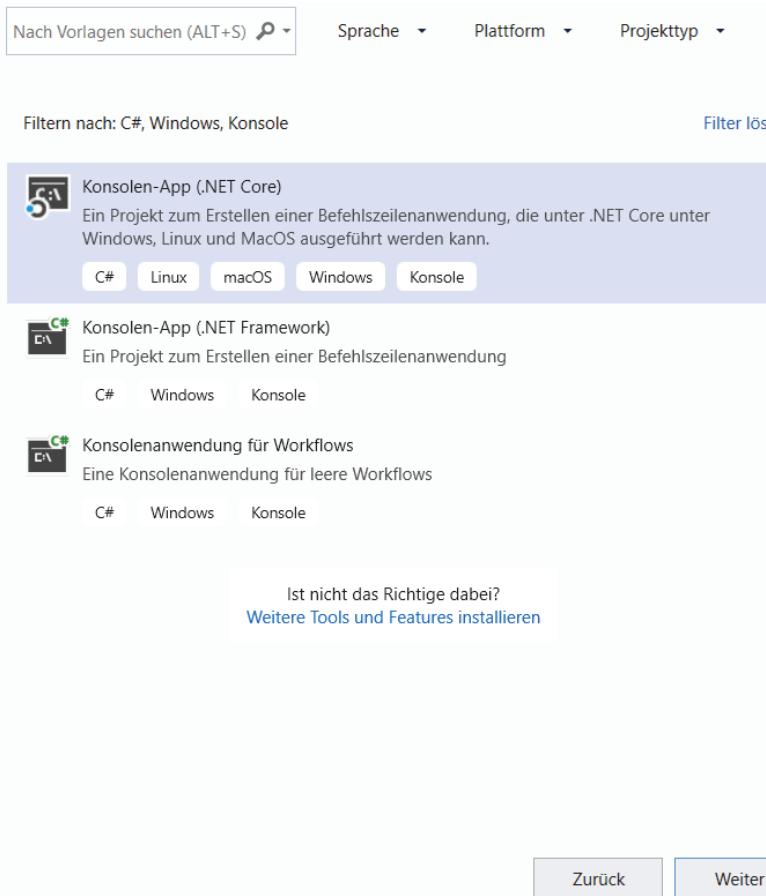


Abb. 2.2.5 Die Auswahl des Projekttyps

## 2 Visual Studio: die professionelle Entwicklungsumgebung für C#

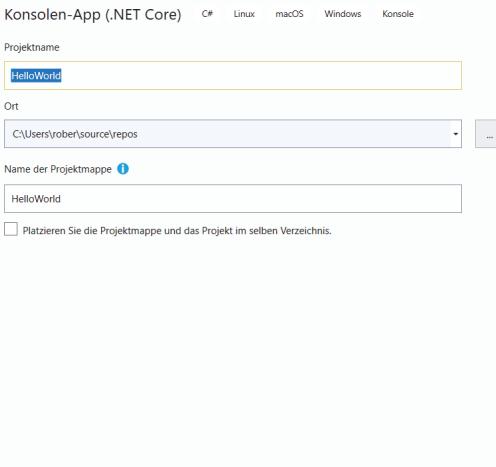
In der Dropdownbox Projekttyp können Sie die Projektart für das zu erstellen Projekt auswählen. Wählen Sie hier Konsole aus. Jetzt sind nur noch Projektvorlagen für Konsoleanwendungen in C# unter Windows in der Liste.



**Abb. 2.2.6** Die Auswahl der Projektvorlage

Wählen Sie Konsolen-App (.NET Core) aus und klicken Sie auf die Schaltfläche „Weiter“ und das Fenster „Neues Projekt konfigurieren“ erscheint am Bildschirm.

## Neues Projekt konfigurieren



**Abb. 2.2.7** Das neue Projekt konfigurieren

Hier können Sie den Namen des Projekts festlegen. In unserem Fall heißt es „Hello-World“. Im Feld Ort können Sie einen Speicherort für Ihr Projekt festlegen. Für den Namen der Projektmappe verwendet Visual Studio standardmäßig denselben Namen wie für das Projekt. Sie könnten ihn zwar ändern, aber wir lassen ihn vorerst, wie er ist. Die Checkbox „Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis“ lassen wir auch leer. Auf das Thema Projektmappen werde ich in einem späteren Kapitel eingehen. Wenn Sie jetzt auf die Schaltfläche erstellen klicken, erzeugt Visual Studio ein Konsolen-App-Projekt für C# für die Plattform .NET Core. Das heißt, das zu erstellende Programm ist unter Windows, Linux und MacOS lauffähig.

## 2 Visual Studio: die professionelle Entwicklungsumgebung für C#

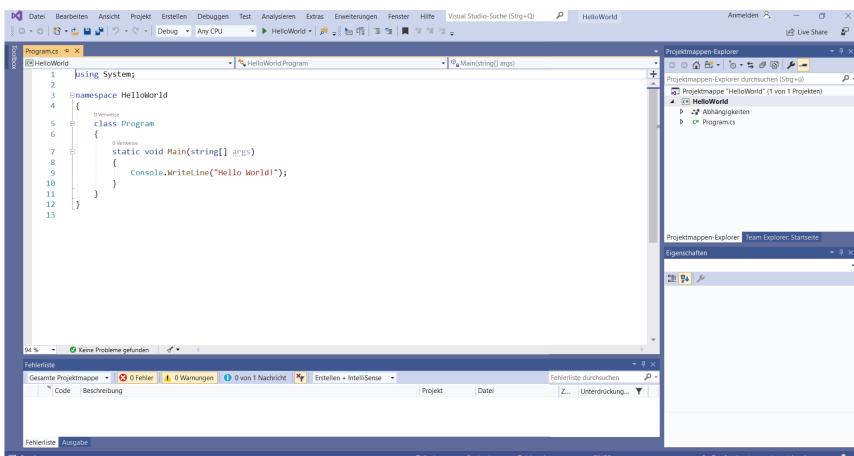
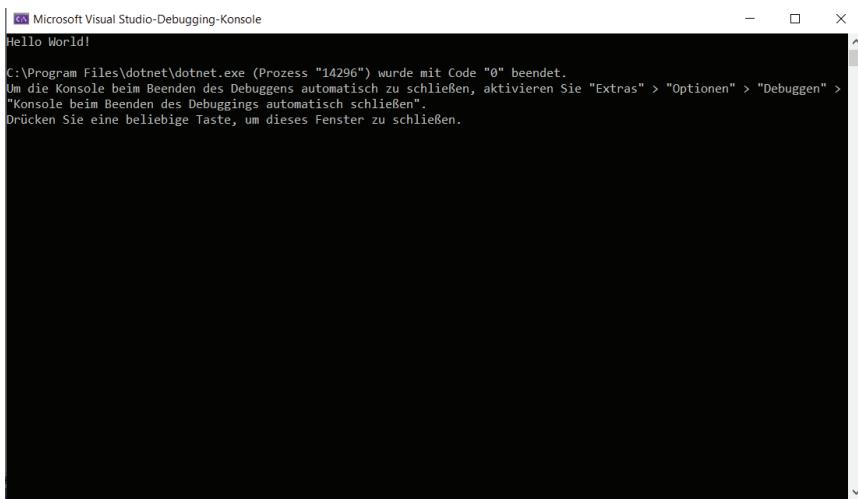


Abb. 2.2.8 Ein neu erstelltes Projekt

Im Hauptfenster sehen Sie ein von Visual Studio erzeugtes Beispielprogramm und rechts oben sehen Sie den sog. „Projektmappen-Explorer“. Er stellt die Projektmappe und die darin enthaltenen Projekte dar. Eine Projektmappe ist eine Sammlung von zusammengehörigen Projekten, die man als Programmierer gemeinsam bearbeiten möchte. Unsere erste Projektmappe heißt „HelloWorld“ und das einzige darin enthaltene „Projekt“ heißt auch „HelloWorld“. In ernsthaften Softwareprojekten hat man üblicherweise mehr als ein Projekt in einer Projektmappe. In diesem Buch kommen wir vorerst mit einem Projekt in unserer Projektmappe aus. Im Kapitel: „Visual Studio reloaded: Funktionalitäten für Fortgeschrittene“ werden wir sehen, wann es hilfreich ist, mehr als ein Projekt in einer Projektmappe zu haben.

Jetzt müssen Sie das Programm nur noch starten. Klicken Sie dazu auf die Schaltfläche mit dem grünen Dreieck in der Menüleiste. Es erscheint ein Ausgabefenster mit dem Text „Hello World!“.

## 2.2 Ein Projekt in Visual Studio erstellen



The screenshot shows a Microsoft Visual Studio Debugging Console window titled "Microsoft Visual Studio-Debugging-Konsole". The console displays the output of a "Hello World!" application. The text in the window reads:

```
Hello World!
C:\Program Files\dotnet\dotnet.exe (Prozess "14296") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

2

Abb. 2.2.9 Die Konsolen-App "Hello World!"

Im nächsten Kapitel werden wir dieses von Visual Studio automatisch erstellte Programm genauer untersuchen.

## Kapitel 3

# Das erste Programm in C#

Als Brian W. Kernighan und Dennis M. Ritchie 1978 das Buch „The C Programming Language“ veröffentlichten, haben sie damit maßgeblich zur Verbreitung der Programmiersprache C beigetragen. In diesem Buch wird dem Leser der Einstieg in die Programmierung mit dem sogenannten „Hello World!“-Programm vermittelt. Das „Hello World!“-Programm ist ein einfaches Computerprogramm, das nichts weiter tut, als den Text „Hello World!“ auf dem Bildschirm auszugeben. Damit wurde die Tradition geboren, ein Buch über eine Programmiersprache mit einem „Hello World!“-Programm zu beginnen. Da sich die Syntax von C# sehr stark an die Syntax der Programmiersprache C anlehnt, möchte ich mich in die Tradition von Kernighan und Ritchie und vielen anderen Fachbuchautoren einreihen und dieses Kompendium mit einem „Hello World!“-Programm eröffnen.

### 3.1 Der Aufbau einer Konsolenanwendung

Eine Konsolen-App besteht aus mindestens einer Programmdatei. Bis wir zum Kapitel „Einführung in die Objektorientierte Programmierung“ kommen, benötigen wir auch nur diese eine Programmdatei. Diese Datei heißt „Program.cs“ und sieht wie folgt aus.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
```

Unser Programm beginnt mit einem sogenannten `using`-Statement. Mit `using`-Statements legen Sie fest, welche externen Programmbibliotheken das Programm verwenden möchte. Wenn Sie bereits andere Programmiersprachen kennen, wissen Sie sicher bereits, was eine Programmbibliothek ist. Falls Sie mit diesem Buch Ihre erste Programmiersprache erlernen, stellen Sie sich eine Programmbibliothek als eine Sammlung kleiner Hilfsprogramme oder Unterprogramme vor, die jemand anderes geschrieben hat und die Sie in Ihren Programmen verwenden können. Mit dem `using`-Statement

```
1 using System;
```

legen Sie fest, dass Sie die Programmbibliothek `System` in Ihrem Programm verwenden möchten. Die Programmbibliothek „`System`“ ist Bestandteil des .NET Frameworks. Im weiteren Verlauf dieses Buches werden Sie noch viele andere Programmbibliotheken kennen lernen.

3

Die nächste Komponente des Programms ist ein sogenannter Namespace oder Namensraum. Ein Namensraum benötigt einen eindeutigen Namen gefolgt von einem zum Namensraum gehörenden Programmcode, der in geschweifte Klammern eingeschlossen ist.

```
1 namespace HelloWorld
2 {
3     ...Weiterer Programmcode
4 }
```

Der Name unseres Namensraums heißt genauso wie unser Programm `HelloWorld`. Im Kapitel „Eine Klassenbibliothek erstellen“ werde ich detaillierter erklären, wozu ein Namensraum gut ist und was man damit machen kann. Bis dahin genügt es zu wissen, dass wir unser Programm mit einem Namensraum umgeben müssen.

Als nächstes kommt unser eigentliches Programm:

```
1 class Program
2 {
3     ...Weiterer Programmcode
4 }
```

Es beginnt mit dem Schlüsselwort `class` und dem Namen `Program`. Mit diesem Namen legen wir fest, dass es sich um unser Hauptprogramm handelt. Später werden wir auch noch Unterprogramme kennenlernen. Der Inhalt unseres Hauptprogramms ist wie beim Namensraum in geschweiften Klammern eingeschlossen.

Jedes Hauptprogramm benötigt einen Startpunkt. Das ist der Punkt, an dem der Computer mit der Ausführung des Programms beginnt. Bei C# Konsolen-Apps ist der Startpunkt die sogenannte Hauptmethode. Sie sieht wie folgt aus:

```
1 static void Main(string[] args)
2 {
3     ...Weiterer Programmcode;
4 }
```

Der Name der Hauptmethode ist vorgeschrieben und heißt `Main`, dadurch weiß der Computer, wo er mit der Ausführung des Programms beginnen soll. Der Inhalt der Hauptmethode, das eigentliche Programm, ist wieder in geschweiften Klammern eingeschlossen. Dass die Hauptmethode mit dem Schlüsselwort `static` beginnt, müs-

### 3 Das erste Programm in C#

sen Sie erst mal so hinnehmen. In einem späteren Kapitel werde ich erklären, was es damit auf sich hat.

Jetzt kommen wir zum eigentlichen Programm. In unserem Fall besteht es aus nur einem Programmbebefhl.

```
1 Console.WriteLine("Hello World!");
```

Damit rufen wir das Hilfsprogramm `Console` auf, welches sich in der Programmblibliothek `System` befindet (siehe `using`-Statement am Anfang). Nach `Console` kommt ein „.“ und danach das Unterprogramm `WriteLine()` aus dem Programm `Console`. Das Unterprogramm `WriteLine()` gibt einen Text am Bildschirm aus. Der Text, den `WriteLine()` ausgeben soll, folgt direkt und ist mit runden Klammern umgeben. Und damit der Computer unterscheiden kann, was freier Text ist und was nicht, müssen wir den freien Text „Hello World!“ mit oberen Anführungszeichen umschließen. Zum Abschluss eines Programmbebefhls benötigen wir in C# noch ein Semikolon.

## 3.2 Aufrufargumente verarbeiten

Jetzt haben wir die Funktionsweise unseres ersten „Hello World!“-Programms zum größten Teil verstanden. Was noch fehlt, ist die Erklärung, warum nach unserer Hauptmethode in runden Klammern

```
1 string[] args
```

steht. Dazu klicken Sie mit der rechten Maustaste auf das Projekt „Hello World!“ im Projektmappen-Explorer und klicken dann im Kontextmenü auf „Eigenschaften“.

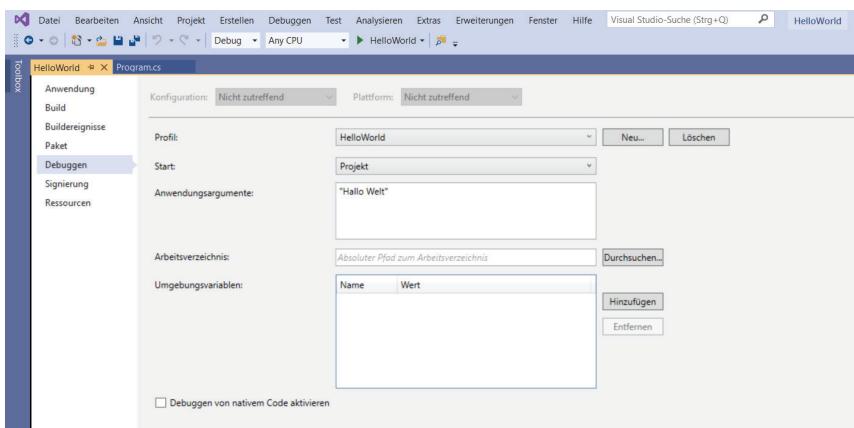


Abb. 3.2.1 Anwendungsargumente in den Projekteigenschaften festlegen

Im Textfeld Anwendungsargumente geben Sie „Hallo Welt“ ein.

Um zu verstehen, was Anwendungsargumente sind, müssen Sie sich vor Augen halten, dass Sie ein Programm normalerweise nicht in Form einer Programmcode-Datei an Benutzer geben und die Benutzer Ihr Programm dann mit Hilfe von Visual Studio ausführen. Normalerweise erstellen Sie mit Visual Studio eine Programmdatei, die in unserem Fall dann „HelloWorld.exe“ heißt und liefern diese dann an Benutzer aus. Wenn ein Benutzer dann die Datei „HelloWorld.exe“ auf seinen Computer in ein Verzeichnis kopiert, kann er das Windows-Befehlszeilenfenster öffnen, in das Verzeichnis, das „HelloWorld.exe“ enthält, wechseln und dann mit dem Befehl „HelloWorld“ das Programm starten. Der Benutzer kann im Befehlszeilenfenster aber auch den folgenden Befehl eingeben:

HelloWorld „Das ist ein beliebiger Text“

oder:

HelloWorld Hallo Welt

Diese beiden Befehle würden zu keinem Fehler führen, sondern einfach nur unser „HelloWorld“-Programm starten. Alles, was nach dem Befehl „HelloWorld“ kommt, nennt man die Anwendungsargumente. Wie Sie vielleicht schon bemerkt haben, stehen die Anwendungsargumente im ersten Beispiel in Anführungszeichen, im zweiten Beispiel dagegen nicht. Der Hintergrund ist, dass es im ersten Beispiel nur ein einziges Anwendungsargument gibt, nämlich den Text „Das ist ein beliebiger Text“. Im zweiten Beispiel gibt es zwei Anwendungsargumente: den Text „Hallo“ und den Text „Welt“.

Wozu sind diese Anwendungsargumente nun gut, unser Programm gibt doch so oder so nur den Text „Hello World!“ am Bildschirm aus. Die Anwendungsargumente, die ein Benutzer beim Aufruf des Programms angibt, können wir in unseren Programmen verwenden. Wir können `args[0]` als Platzhalter für das erste Anwendungsargument und `args[1]` für als Platzhalter für das zweite Anwendungsargument verwenden, usw.

Probieren wir das ganze doch einmal aus. Ändern Sie das Programm „HelloWorld“ wie folgt ab:

```
1  namespace HelloWorld
2  {
3      class Program
4      {
5          static void Main(string[] args)
6          {
7              Console.WriteLine(args[0]);
```

### 3 Das erste Programm in C#

```
8 }  
9 }  
10 }
```

Wir übergeben jetzt an die Methode `Console.WriteLine()` nicht mehr den Text „Hello World!“, sondern `args[0]` und zwar ohne Anführungszeichen. Dadurch weiß der Computer, `args[0]` ist nicht irgendein beliebiger Text, sondern die Aufforderung, das erste Anwendungsargument zu nehmen und es am Bildschirm anzuzeigen. Starten wir jetzt unser Programm:

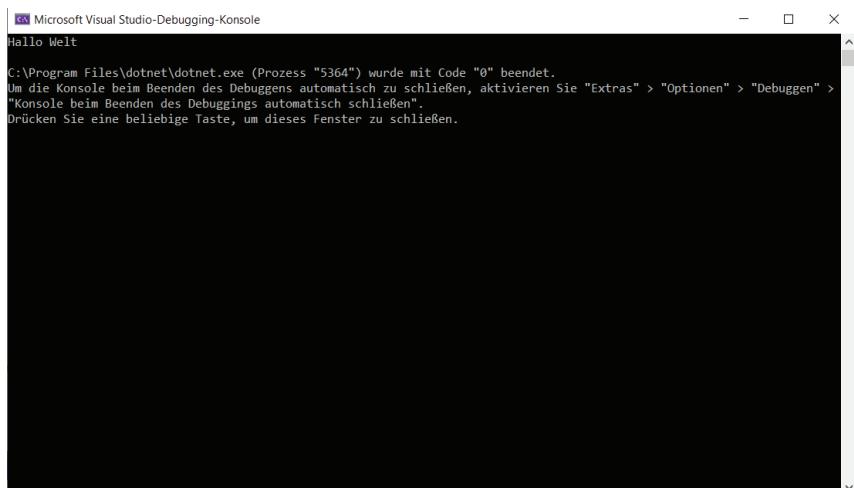


Abb. 3.2.2 "Hallo Welt" statt "Hello World!" durch Anwendungsargumente

Nun gibt unser Programm nicht mehr „Hello World!“ aus, sondern das, was wir in den Projekteigenschaften als Anwendungsargument festgelegt haben. Wenn wir unser Programm jetzt kompilieren und weitergeben, wird es immer das ausgeben, was der Benutzer über die Anwendungsargumente festlegt. Mit Anwendungsargumenten können Sie Ihre Programme dynamisch und flexibel gestalten.

### 3.3 Kommentare erleichtern das Leben

Nachdem Sie jetzt erfolgreich Ihr erstes Programm geschrieben haben, möchte ich ein Thema ansprechen, das in der Praxis von vielen Programmierern gerne vernachlässigt wird. Das Kommentieren von Programmcodes. Unser kleines „HelloWorld!“-Programm ist einfach und übersichtlich und jeder der C# programmieren kann, versteht es sofort. Aber in der Realität ist ein Computerprogramm, das eine halbwegs sinnvolle und brauchbare Aufgabe erledigen soll, deutlich komplexer. Es besteht sehr schnell aus 50 bis 100 Dateien und eine Datei umfasst auch mal schnell mehr als 1.000 Zeilen. Diese

genannten Zahlen gelten aber nur für kleine Projekte, bei großen Projekten kann man sie getrost noch mal mit 10 multiplizieren. Zudem beschreiben Programmcodes meistens auch komplexe logische Sachverhalte, sodass man schnell den Überblick verlieren kann. Kommentare können hier Abhilfe schaffen.

Wenn Sie in ihren Programmcode zwei Schrägstriche // einfügen, dann wird alles, was nach den Schrägstrichen bis zum Zeilenende kommt, ignoriert. Das heißt, das Programm arbeitet so, als hätten Sie den Kommentar nicht eingefügt. Zudem färbt Visual Studio alle Kommentare zur Hervorhebung grün ein:

```
1 //Das ist ein Kommentar
```

Für mehrzeilige Kommentare gibt es eine vereinfachte Syntax, sodass nicht vor jede Zeile zwei Schrägstriche gestellt werden. Ein Kommentar-Block beginnt mit den Zeichen /\* und endet mit den Zeichen \*/

```
1 /* Das ist ein mehrzeiliger Kommentar.
2 Das ist die zweite Zeile des Kommentars
3 und hier die dritte*/
```

Visual Studio kann auch strukturierte Kommentare automatisch erzeugen. Wenn Sie über die Zeile `class Program` drei Schrägstriche schreiben, fügt Visual Studio das folgende Codeschnipsel ein:

```
1 /// <summary>
2 ///
3 /// </summary>
```

In der mittleren Zeile können Sie Ihren Kommentar schreiben. Diese so strukturierten Kommentare können verwendet werden, um aus Ihrem Programmcode eine Dokumentation zu generieren.

Platzieren Sie jetzt den Cursor über der Zeile `static void Main(string[] args)`, tippen Sie drei Schrägstriche ein und so erhalten Sie wieder einen strukturierten Kommentar.

```
1 /// <summary>
2 ///
3 /// </summary>
4 /// <param name="args"></param>
```

Jetzt erkennt Visual Studio, dass es hier nicht nur unsere Hauptmethode „Main“, sondern auch die Anwendungsargumente „args“ zu kommentieren gibt. Hier können Sie zum Beispiel eine Beschreibung hinterlassen, die angibt, wie die Anwendungsargumente in Ihrem Programm verwendet werden.

### 3 Das erste Programm in C#

Zum Abschluss des Kapitels möchte ich Ihnen jetzt noch eine aufwändig kommentierte Version der „Hello World!“-App zeigen:

```
1  ****
2  * Das ist eine dynamische Variante des klassischen
3  * Hello World-Programms, welches das erste
4  * Anwendungsargument am Bildschirm ausgibt.
5  * Aufruf:
6  *
7  * HelloWorld "Hallo Welt!"
8  ****
9
10 using System;
11
12 namespace HelloWorld
13 {
14     /// <summary>
15     /// Das Hauptprogramm
16     /// </summary>
17     class Program
18     {
19         /// <summary>
20         /// Die Hauptmethode
21         /// </summary>
22         /// <param name="args">Die Anwendungsargumente: Nur das
23         erste
24         /// wird am Bildschirm ausgegeben.</param>
25         static void Main(string[] args)
26         {
27             Console.WriteLine(args[0]);
28         }
29     }
30 }
```

#### 3.4 Übungsaufgabe: Eine Erweiterung für „Hello World!“

Schreiben Sie ein Programm, das „Hello World!“ in drei verschiedenen Sprachen am Bildschirm ausgibt. Falls Sie keine dritte Sprache können oder falls Sie kein Übersetzungsprogramm bemühen wollen: „Hello World!“ heißt auf Spanisch „Hola mundo!“. Benutzen Sie dabei auch strukturierte Kommentare.

**Musterlösung:**

```
1  ****
2  * Hello World! in drei Sprachen
3  * Englisch, Deutsch, Spanisch
4  * ****
5
6  using System;
7
8  namespace Uebung_3_5
9  {
10     /// <summary>
11     /// Das Hauptprogramm
12     /// </summary>
13     class Program
14     {
15         /// <summary>
16         /// Die Hauptmethode, der Startpunkt des Programms
17         /// </summary>
18         /// <param name="args">Die Anwendungsbargumente,
19         /// werden hier nicht benutzt.</param>
20         static void Main(string[] args)
21         {
22             Console.WriteLine("Hello World!"); //Englisch
23             Console.WriteLine("Hallo Welt!"); //Deutsch
24             Console.WriteLine("Hola Mundo!"); //Spanisch
25         }
26     }
27 }
```

3

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 4

# Variablen und ihre Typen

In diesem Kapitel geht es um ein grundlegendes Konzept, das sich in fast allen Programmiersprachen, wenn auch in unterschiedlichen Formen, findet: die Variablen.

C# gehört zu den sogenannten streng typisierten Programmiersprachen. Was das im Detail bedeutet, werden Sie unter anderem in diesem Kapitel erfahren.

4

### 4.1 Variablen in der Programmierung: Fast so wie in der Mathematik

Auch wenn Sie noch nie programmiert haben, von Variablen haben Sie mit Sicherheit in der Schulmathematik schon mal gehört. Eigentlich wird ein Grundverständnis von Variablen heute bereits in der ersten Klasse gelehrt. Dort gibt es Aufgaben, die in etwa so lauten:

$5 + \square = 7$  (Liest sich: Fünf plus Kästchen ist gleich Sieben). Fülle das Kästchen aus. In höheren Klassen werden dann die Zahlen größer, die Terme komplexer: Statt Kästchen wird „x“ geschrieben und „x“ wird dann tatsächlich auch als Variable bezeichnet. Der Lehrer erklärt dann, dass man seine Variablen beliebig benennen kann, dass es völlig egal ist, wie eine Variable heißt, aber an der Tafel bleibt er dann trotzdem beim „x“. Das wäre dann auch schon der erste Unterschied zwischen Mathematik und Programmierung. In der Programmierung achten wir sehr wohl darauf, unseren Variablen vernünftige Bezeichnungen zu geben. Variablen sind Speicherplätze, in denen Informationen gespeichert werden können. Informationen können alles Mögliche sein, wie zum Beispiel ein Kontostand in einer Bankanwendung oder der Text „Es geht ein Biba-Butzemann in unserem Kreis herum“ in einer Applikation für Kinderlieder. Daraus sollten wir unseren Variablen dann auch sprechende Namen geben wie „Kontostand“ oder „Liedtitel“. Das erhöht die Lesbarkeit Ihrer Programme beträchtlich.

In der Mathematik haben Variablen einen konkreten Wert, während Variablen in der Programmierung nur auf einen Speicherplatz verweisen, um auf den dort abgelegten Wert zu referenzieren, ohne dass dieser eine bestimmte Gleichung erfüllen muss.

Ein weiterer wichtiger Unterschied zur Mathematik ist die Definition des Wertebereichs einer Variablen. In der Mathematik ist das die sogenannte Grundmenge. Zu einer Mathematikaufgabe gehört immer eine Grundmenge. Das heißt nur Zahlen, die in dieser Grundmenge enthalten sind, sind für die Lösung auch erlaubt. Die erste Grundmenge, die man im Schulunterricht kennenlernt, ist die Menge der natürlichen Zahlen. Das sind ganze Zahlen wie 5, 7 oder 3.456.783.365, aber auch Zahlen

#### 4 Variablen und ihre Typen

mit etlichen Quadrilliarden von Stellen, denn die Menge der natürlichen Zahlen ist unendlich groß. Und das geht in der Programmierung nicht, denn der Speicherplatz eines Computers ist eben **nicht** unendlich groß. In der Programmierung kann man in Variablen nur eine begrenzte Menge unterschiedlicher Werte speichern.

Aus der Schule sind wir es gewohnt, dass Variablen nur Zahlen enthalten können. In der Programmierung können Variablen auch Texte, Listen und Verzeichnisse enthalten. Es gibt sogar mehrdimensionale Datenstrukturen.

## 4.2 Variablen, Typen und Operatoren

Nach so viel Theorie widmen wir uns der Praxis. Wenn wir in einem C#-Programm eine Variable benötigen, schreiben wir zum Beispiel:

```
1 int x;
```

Jetzt haben wir einen Speicherplatz, in dem wir Information ablegen können. Mit dem Schlüsselwort `int` haben wir den Typ der Variablen festgelegt, denn `int` steht für „Integer“ und das bedeutet ganze Zahlen, inklusive der Null und es sind sowohl negative als auch positive Zahlen erlaubt. Die kleinste Zahl, die wir in der Variablen „`x`“ speichern können, ist `-2.147.483.648` und die größte Zahl ist `2.147.483.647`. Das sind genau `4.294.967.295` verschiedene Werte und das ist genau die Anzahl von verschiedenen Werten, die man mit einer 32-Bit-Zahl darstellen kann. Ein Bit ist eine Stelle einer Zahl. Allerdings rechnen Computer nicht wie wir im Dezimalsystem, wo ein Stelle Werte von `0-9` haben kann. Computer rechnen im Binärsystem und dort kann eine Stelle nur die Werte `0` oder `1` haben. Wenn wir also eine Binärzahl mit 32 Stellen betrachten, sehen wir 32 Stellen, die entweder Null oder Eins sein können. Das heißt, wir haben  $2^{32}$  Möglichkeiten und  $2^{32}$  ergibt genau `4.294.967.295`. Um positive und negative Zahlen speichern zu können, verwenden Computer das erste Bit, um festzulegen, ob die Zahl positiv oder negativ ist. Damit verbleiben für die eigentliche Zahl 31 Bit, was `2.147.483.648` Möglichkeiten oder Zahlen von `0` bis `2.147.483.648` entspricht. Die Null wird den positiven Zahlen zugeschlagen, daher reicht der Größte `int`-Wert „nur“ bis `2.147.483.647`, während für negative Zahlen die `2.147.483.648` Möglichkeiten der 31 Bit als Zahlen von `-1` bis `-2.147.483.648` verwendet werden. Dass wir 32 Bit und nicht mehr oder weniger für den Typ `int` verwenden, liegt daran, dass Computer 32-Bit Zahlen auf einen Schlag sehr schnell verarbeiten können. Aber fürs praktische Programmieren reicht es, wenn Sie sich merken, dass der Wertebereich für Variablen vom Typ „`int`“ ungefähr von `-2 Milliarden` bis ungefähr `+2 Milliarden` reicht.

Wenn Sie mehrere Variablen vom gleichen Typ benötigen, können sie das mit folgender Programmzeile erledigen:

```
1 int x, y;
```

Jetzt haben Sie 2 Variablen vom Typ `int` deklariert, mit den Namen `x` und `y`. Variablen vom Typ `int` haben nach ihrer Deklaration den Wert Null. Wenn sie einer Variablen einen Wert zuweisen wollen, können sie das mit dem Zuweisungsoperator = tun.

```
1 x = 5;
```

Damit haben wir den Wert 5 in der Variablen `x` gespeichert. Einer Variablen kann auch gleich bei Ihrer Deklaration ein Wert zugewiesen werden:

```
1 int x = 5;
```

4

Man kann mit dem Zuweisungsoperator auch den Inhalt einer Variablen auf eine andere Variable übertragen.

```
1 y = x;
```

Damit bekommt `y` den gleichen Wert wie `x`.

Wie Sie sich sicher jetzt denken können, können wir mit Variablen auch rechnen. Bei `int`-Variablen sind die Operatoren für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division, die Zeichen: +, -, \* und /. Es gilt die aus der Mathematik bekannte „Punkt vor Strich“-Regel, das heißt Multiplikationen und Divisionen werden vor Additionen und Subtraktionen ausgeführt. Zudem können Klammern genauso wie in der Mathematik verwendet werden. Mit dem folgenden Code-Schnipsel können Sie zum Beispiel das Volumen eines Körpers berechnen.

```
1 int laenge = 20;
2 int breite = 30;
3 int tiefe = 40;
4 int volumen = laenge * breite * tiefe;
```

Bei einer Division von zwei `int`-Variablen führt C# auch eine echte mathematische Integer-Division durch:

```
1 x = 5 / 2;
```

Die Variable `x` enthält jetzt den Wert 2, denn 5 geteilt durch 2 ist 2, Rest 1. Der Rest 1 wird dann elegant vergessen. Sind Sie am Rest der Integer-Division interessiert, können Sie den Modulo-Operator % verwenden.

```
1 x = 5 % 2;
```

Damit haben wir `x` nicht das Ergebnis der Integer-Division zugewiesen, sondern den Rest 1.

Das Schlüsselwort `int` ist eigentlich nur ein Alias, eine andere Bezeichnung für den eigentlichen Namen des Variablentyps, der in Wirklichkeit `Int32` heißt. Das heißt,

#### 4 Variablen und ihre Typen

es besteht kein Unterschied, ob Sie eine Variable mit `int` oder `Int32` deklarieren. Außerdem ist `int` beziehungsweise `Int32` nicht der einzige Variablentyp für ganze Zahlen, den es in C# gibt. Die folgende Tabelle zeigt Ihnen eine Übersicht über Ganzzahltypen in C#:

Datentyp	Alias	Wertebereich
Sbyte	sbyte	-128 bis +127
Int16	short	-32768 bis +32.767
Int32	int	-2.147.483.648 bis 2.147.483.647
Int64	long	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
Byte	byte	0 bis 255
UInt16	ushort	0 bis 65.535
UInt32	uint	0 bis 4.294.967.295
UInt64	ulong	0 bis 18.446.744.073.709.551.615

Bevor ich mich der nächsten Gruppe von Datentypen widme, möchte ich noch kurz den Begriff des Literals klären. Als Literal bezeichnet man in der Programmierung einen konkreten Wert. Zum Beispiel ist `5` ein Literal für eine Integer-Variable. Da die Literale für Integer-Variablen genauso geschrieben werden, wie wir die Zahlen auch sonst schreiben würden, nur ohne Tausender-Punkt, war bei den Integer-Variablen das Thema noch ohne Bedeutung. Bei anderen Variablentypen ist das aber nicht so, daher gebe ich bei den weiteren Variablentypen immer auch die Schreibweise der zugehörigen Literale an.

Die nächste Gruppe der Variablentypen, die wir betrachten, kann Gleitkommazahlen speichern. Die Deklaration dieser Variablen erfolgt genauso wie die Deklaration von Integer-Variablen, nur muss dann das Schlüsselwort für den jeweiligen Gleitkommavariablentyp verwendet werden, zum Beispiel:

```
1 double x;
```

Die nächste Tabelle gibt Ihnen einen Überblick über die Variablentypen für Gleitkommazahlen in C#:

Datentyp	Alias	Wertebereich	Literal
Single	float	-3.402823 E38 bis 3.402823 E38 (auf 8 Stellen genau)	7.5f

Double	double	-1.79769313486232 E308 bis 1.79769313486232 E308 (auf 15 bis 16 Stellen genau)	12.456d
Decimal	decimal	+/- 1,0 E-28 bis +/- 7,9 E28 (28 Stellen genau)	454.458m

Möglicherweise sind Sie jetzt etwas verwirrt. Wozu benötigt man 3 Gleitkommatypen und wozu brauchen wir dann noch die vorher beschriebenen Integer-Typen. Wenn wir nur float verwenden würden, könnten wir alles rechnen und der Wertebereich von float ist für die allermeisten Anwendungen völlig ausreichend. Der Teufel liegt wie immer im Detail. Testen wir einmal das folgende Programm:

```

1  using System;
2
3  namespace Variablen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              float x = 0.123456789f;
10             Console.WriteLine(x);
11         }
12     }
13 }
```

Natürlich würden Sie erwarten, dass dieses Programm 0.123456789 am Bildschirm ausgegeben wird. Wenn Sie es ausprobieren, sehen Sie aber 0,12345678. Nach der achten Nachkommastelle wird abgeschnitten. Im Programmcode haben Sie die Gleitkommazahl mit einem Punkt eingegeben, weil Visual Studio sonst den Dienst verweigert, und am Bildschirm wird die Zahl mit einem Komma dargestellt. Nach der achten Nachkommastelle wird abgeschnitten, weil Gleitkommatypen nur eine begrenzte Genauigkeit haben und beim Typ „float“ liegt die eben bei 8 Stellen. Dass die Ausgabe statt eines Punktes ein Komma verwendet, liegt daran, dass in Ihrem Windows vermutlich festgelegt ist, dass Sie eine deutsche Kultureinstellung verwenden und in Deutschland werden Gleitkommazahlen, wie der Name schon sagt, mit einem Komma geschrieben. Auf Englisch heißen sie floating point numbers und werden daher in England und den USA natürlich mit einem Punkt geschrieben. Noch schlimmer wird es, wenn wir das folgende Programm betrachten:

```

1  using System;
2
3  namespace Variablen
4  {
5      class Program
6      {
7          static void Main(string[] args)
```

## 4 Variablen und ihre Typen

```
8     {
9         double a = 69.82d;
10        double b = 69.2d + 0.62d;
11        Console.WriteLine(a - b);
12    }
13}
14}
```

Die zu erwartenden Ausgabe wäre eigentlich 0 oder 0,0 oder so ähnlich. Wir erhalten aber:

-1,4210854715202E-14

Das ist eine sehr kleine Zahl, aber leider nur fast richtig. Unser Gleitkommatyp rechnet ein bisschen ungenau, obwohl wir nur mit 2 Stellen nach dem Komma arbeiten. Und jetzt treiben wir den Irrsinn noch auf die Spitze und ändern unser Programm leicht ab:

```
1 using System;
2
3 namespace Variablen
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             float a = 69.82f;
10            float b = 69.2f + 0.62f;
11            Console.WriteLine(a - b);
12        }
13    }
14}
```

Statt dem Typ „double“ verwendet wir jetzt den Typ „float“, ansonsten bleibt alles beim Alten. Aber am Bildschirm wird jetzt korrekt 0 ausgegeben. Was ist das denn? Wir verwenden einen Variablentyp mit der geringeren Genauigkeit und erhalten ein besseres Ergebnis?

Das liegt an der internen Speicherung von float- und double-Werten. Manchmal, wenn die Werte ungünstig liegen, treten Ungenauigkeiten auf. Bei diesem extremen Beispiel funktioniert float besser als double, aber meistens ist es umgekehrt.

Natürlich stellt sich jetzt die Frage, welchen Datentyp Sie wofür verwenden sollen. Wenn Sie mit dem Wertebereich von Integer-Typen auskommen und wenn Sie nur ganze Zahlen benötigen, verwenden Sie am besten den jeweiligen Integer-Typen, dann müssen Sie sich nicht um Genauigkeiten und Nachkommastellen kümmern. Wenn Ihnen ganze Zahlen nicht mehr ausreichen, verwenden Sie den Typ „double“. Wenn Sie grafische Anwendungen wie physikalische Simulationen oder anspruchs-

volle Computerspiele programmieren, verwenden Sie den Typ „float“, da hier die geringere Genauigkeit weniger zu Buche schlägt, aber Ihre Programme profitieren von der höheren Geschwindigkeit, mit denen float-Typen verarbeitet werden können. Bei finanzmathematischen Anwendungen sollten Sie unbedingt den Typ „decimal“ verwenden, da hier exakte Rechenergebnisse zwingend vorgeschrieben sind. Sie erkennen sich damit die größere Genauigkeit durch eine geringere Rechengeschwindigkeit, aber das ist hier zweitrangig.

Vielleicht haben Sie in Ihrer Schulzeit schon mal von Boole'scher Algebra gehört. Wenn nicht, dann holen wir das kurz nach. Die Boole'sche Algebra heißt so, weil Sie von einem Herrn Boole erfunden wurde und beschreibt das „Rechnen“ mit sogenannten Wahrheitsausdrücken. Natürlich gibt es hier für die Mathematiker wieder eine Grundmenge und für die Programmierer wieder einen Wertebereich. Erfreulicherweise sind hier Grundmenge und Wertebereich identisch, denn bei der Bool'schen Algebra ist der Wertebereich endlich. Er besteht aus nur zwei Werten: „true“ und „false“, also richtig und falsch. Auf derartige Werte können wir nicht unsere vier Grundrechenarten Addition, Subtraktion, Multiplikationen und Division loslassen. Für Bool'sche Werte gibt es die Operationen „logical and“, „logical or“ und „logical not“ oder kurz „and“, „or“ und „not“.

In C# verwenden wir für den Operator „logical and“ die Zeichen „&&“ und für „logical or“ „||“. Die senkrechte Linie erzeugen wir mit der Taste für „<>“ und gleichzeitigem Drücken der Taste „AltGr“. Für den Operator „logical not“ verwenden wir das Zeichen „!“. Wie mit Wahrheitsausdrücken gerechnet werden kann, zeigt Ihnen die folgende Rechentafel:

Erster Operand	Operator	Zweiter Operand	Ergebnis
True	&&	true	true
True	&&	false	false
False	&&	true	false
False	&&	false	false
True		true	true
True		false	true
False		true	true
False		false	false

Den Operator „!“ können wir vor einen Operanden schreiben und wir drehen dadurch den Operanden um. Das heißt, aus `true` wird `false` und aus `false` wird `true`. Außerdem gibt es noch den Vergleichsoperator, der auf alle Variablen, die den gleichen

#### 4 Variablen und ihre Typen

Typ haben, angewendet werden kann. Aber das Ergebnis des Vergleichsoperators ist immer vom Typ `bool` und zwar `true`, wenn zwei Variablen gleich sind, und `false`, wenn nicht. Der Vergleichsoperator wird mit den Zeichen „`==`“ geschrieben.

Bool'sche Variablen werden mit Hilfe des Schlüsselworts `Boolean` beziehungsweise mit dem Alias `bool` deklariert. Wenn Bool'schen Variablen bei der Deklaration kein Wert zugewiesen wird, enthalten sie zunächst den Wert `false`. Sehen wir uns dazu ein paar Beispiele an.

```
1 bool a = true;
2 bool b = a;
3 bool c = !a;
4 bool x = a && b;
5 bool y = a || c;
6 bool z = a == b;
```

Jetzt enthalten die Variablen die folgenden Werte

a	true
b	true
c	false
x	true
y	true
z	true

Wenn Sie jetzt der Meinung sind, dass Boole'sche Variablen nur von akademischem Wert sind, warten Sie bitte bis wir uns mit Verzweigungen und Schleifen beschäftigen. In Kombination damit entfalten die Boole'schen Variablen ihr volles Potenzial.

Den nächsten Variablentyp haben Sie schon im „Hello World!“-Programm gesehen, zumindest als Literal. „Hello World!“ ist ein Literal für den Variablentyp Zeichenkette. Er wird mit dem Schlüsselwort `String` oder mit dem zugehörigen Alias `string` deklariert, zum Beispiel:

```
1 string helloText = "Hello World!";
```

Zeichenkettenvariablen oder kurz „Strings“ werden als Literal in doppelten Anführungszeichen eingeschlossen. Sie können sämtliche Zeichen des sogenannten Unicode-Zeichensatzes enthalten. Das sind 1.111.998 verschiedene Zeichen. Das beinhaltet alle Groß- und Kleinbuchstaben des lateinischen Alphabets, jede Menge Sonderzeichen, Interpunktionszeichen und nicht druckbare Zeichen, aber auch Alphabete anderer Kulturen, wie Griechisch, Kyrillisch, Hebräisch, Arabisch, Chinesisch, Japanisch, Thai, Hindi und viele andere. Eine vollständige Beschreibung des Unicode-Zeichensatzes finden Sie auf Wikipedia unter diesem Link:

<https://de.wikipedia.org/wiki/Unicode>

Nicht druckbare Zeichen sowie Zeichen, die für die Kennzeichnung als Literal Verwendung finden, werden in C# mit sogenannten Escape-Sequenzen dargestellt. Dem Zeichen wird ein sogenannter Backslash \ vorangestellt. Wenn Sie in einem String Literal \" verwenden, bedeutet das nicht, dass die doppelten oberen Anführungszeichen das Literal beenden, sondern dass der String ein doppeltes oberes Anführungszeichen beinhaltet. Die wichtigsten Escape-Sequenzen für die Programmierpraxis sind:

\r	Wagenrücklauf
\n	neue Zeile
\t	Tabulator
\"	doppelte obere Anführungszeichen
\\	Backslash

4

Wenn Sie das folgende Programm ausführen, sehen Sie den Effekt von Escape-Sequenzen:

```
1 namespace Variablen
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             string text = "Programmieren lernen mit \r\n\"Hello
8                 World\""
9             Console.WriteLine(text);
10        }
11    }
12 }
```

Mit den Escape-Sequenzen \r\n sorgen wir dafür, dass alle folgenden Zeichen in einer neuen Zeile ausgegeben werden und die Escape Sequenz \" gibt ein doppeltes oberes Anführungszeichen aus. In diesem Beispiel erzeugen wir den Zeilenumbruch mit den zwei Escape Sequenzen \r\n, aber es würde auch mit \n allein funktionieren. Der Grund, warum beide Varianten existieren, sehen wir, wenn wir uns mit dem Einlesen von Textdateien beschäftigen. Lesen wir eine Textdatei ein, die mit einem Windowsprogramm, zum Beispiel „Notepad“, erzeugt wurde, so finden wir zwei Zeichen für den Zeilenumbruch. Kommt unsere Textdatei aber von einem Unix- oder Linux-System, so gibt es nur ein Zeichen für den Zeilenumbruch.

C# verfügt noch über eine weitere Methode, ein String-Literal darzustellen, die besonders für längere Texte geeignet ist. Wenn Sie vor das Literal den Klammeraffen @ setzen, können Sie Zeilenumbrüche, Tabulatoreinzüge und den Backslash einfach tippen, ohne Escape-Sequenzen verwenden zu müssen. Benötigen Sie in Ihrem Text

## 4 Variablen und ihre Typen

doppelte Anführungszeichen, müssen Sie diese „ zweimal tippen, damit der Compiler zwischen dem Literal Kennzeichnung und dem Literal Textinhalt unterscheiden kann:

```
1 string text = @"Das ist ein langer Text
2 mit Tabulatoren einzügen allerdings müssen Sie
3 ""Anführungszeichen"" hier doppelt schreiben.
4 Der Backslash \ geht aber ohne Escape Sequenz";
```

Wenn Sie ein String-Literal mit String-Variablen kombinieren wollen, benötigen Sie eine weitere Variante von String-Literalen. Bei dieser Variante setzen sie das Dollarzeichen „\$“ vor das Literal. Escape-Sequenzen können und müssen Sie hier genauso verwenden wie in der Variante, die einfach nur mit oberen Anführungszeichen arbeitet. Aber jetzt können Sie Variablennamen, die mit geschweiften Klammern umgeben sind, in Ihren String-Literalen verwenden und der Computer ersetzt die Variablennamen mit dem Inhalt der Variablen. Das nächste Beispielprogramm zeigt diese Variante von String-Literalen:

```
1 namespace Variablen
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             int a = 5;
8             int b = 2;
9             int c = a + b;
10            string ausgabe = $"a) + {b} = {c}";
11            Console.WriteLine(ausgabe);
12        }
13    }
14 }
```

Die Ausgabe des Programms lautet – wie nicht anders zu erwarten – „ $5 + 2 = 7$ “. Es funktioniert, obwohl wir hier ein String-Literal mit Integer-Variablen kombinieren. Die Konvertierung von `int` zu `string` erfolgt automatisch.

Wenn Sie nur ein einziges Unicode-Zeichen speichern wollen, hat C# dafür einen eigenen Typen. Er wird mit dem Schlüsselwort `Char` beziehungsweise dem Alias `char` deklariert. Als Literal wird er mit einem einfachen oberen Anführungszeichen umgeben. Auch die Verwendung von Escape-Sequenzen ist möglich, wie die folgenden Beispiele zeigen:

```
1 char nurEinZeichen = 'x';
2 char tabulator = '\t';
```

Bei den letzten Variablentypen, die ich in diesem Kapitel vorstellen möchte, geht es um Datum und Uhrzeit. Der erste Typ, den wir aus dieser Gruppe betrachten, heißt

DateTime. Er besitzt keinen Alias und dient zur Speicherung von Datum und Uhrzeit. Das heißt, er speichert eigentlich einen Zeitpunkt, eine Uhrzeit an einem bestimmten Datum. Für DateTime gibt es kein Literal. Aber Sie können einer DateTime-Variablen mit der Parse-Methode einen Wert zuweisen:

```
1 DateTime zeitpunkt = DateTime.Parse("2020-12-14 20:15:28.456");
```

Der String, der vom DateTime.Parse akzeptiert wird, besitzt das Format:

„Jahr-Monat-Tag Stunden:Minuten:Sekunden.Millisekunden“

4

Die Uhrzeit kann entfallen. Dann speichert die DateTime-Variable die Uhrzeit „0:00“.

„2020-12-14“ ist gleichbedeutend mit „2020-12-14 00:00:00.000“. Falls Sie eine Urzeit in Ihrem Datumsformat angeben, können auch die Sekunden und Millisekunden entfallen. Der Werte-Bereich des Variablentyps „DateTime“ reicht von „0001-01-01 00:00:00.000“ bis „9999.12.31 23:59:59.999“. Für Zeitpunkte, die außerhalb dieses Wertebereichs liegen, gibt es in C# keinen Datentyp.

Wenn Sie die den aktuellen Zeitpunkt in einem Programm ermitteln wollen, erhalten Sie ihn mit dem Ausdruck „DateTime.Now“. Intern wird eine DateTime-Variable in Ticks gespeichert. Ein Tick entspricht einer 10-millionstel Sekunde, also 100 Nanosekunden. Das heißt, eine DateTime-Variable ist eigentlich eine Integer-Variable, die die Anzahl der vergangenen Ticks, die seit dem 1. Januar 0001 um 00:00 Uhr vergangen sind, speichert. Mit dem folgenden Programmcode können Sie die Anzahl der Ticks des aktuellen Zeitpunkts ausgeben:

```
1 DateTime jetzt = DateTime.Now;
2 Console.WriteLine(jetzt.Ticks);
```

Ein weiterer zeitbezogener Variablentyp ist TimeSpan. Er stellt einen Zeitraum dar und wird ebenfalls in Ticks gemessen. Genauso wie bei DateTime gibt es für ihn kein Literal. Zum Zuweisen eines Wertes bietet TimeSpan die Methoden „FromDays()“, „FromHours()“, „FromMinutes()“, „FromSeconds()“, „FromMilliseconds()“ und „FromTicks()“. Mit folgendem Beispielcode können Sie einer TimeSpan-Variablen einen Wert zuweisen:

```
1 DateTime ein10SekundenZeitraum = TimeSpan.FromSeconds(10);
```

Mit einer geschickten Kombination aus „DateTime“ und „TimeSpan“ lässt sich die Ausführungszeit eines Programms beziehungsweise eines Programmteils messen, wie das folgende Beispiel zeigt:

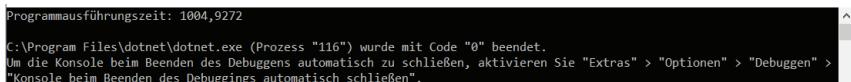
```
1 using System;
2 using System.Threading;
```

## 4 Variablen und ihre Typen

```

3
4 namespace ZeitMessung
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             DateTime start = DateTime.Now.Ticks;
11             Thread.Sleep(1000);
12             DateTime ende = DateTime.Now.Ticks;
13             TimeSpan dauer = TimeSpan.FromTicks(ende - start);
14             Console.WriteLine($"Programmausführungszeit: {dauer.
15             TotalMilliseconds}");
16         }
17     }
18 }
```

Zum Start unseres Programms messen wir die Zeit und speichern sie in Form von Ticks in der Variablen „start“. Danach simulieren wir die Programmausführungszeit mit dem Aufruf `Thread.Sleep(1000)`, der nichts anderes tut als 1000 Millisekunden zu warten. Damit wir diese Methode verwenden können, müssen wir im using-Block auch die Klassenbibliothek „System.Threading“ einbinden. Nachdem das Programm 1000 Millisekunden gewartet hat, messen wir die Zeit wieder und speichern sie als Ticks in der Variablen „ende“. Nun weisen wir der Variablen „dauer“ mit der Methode `TimeSpan.FromTicks` einen Zeitraum zu, der genau den vergangenen Ticks entspricht. Die vergangenen Ticks berechnen wir mit dem Ausdruck „ende – start“. Zum Schluss geben wir „dauer.TotalMilliseconds“ aus, was der vergangenen Zeit in Millisekunden entspricht.



```
Programmausführungszeit: 1004,9272
C:\Program Files\dotnet\dotnet.exe (Prozess "116") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
```

**Abb. 4.2.1** Die gemessene Programmausführungszeit

Unser Messwert ist 1004,9272 Millisekunden. Dabei hätten wir eigentlich 1000 Millisekunden erwartet. Das liegt zum einen daran, dass die Methode `Thread.Sleep` nicht hundertprozentig exakt arbeitet und daran, dass unser Messwert auch einen kleinen Overhead für die Berechnung der Dauer enthält.

### 4.3 Variablen in einem C#-Programm verwenden

Wir haben zwar schon gesehen, wie man eine Variable deklariert, aber ich möchte ich hier nochmal detailliert darauf eingehen. Bei der Deklaration einer Variablen wird zuerst der Typ der Variablen zum Beispiel `int` geschrieben. Es kann der eigentliche Typ der Variable zum Beispiel `Int32` oder der zugehörige Alias zum Beispiel `int` verwendet werden. In der Praxis hat sich eingebürgert, dass immer der Alias verwendet wird.

Nach dem Typ der Variablen folgt ein Leerzeichen und dann der Name der Variablen. Ein Variablenname kann Großbuchstaben, Kleinbuchstaben, Zahlen und den Unterstrich „\_“ enthalten. Zudem gilt die Regel, dass Variablennamen nicht mit einer Zahl beginnen dürfen. Umlaute und andere kulturspezifische Buchstaben sind zwar erlaubt, aber deren Verwendung gilt in der Praxis als schlechter Programmierstil. Variablennamen sollten sprechend sein, das heißt, sie sollten semantisch zu den Variablen passen. Wenn Sie in einer Variablen den Vornamen einer Person speichern wollen, dann sollte die Variable auch „Vorname“, „vorName“ oder „vorname“ oder so ähnlich heißen, aber bestimmt nicht „abc25“. Das verbessert die Lesbarkeit Ihrer Programme beträchtlich. Wenn Sie beruflich in der Softwareentwicklung arbeiten, werden Ihre Programme auch von Kollegen gelesen und die werden es Ihnen danken, wenn Sie sprechende Variablennamen verwenden.

Unter Programmierern gibt es immer wieder die Diskussion, ob Variablennamen Deutsch oder Englisch gehalten werden sollten. Englische Variablennamen haben den Vorteil, dass sie internationale Zusammenarbeit erleichtern und dass man auch mal einen Programmierer engagieren kann, der kein Deutsch spricht, wenn Not am Mann ist. Deutsche Variablennamen sind hilfreich für Kollegen, deren Englischkenntnisse etwas eingerostet sind. In der Praxis verwende ich stets englische Variablennamen, sofern es von meinem Kunden keine anderen Vorgaben gibt. Da dieses Buch allerdings auf Deutsch geschrieben ist, verwende ich hier deutsche Variablennamen um Lesern, die mit Ihrem Englisch etwas aus der Übung sind, das Verständnis zu erleichtern.

Variablennamen müssen zudem innerhalb ihres Gültigkeitsbereichs eindeutig sein. Bisher haben wir Variablennamen nur innerhalb der Hauptmethode einer Konsole-App deklariert. Die Hauptmethode ist einer von vielen Gültigkeitsbereichen. Das heißt, innerhalb der Hauptmethode darf ein Variablenname nicht doppelt vorkommen. Im Laufe dieses Buchs werden wir noch mehr Gültigkeitsbereiche kennenlernen.

In der C#-Programmierung werden für die Vergabe von Namen bestimmte Konventionen verwendet. Die Konventionen sind nicht zwingend vorgeschrieben. Das bedeutet, wenn man gegen sie verstößt, funktionieren die Programme trotzdem, aber es gilt als schlechter Stil, wenn sie nicht eingehalten werden. In diesem Buch bin ich immer dann, wenn es vom Thema her passt, auf die dazugehörigen Konventionen eingegangen. Variablen in der Hauptmethode sind sogenannte lokale Variablen. Für lokale Variablen gilt die Konvention, dass sie in sogenanntem Camel Case geschrieben werden. Sie beginnen mit einem Kleinbuchstaben und wenn Sie aus zwei oder mehreren Wörtern bestehen, werden diese zusammengeschrieben und das zweite und jedes weitere Wort beginnt mit einem Großbuchstaben, zum Beispiel:

```
1 string akademischerTitel = "Prof. Dr. Dr.;"
```

## 4 Variablen und ihre Typen

Eine weitere Möglichkeit, lokale Variablen zu deklarieren, ist die sogenannte implizite Deklaration. Dabei verwenden Sie einfach, statt der Angabe des Typs, das Schlüsselwort `var`. Wenn Sie jetzt glauben, Sie können damit auf eine Typisierung von Variablen verzichten, dann haben Sie sich leider geirrt. Versuchen Sie doch mal die folgende Zeile in einem C#-Programm:

```
1 var meineVariable;
```

Visual Studio wird den Variablennamen mit einer roten Wellenlinie unterlegen und wenn Sie die Maus über den Variablennamen bewegen, sehen Sie die Fehlermeldung: „Implizit deklarierte Variablen müssen initialisiert werden.“ Ok, dann versuchen wir mal eine Initialisierung:

```
1 var meineVariable = "Hallo";
```

Die rote Wellenlinie verschwindet und das Programm kann wieder kompiliert werden. Sieht so aus, als hätten wir das Problem gelöst und wir haben unsere nicht typisierte Variable doch noch bekommen. Doch schon mit einer weiteren Codezeile fallen wir auf wieder auf die Nase:

```
1 var meineVariable = "Hallo";
2 meineVaridable = 5;
```

Jetzt ist die 5 mit einer roten Wellenlinie unterlegt und die zugehörige Fehlermeldung heißt: „Der Typ int kann nicht implizit in string konvertiert werden.“

Die Variable „meineVariable“ ist eben doch **nicht** untypisiert. Durch die Zuweisung des string-Literals „Hallo“ erhält sie den Typ `string` und dieser Typ kann nachträglich auch nicht mehr geändert werden. C# ist eine streng typisierte Sprache. Beim Thema „Linq“ werden wir diese „Implizite Deklaration von lokalen Variablen“ noch sehr gut gebrauchen können. Damit Sie sich an diese Art der Variablen-deklaration gewöhnen, werde ich sie von jetzt an in den Beispielen verwenden.

### 4.4 Benutzereingaben in Variablen speichern

Bisher haben wir nur die Anwendungsargumente kennengelernt, um Informationen von außen in ein Computerprogramm einzubringen. In diesem Kapitel möchte ich Ihnen eine Methode vorstellen, wie ein Programm eine Dateneingabe direkt vom Benutzer abfragen kann. Am besten wird das anhand eines Beispielprogramms deutlich.

```
1 using System;
2
3 namespace Benutzereingabe
4 {
5     class Program
```

## 4.4 Benutzereingaben in Variablen speichern

```
6  {
7      static void Main(string[] args)
8      {
9          Console.WriteLine("Wie heißen Sie?");
10         var benutzerName = Console.ReadLine();
11         var ausgabe = $"Guten Tag {benutzerName}." ;
12         Console.WriteLine(ausgabe);
13     }
14 }
15 }
```

4

Außer dem Aufruf `Console.ReadLine()` gibt es in diesem Programm nichts, was Sie nicht schon kennen. `Console.ReadLine()` ist ein weiteres Unterprogramm des Hilfsprogramms `Console`. An dieser Stelle möchte ich aufhören, in diesem Zusammenhang die Begriffe Unterprogramm und Hilfsprogramm zu verwenden. Ich habe sie bisher nur zur besseren Veranschaulichung benutzt. Ich möchte diese Begriffe durch die professionelleren Bezeichnungen „statische Methode“ und „Klasse“ ersetzen. Man sagt `ReadLine()` ist eine statische Methode der Klasse `Console`, die von der Klassenbibliothek `System` zur Verfügung gestellt wird. In einem späteren Kapitel werde ich diese Begriffe detailliert erläutern. Die Methode `ReadLine()` hält die Programmausführung an und wartet, bis der Benutzer mit der Tastatur Informationen eingibt und diese mit der Taste „Enter“ bestätigt. Dann weist `ReadLine()` die Eingabe der Variablen `benutzerName` zu. Die Variable `benutzerName` ist implizit deklariert und bekommt den Typ `string`, da `ReadLine()` generell einen String erzeugt. Starten wir jetzt unser neues Programm.

Ein Fenster mit der Ausgabe „Wie heißen Sie?“ wird angezeigt und in der zweiten Zeile sehen Sie einen blinkenden Cursor.



Abb. 4.4.1 Die Konsolen-App fragt eine Benutzereingabe ab

Geben Sie jetzt Ihren Namen ein und drücken Sie die Taste „Enter“.



Abb. 4.4.2 Der Benutzer macht eine Eingabe

## 4 Variablen und ihre Typen

Unser neues Programm begrüßt Sie mit Ihrem Namen.

```
Microsoft Visual Studio-Debugging-Konsole
Wie heißen Sie?
Robert
Guten Tag Robert.

C:\Program Files\dotnet\dotnet.exe (Prozess "16064") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 4.4.3** Die Ausgabe der Konsolen-App unter Verwendung der Eingabe

## 4.5 Konvertieren von Variablen

Im letzten Kapitel haben wir gesehen, wie wir Benutzereingaben über die Tastatur in einer Konsolen-App verarbeiten können. Damit könnten wir eigentlich ein Programm schreiben, das zwei Zahlen vom Benutzer entgegennimmt, diese dann addiert und das Ergebnis am Bildschirm ausgibt. Allerdings sind die Zahlen, die der Benutzer eingibt, wie wir bereits wissen, in Strings gespeichert. Und mit Strings können wir nicht rechnen. Aber sehen wir mal, was passiert, wenn wir es trotzdem versuchen.

```
1 using System;
2
3 namespace Variablen
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var a = "4"; // Die Zahl 4 als string
10            var b = "2"; // Die Zahl 2 als string
11            var ausgabe = a + b; //Mal sehen was da gerechnet
12            wird
13            Console.WriteLine(ausgabe);
14        }
15    }
16 }
```

Die Antwort ist:

```
Microsoft Visual Studio-Debugging-Konsole
42

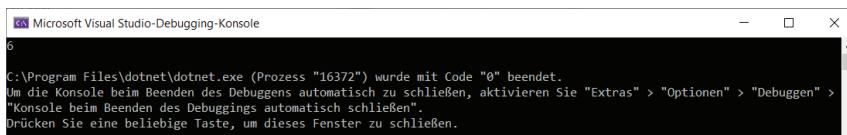
C:\Program Files\dotnet\dotnet.exe (Prozess "15860") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 4.5.1** „4“ plus „2“ gleich „42“

Dieses unerwartete Verhalten kommt daher, dass der Operator „+“ bei Strings nicht die gleiche Funktion wie bei Integer-Variablen hat. Bei Strings führt das Pluszeichen keine Addition aus, sondern eine sogenannte Konkatenation. Das heißt, er fügt die Strings einfach zusammen. Wenn wir also rechnen wollen, müssen wir die Strings in Integer-Variablen verwandeln. Die Variablentypen, die wir bisher kennengelernt haben, würde man in anderen Programmiersprachen als „Primitives“ oder primitive Datentypen bezeichnen. In C# gibt es aber keine primitiven Typen, alle Variablentypen sind Klassen – genauso wie die Klasse „Console“, die Sie bereits kennen. All diese Typen, die Sie bereits kennen, haben eine statische Methode, die Parse() heißt. Sie dient, dazu einen String in den jeweiligen Variablentyp zu verwandeln. Programmierer sprechen von konvertieren. Ändern wir also unser Rechenprogramm etwas ab:

```
1  using System;
2
3  namespace Variablen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var a = int.Parse("4");
10             var b = int.Parse("2");
11             var ausgabe = a + b;
12             Console.WriteLine(ausgabe);
13         }
14     }
15 }
```

Und diesmal stimmt die Rechnung.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The console output is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
6
C:\Program Files\dotnet\dotnet.exe (Prozess "16372") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 4.5.2 Vier plus zwei gleich sechs

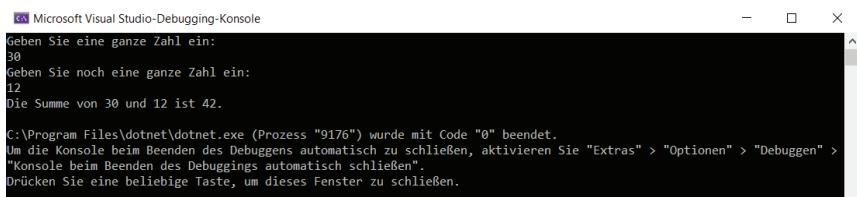
int.Parse() liefert eine Variable vom Typ int, dadurch bekommt die implizit deklarierte Variable a den Typ int. Das Gleiche gilt für die Variable b. Beim Ausdruck a + b wird der Operator + jetzt auf 2 Variablen vom Typ int angewendet, daher gibt er jetzt auch den Typ int zurück und somit wird die Variable ausgabe auch zu int. Die Methode WriteLine() konvertiert automatisch das nun richtig gerechnete Ergebnis in einen String zurück und gibt ihn am Bildschirm aus.

#### 4.6 Übungsaufgabe: Rechnen mit Variablen

Schreiben Sie ein Programm, das vom Benutzer nacheinander zwei ganze Zahlen einliest, die beiden Zahlen dann addiert und das Ergebnis am Bildschirm ausgibt.

**Musterlösung:**

```
1  using System;
2
3  namespace Variablen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
10             var a = Console.ReadLine();
11             Console.WriteLine("Geben Sie noch eine ganze Zahl
12             ein:");
13             var b = Console.ReadLine();
14             var ausgabe = int.Parse(a) + int.Parse(b);
15             Console.WriteLine($"Die Summe von {a} und {b} ist
16             {ausgabe}.");
17         }
18     }
19 }
```



```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
30
Geben Sie noch eine ganze Zahl ein:
12
Die Summe von 30 und 12 ist 42.

C:\Program Files\dotnet\dotnet.exe (Prozess "9176") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 4.6.1** Ausgabe der Musterlösung

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 5

# Verzweigungen zur Steuerung des Programmablaufs

Bisher haben wir nur sogenannte sequenzielle Programme geschrieben. Jedes unserer bisherigen Programme startet, arbeitet dann eine feste Abfolge von Schritten ab und beendet sich schließlich wieder. Mit den sogenannten Verzweigungen werden wir das ändern. Damit können wir Programme schreiben, die – abhängig von bestimmten Bedingungen – verschiedene Programmzweige durchlaufen.

### 5.1 Die einfache, bedingte Verzweigung

Die einfache, bedingte Verzweigung, auch als if-else-Verzweigung bekannt, sieht formal wie folgt aus:

```
1  if (Boole'scher Ausdruck)
2      //Anweisung(en) falls der Boole'sche Ausdruck wahr ist
3  else
4      //Anweisung(en) falls der Boole'sche Ausdruck falsch ist
```

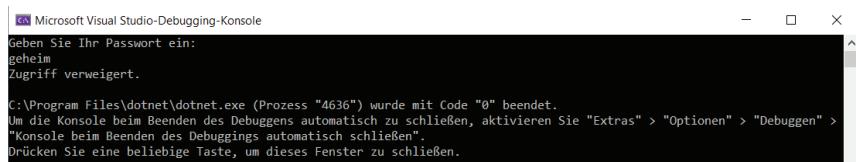
Die Verzweigung beginnt mit dem Schlüsselwort `if`, gefolgt vom einem Boole'schen Ausdruck in runden Klammern. Ein Boole'scher Ausdruck ist entweder eine Variable vom Typ `bool` oder ein Ausdruck aus Variablen, Operatoren und Literalen, der den Typ `bool` liefert. Nach der `if`-Anweisung folgen eine oder mehrere Anweisungen, die das Programm ausführen, falls der Boole'sche Ausdruck wahr ist. Bei mehr als einer Anweisung müssen die Anweisungen von geschweiften Klammern eingeschlossen sein. Nach den Anweisungen für den Wahr-Fall folgt das Schlüsselwort „`else`“ und dann die Anweisung(en) für den Falsch-Fall. Gibt es hier mehr als eine Anweisung, so müssen diese auch mit geschweiften Klammern umgeben werden. Das nächste Beispielprogramm zeigt eine Anwendung der einfachen bedingten Verzweigung:

```
1  using System;
2
3  namespace EinfacheBedingteVerzweigung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie Ihr Passwort ein:");
10             var password = Console.ReadLine();
11             if(password == "1234")
12             {
```

## 5 Verzweigungen zur Steuerung des Programmablaufs

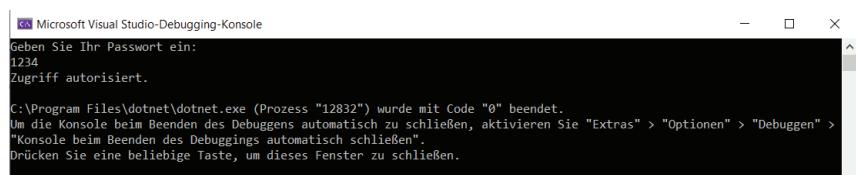
```
13             Console.WriteLine("Zugriff autorisiert.");
14         }
15     else
16     {
17         Console.WriteLine("Zugriff verweigert.");
18     }
19 }
20 }
21 }
```

Zuerst fordert das Beispielprogramm den Benutzer zur Eingabe seines Passworts auf und dann liest es eine Eingabe ein. Dann kommt unsere einfache bedingte Verzweigung. Der auszuwertende Boole'sche Ausdruck ist `passwort == "1234"`. Das Gleichzeichen schreiben wir doppelt, da es sich nicht um eine Zuweisung, sondern um einen Vergleichsoperator handelt. Der Vergleichsoperator vergleicht die String-Variable `passwort` mit dem Literal „1234“. Sind beide gleich, so ergibt der Ausdruck den Boole'schen Wert „true“, sonst den Boole'schen Wert „false“. Geben wir als Passwort „1234“ ein, so führt das Programm den Wahrzweig der `if`-Anweisung aus und gibt „Zugriff autorisiert.“ am Bildschirm aus. Geben wir aber ein falsches Passwort ein, so führt das Programm den Falschzweig der `else`-Anweisung aus und wir lesen „Zugriff verweigert.“ am Bildschirm.



The screenshot shows a Windows command-line window titled "Microsoft Visual Studio-Debugging-Konsole". It displays the following text:  
Geben Sie Ihr Passwort ein:  
geheim  
Zugriff verweigert.  
  
C:\Program Files\dotnet\dotnet.exe (Prozess "4636") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

Abb. 5.1.1 Ausgabe bei falschem Passwort



The screenshot shows a Windows command-line window titled "Microsoft Visual Studio-Debugging-Konsole". It displays the following text:  
Geben Sie Ihr Passwort ein:  
1234  
Zugriff autorisiert.  
  
C:\Program Files\dotnet\dotnet.exe (Prozess "12832") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

Abb. 5.1.2 Ausgabe bei richtigem Passwort

## 5.2 Vergleichsoperatoren und logische Operatoren

Vergleichsoperatoren vergleichen zwei Operanden und geben als Ergebnis einen Boole'schen Wert zurück. Dabei müssen die Operanden miteinander kompatible Typen besitzen. Ansonsten zeigt Visual Studio einen Fehler an.

Im vorigen Kapitel haben wir bereits den Gleichheitsoperator kennengelernt. Er kann auf Strings und auf Zahlen angewendet werden. Es können ganze Zahlen mit Gleitkommazahlen, aber nicht Zahlen mit Strings verglichen werden. Wenn wir folgende drei Variablen deklariert haben,

```
1 var a = 5;
2 var b = 5.5d;
3 var c = "Fünf";
```

dann können wir Verzweigungen wie `if(a == 5)`, `if(b == 5.5d)`, `if(a == b)` oder `if(c == "Fünf")` verwenden, aber nicht `if(a == "Hallo")` oder `if(c == 5)`, da Strings nicht mit Zahlenwerten verglichen werden dürfen.

5

Zahlen können auch mit den Ungleichheitsoperatoren "`<`", "`>`", "`<=`" und "`>=`" verglichen werden. Im nächsten Beispiel-Programm steuern wir den Programmablauf mit einer auf einem Ungleichheitsoperator basierenden Verzweigung.

```
1 using System;
2
3 namespace Alterskontrolle
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Wie alt sind Sie?");
10            var alter = int.Parse(Console.ReadLine());
11            if (alter >= 18)
12                Console.WriteLine("Sie sind bereits
13                volljährig!");
14            else
15                Console.WriteLine("Sie sind noch nicht
16                volljährig!");
17        }
18    }
19 }
```

Mit der Anweisung

```
1 var alter = int.Parse(Console.ReadLine());
```

fragen wir das Alter des Benutzers ab. Hierbei schachteln wir zwei Methoden ineinander. Der Rückgabewert der Methode `ReadLine()` geht dabei als Übergabeparameter in die Methode `Parse()`.

Nachdem der Benutzer sein Alter eingibt, steuert der Ausdruck „`alter >= 18`“, welche Aussage das Programm über die Volljährigkeit des Benutzers macht.

## 5 Verzweigungen zur Steuerung des Programmablaufs

Bisher haben unsere Verzweigungen immer nur eine Bedingung ausgewertet, um eine Entscheidung zu treffen. Da eine Bedingung wie „`alter >= 18`“ für einen Bool'schen Wert steht, können Bedingungen mit den logischen Operatoren „`&&`“, „`||`“ oder „`!`“, die wir bereits aus dem Kapitel „Variablen und Ihre Typen“ kennen, verknüpft werden. Damit kann eine Entscheidung von mehreren Bedingungen abhängig gemacht werden. Aber dazu erst mal ein Beispiel:

```

1  using System;
2
3  namespace BenutzerUndPasswort
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie Ihren Benutzernamen:");
10             var benutzer = Console.ReadLine();
11             Console.WriteLine("Geben Sie Ihr Passwort ein:");
12             var passwort = Console.ReadLine();
13             if(benutzer == "Robert" && passwort == "1234")
14             {
15                 Console.WriteLine("Zugriff autorisiert!");
16             }
17             else
18             {
19                 Console.WriteLine("Zugriff verweigert!");
20             }
21         }
22     }
23 }
```

Jetzt hängt die Entscheidung, die an der if-else-Verzweigung getroffen wird, von den beiden Bedingungen `benutzer == "Robert"` und `passwort == "1234"` ab. Die beiden Bedingungen werden dazu mit einem logischen „und“ verknüpft. Wenn beide Bedingungen wahr sind, so ist auch der Gesamtausdruck wahr.

Ein Boole'scher Ausdruck kann auch deutlich komplexer werden und wie in der Mathematik Klammern zur Bestimmung der Auswertungsreihenfolge enthalten.

```
1  if(alter >= 67 || (alter < 67 && beitragsJahre >= 45))
```

Mit diesem Ausdruck können wir bestimmen, ob eine Rente mit oder ohne Abschlag gezahlt wird. Die Rentenregel für Männer lautet: „Man muss mindestens 67 Jahre alt sein oder 45 Jahre lang Rentenbeiträge gezahlt haben, um seine Rente ohne Abschläge zu erhalten.“ Damit haben wir eine verwaltungstechnische Vorschrift in einen mathematischen Ausdruck gegossen.

Die Auswertung eines Boole'schen Ausdrucks erfolgt nicht vollständig. Sobald das Ergebnis vorzeitig klar ist, wird die Auswertung abgebrochen. Das hat einen kleinen,

aber feinen Einfluss auf die Programmierung. Wenn wir zwei Boole'sche Ausdrücke a und b haben und wenn wir jetzt die logische Verknüpfung „a und b“ auswerten, dann können wir abbrechen, falls a falsch ist. Dann muss der Gesamtausdruck falsch sein, egal welchen Wert b hat. Sie erinnern sich: Eine und-Verknüpfung ist nur dann wahr, wenn beide Operanden wahr sind. Umgekehrt verhält es sich, wenn wir „a oder b“ auswerten. Falls a wahr ist, ist der Gesamtausdruck wahr, egal ob b wahr oder falsch ist. Bei der oder-Verknüpfung genügt es, dass ein Operand wahr ist, damit der Gesamtausdruck wahr ist. Betrachten wir einmal folgendes Codefragment:

```
1 var a = 0;
2 var b = 5;
3 if(a != 0 && (b / a) > 1)
4 {
5     //Programmcode für den Wahrzweig
6 }
```

Mit diesem Ausdruck schützen wir unseren Code vor einer Division durch 0, welche nicht erlaubt ist, weil sie mathematisch keinen Sinn ergibt. Wenn a == 0 ist, dann ist der linke Operand der und-Verknüpfung a != 0 falsch, dadurch wird der rechte Operand der und-Verknüpfung nicht mehr ausgewertet, weil schon klar ist, dass der Gesamtausdruck falsch ist. Wenn a nicht 0 ist, dann ist der linke Operand wahr und der rechte Operand muss auch ausgewertet werden. Aber diesmal ist das kein Problem, weil wir dann ja nicht mehr durch 0 teilen.

Dieser kleine, aber elegante Trick wird uns später, wenn wir die sogenannten „null“-Werte einführen, noch einmal sehr hilfreich sein.

### 5.3 Geschachtelte bedingte Verzweigungen

Bedingte Verzweigungen können ineinander geschachtelt werden. Das heißt, ein Wahrzweig oder Falschzweig oder beide können wieder eine bedingte Verzweigung enthalten und deren Zweige dann wieder und so weiter. Damit können Sie zum Beispiel einen komplexen Boole'schen Ausdruck auf geschachtelte bedingte Verzweigungen verteilen. Das Beispielprogramm, das den Benutzernamen und das Passwort kontrolliert, kann mit geschachtelten bedingten Verzweigungen auch anders geschrieben werden.

```
1 using System;
2
3 namespace BenutzerUndPasswort
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Geben Sie Ihren Benutzernamen:");
10            string name = Console.ReadLine();
11            if(name == "Aalen")
12            {
13                Console.WriteLine("Geben Sie Ihr Passwort:");
14                string password = Console.ReadLine();
15                if(password == "1234")
16                {
17                    Console.WriteLine("Willkommen!");
18                }
19            }
20        }
21    }
22 }
```

## 5 Verzweigungen zur Steuerung des Programmablaufs

```

10     var benutzer = Console.ReadLine();
11     Console.WriteLine("Geben Sie Ihr Passwort ein:");
12     var passwort = Console.ReadLine();
13     if(benutzer == "Robert")
14     {
15         if(passwort == "1234")
16             Console.WriteLine("Zugriff autorisiert!");
17         else
18             Console.WriteLine("Zugriff verweigert!");
19     }
20     else
21     {
22         Console.WriteLine("Benutzer unbekannt!");
23     }
24 }
25 }
26 }
```

Jetzt kontrollieren wir zuerst, ob `benutzer == „Robert“` ist, und wenn das schon falsch ist, wird der Falschweg ausgeführt und „Benutzer unbekannt!“ am Bildschirm ausgegeben. Wenn die Bedingung aber wahr ist, kommt die Überprüfung des Passworts. Ist das Passwort richtig, geben wir „Zugriff autorisiert!“ aus, sonst „Zugriff verweigert!“.

Wie komplex Sie jetzt Ihre Boole'schen Ausdrücke in Ihre if-else-Bedingungen einbauen oder wie komplex Sie Ihre if-else-Bedingungen verschachteln, obliegt Ihrer Kreativität als Programmierer. Aber denken Sie daran: Ein Kollege sollte in der Lage sein, anhand Ihres Programmcodes auch Ihre Gedanken nachvollziehen zu können.

### 5.4 Switch-Case: Die Mehrfachverzweigung

Mit der if-else-Verzweigung bekommen wir immer nur zwei mögliche Zweige für den Programmablauf. Benötigen wir mehr als zwei Zweige, so müssen wir mehrere if-else-Bedingungen schachteln. Wäre es da nicht schön, wenn wir ein Konstrukt hätten, das von vornherein mehr als zwei Zweige zur Programmausführung unterstützt?

So ein Konstrukt ist die Switch-Case-Mehrfachverzweigung. Bevor wir diese Verzweigung näher betrachten, möchte ich Ihnen zuerst die sogenannten Aufzählungstypen oder Enumerationen vorstellen, da diese im Zusammenhang mit Mehrfachverzweigungen für einen eleganten und gut lesbaren Programmcode sorgen. Eine Aufzählung beginnt mit dem Schlüsselwort `enum`, dann folgt ein Name für die Aufzählung und dann kommt eine in geschweifte Klammern eingeschlossene und durch Komma getrennte Liste der Elemente des Aufzählungstyps. Eine Aufzählung für Wochentage könnte so aussehen:

```

1 enum Wochentag
2 {
```

```
3     Montag,
4     Dienstag,
5     Mittwoch,
6     Donnerstag,
7     Freitag,
8     Samstag,
9     Sonntag
10 }
```

Wie bereits erwähnt, ist eine Aufzählung ein Typ, das heißt, wir haben uns gerade einen neuen Variabtentyp für Wochentage geschaffen. Wir können also eine Variable mit dem Typ Wochentag deklarieren:

```
1 Wochentag wochentag;
```

5

Wenn wir unserer Variablen einen Wert zuweisen wollen, geht das wie folgt:

```
1 wochentag = Wochentag.Montag;
```

Natürlich können wir der Variablen auch gleich bei der Deklaration einen Wert zuweisen:

```
1 Wochentag wochentag = Wochentag.Montag;
```

Intern speichert C# Aufzählungen als Integer-Werte, so wird durch unsere Definition der Aufzählung „Wochentag“ der Wert „Wochentag.Montag“ als 0 gespeichert und der Wert Wochentag.Sonntag als 6. Wenn wir die Aufzählung der Wochentage nicht mit 0, sondern mit 1 beginnen lassen wollen, so können wir das bei der Definition der Aufzählung angeben:

```
1 enum Wochentag
2 {
3     Montag = 1,
4     Dienstag = 2,
5     Mittwoch = 3,
6     Donnerstag = 4,
7     Freitag = 5,
8     Samstag = 6,
9     Sonntag = 7
10 }
```

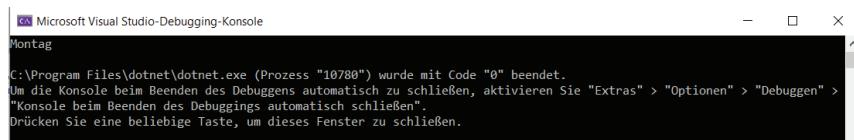
Natürlich können wir auch andere Integer-Werte verwenden. Schauen wir uns doch mal an, was passiert, wenn wir eine Variable vom Typ Wochentag am Bildschirm ausgeben:

```
1 using System;
2
3 namespace Mehrfachverzweigung
4 {
5 }
```

## 5 Verzweigungen zur Steuerung des Programmablaufs

```
6
7     class Program
8     {
9         enum Wochentag
10        {
11            Montag = 1,
12            Dienstag = 2,
13            Mittwoch = 3,
14            Donnerstag = 4,
15            Freitag = 5,
16            Samstag = 6,
17            Sonntag = 7
18        }
19
20        static void Main(string[] args)
21        {
22            Wochentag wochentag = Wochentag.Montag;
23            Console.WriteLine(wochentag);
24        }
25    }
26 }
```

Die Aufzählung befindet sich außerhalb der Hauptmethode. Das liegt daran, dass C# keine Definitionen von Aufzählungen innerhalb von Methoden erlaubt. Aufzählungen sind meistens allgemeingültig und sollten daher nicht auf eine einzelne Methode beschränkt sein. Doch dazu mehr, wenn wir uns tiefer mit Methoden und Gültigkeitsbereichen beschäftigen.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar reads "Microsoft Visual Studio-Debugging-Konsole". The console output displays the text "Montag" followed by a series of error messages in German: "C:\Program Files\dotnet\dotnet.exe (Prozess "10780") wurde mit Code "0" beendet.", "Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".", and "Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.".

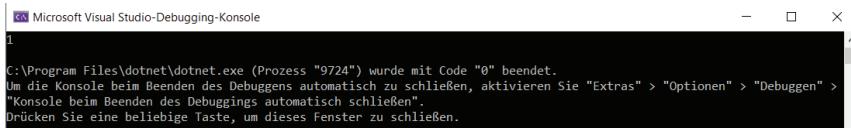
**Abb. 5.4.1** Die Ausgabe des enum-Werts Wochentag.Montag

Diese Ausgabe ist eine Überraschung. Haben wir nicht gerade eben gelernt, dass Aufzählungen als Integer-Werte gespeichert werden? Eigentlich hätten wir „1“ erwartet und nicht „Montag“.

Allerdings haben wir auch gelernt, dass die Methode `Console.WriteLine()` eine Konvertierung in den Typ `string` vornimmt. Bei C# ist die Standard-Konvertierung von `enum` zu `string` so implementiert, dass nicht die dem `enum`-Wert zugrundeliegende Integer-Zahl, sondern der Name des jeweiligen Aufzählungselements verwendet wird. In unserem Fall ist das „Montag“. Wenn wir unsere Anweisung zur Bildschirmausgabe etwas ändern, können wir zeigen, dass für „Wochentag.Montag“ die Zahl 1 gespeichert wird:

```
1 Console.WriteLine((int)wochentag);
```

Wir haben vor, die Variable `wochentag` das Typ-Schlüsselwort `int` in runden Klammern geschrieben. Das ist ein sogenannter Type Cast, damit haben wir C# angewiesen, so zu tun, als hätte die Variable `wochentag` den Typ `int`. Somit bekommt `Console.WriteLine()` einen Integer-Wert übergeben und führt eine Konvertierung von `int` zu `string` aus und die Ausgabe sieht so aus:



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the command `C:\Program Files\dotnet\dotnet.exe` being run, followed by a message indicating it has exited with code 0. Below this, there is a note about automatically closing the console when debugging ends, and at the bottom, a prompt to press any key to close the window.

Abb. 5.4.2 Die Ausgabe des enum-Wertes Wochentag.Montag als Zahl

5

Hiermit beenden wir unseren kleinen Einschub über Aufzählungstypen und verwenden sie gleich, wenn wir uns jetzt mit der Mehrfachverzweigung befassen. Betrachten wir zuerst ein Beispiel:

```
1  using System;
2
3  namespace Mehrfachverzweigung
4  {
5
6
7      class Program
8      {
9          enum Wochentag
10         {
11             Montag = 1,
12             Dienstag = 2,
13             Mittwoch = 3,
14             Donnerstag = 4,
15             Freitag = 5,
16             Samstag = 6,
17             Sonntag = 7
18         }
19
20         static void Main(string[] args)
21         {
22
23             Wochentag wochentag = Wochentag.Montag;
24
25             string ausgabe;
26             switch(wochentag)
27             {
28                 case Wochentag.Montag:
29                     ausgabe = "Oh nein! Die Arbeitswoche hat
30                     begonnen.";
31                     break;
32                 case Wochentag.Dienstag:
33                     ausgabe = "Oje! Die Woche ist noch nicht mal
34                     halb vorbei.";
35                     break;
36             }
37         }
38     }
```

## 5 Verzweigungen zur Steuerung des Programmablaufs

```

36     case Wochentag.Mittwoch:
37         ausgabe = "Na endlich! Die halbe Woche schon
38         vorbei.";
39         break;
40     case Wochentag.Donnerstag:
41         ausgabe = "Wahnsinn! Die Woche ist fast
42         vorbei.";
43         break;
44     case Wochentag.Freitag:
45         ausgabe = "Yaaaaah! Der letzte Arbeitstag der
46         Woche.";
47         break;
48     case Wochentag.Samstag:
49         ausgabe = "Hurra! Endlich Wochenende.";
50         break;
51     default:
52         ausgabe = "Erst mal chillen!";
53         break;
54     }
55
56     Console.WriteLine(ausgabe);
57 }
58 }
59 }
```

Zum Programmstart deklarieren wir die Variable `wochentag` als Aufzählungstyp `Wochentag` und weisen ihr einen Wert zu, zum Beispiel `Wochentag.Montag`. Die Mehrfachverzweigung beginnt mit der Anweisung:

```

1 switch(wochentag)
2 {
3
4 }
```

Damit legen wir fest, dass wir – abhängig vom Inhalt der Variablen `wochentag` – verschiedene Programmzweige durchlaufen wollen. Statt einer Variablen können wir auch einen Ausdruck verwenden. Der Ausdruck beziehungsweise die Variable muss dabei einem der folgenden Typen entsprechen:

- ▶ `char` oder `string`
- ▶ `bool`
- ▶ Ganzahlwerte, wie zum Beispiel `int` oder `long`
- ▶ Aufzählungstypen

Ab C# 7.0, das mit Visual Studio 2017 eingeführt wurde, kann jeder Typ für eine Mehrfachverzweigung verwendet werden.

Innerhalb der geschweiften Klammern kommen die einzelnen Programmzweige. Ein Programmzweig sieht zum Beispiel wie folgt aus:

```
1 case Wochentag.Montag:  
2     ausgabe = "Oh nein! Die Arbeitswoche hat begonnen.";  
3     break;
```

Er beginnt mit dem Schlüsselwort `case`, gefolgt von dem Wert, für den der Zweig durchlaufen wird, und einem abschließenden Doppelpunkt. Danach kommt der eigentliche Programmcode des Zweiges, der mit dem Schlüsselwort `break` und einem Semikolon abgeschlossen wird. Der Programmcode kann aus einer oder mehreren Anweisungen bestehen und muss nicht in geschweifte Klammern eingeschlossen werden. Der letzte Programmzweig kann auch als sogenannter `default`-Zweig formuliert werden:

```
1 default:  
2     ausgabe = "Erst mal chillen!";  
3     break;
```

Der `default`-Zweig wird bei allen Werten durchlaufen, für die keine eigenen Zweige angelegt sind. In unserem Beispiel haben wir eigene Zweige für die Werte `Wochentag.Montag` bis `Wochentag.Samstag` angelegt. Der einzige Wert, für den kein eigener Zweig angelegt ist, ist der Wert `Wochentag.Sonntag`. In diesem Fall landen wir im `default`-Zweig und setzen die Variable `ausgabe` auf den Kommentar zum Sonntag: „Erst mal chillen!“

Für die Mehrfachverzweigung gibt es auch unter bestimmten Voraussetzungen eine verkürzte Schreibweise. Verkürzte und vereinfachte Schreibweisen bezeichnet man in der Programmierung auch als sogenannten „syntactic sugar“. Immer dann, wenn in einigen der Zweige der Mehrfachverzweigung der identische Code ablaufen soll, können wir die folgende Abkürzung verwenden.

```
1 using System;  
2  
3 namespace MehrfachverzweigungKurzform  
4 {  
5     class Program  
6     {  
7         enum Land  
8         {  
9             Deutschland,  
10            Oesterreich,  
11            Schweiz,  
12            Spanien,  
13            Italien,  
14            USA,  
15            England  
16        }  
17    }
```

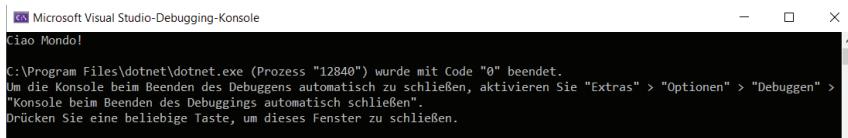
## 5 Verzweigungen zur Steuerung des Programmablaufs

```
18     static void Main(string[] args)
19     {
20         var land = Land.Italien;
21
22         switch(land)
23         {
24             case Land.Deutschland:
25             case Land.Oesterreich:
26             case Land.Schweiz:
27                 Console.WriteLine("Hallo Welt!");
28                 break;
29             case Land.USA:
30             case Land.England:
31                 Console.WriteLine("Hello World!");
32                 break;
33             case Land.Spanien:
34                 Console.WriteLine("Hola Mundo!");
35                 break;
36             case Land.Italien:
37                 Console.WriteLine("Ciao Mondo!");
38                 break;
39         }
40     }
41 }
42 }
43 }
```

Die Variable `land` ist auf einen bestimmten Wert der Aufzählung `Land` gesetzt. Die Aufzählung `Land` beinhaltet die Werte Deutschland, Oesterreich, Schweiz, Spanien, Italien, USA und England. Wir wollen nun mit Hilfe einer Mehrfachverzweigung den Satz „Hello World!“ in der Landessprache des in der Variable `Land` gesetzten Landes ausgeben. Da in Deutschland, Oesterreich und der Schweiz, Deutsch gesprochen wird und in England und den USA Englisch gesprochen wird, würden mehrere Zweige den identischen Code beinhalten. Um unseren Code kompakter zu schreiben, sortieren wir unsere Zweige so, dass Zweige für Länder mit der gleichen Sprache untereinander stehen. So stehen zum Beispiel die Zweige für Deutschland, Österreich und die Schweiz untereinander. Die ersten beiden Zweige Deutschland und Österreich sind leer. Sie enthalten keinen Code. Der Zweig Schweiz enthält den Code für die Ausgabe in Deutsch und ein abschließendes `break;`. Mit den Zweigen für die USA und England verfahren wir analog.

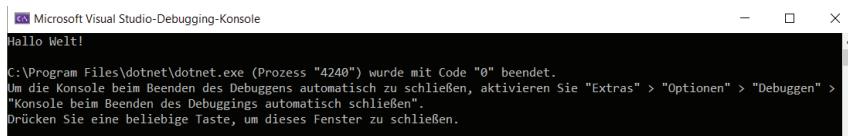
Sobald eine Case-Bedingung der Mehrfachverzweigung erfüllt ist, steigt die Programmierausführung in die Codeausführung ein. Ist die Variable `Land` zum Beispiel auf `Land.Deutschland` gesetzt, wird versucht, den Code, der auf Case `Land.Deutschland:` folgt, auszuführen. Dort gibt es aber keinen Code, daher wird versucht, den Code von `Land.Oesterreich:` auszuführen. Hier gibt es auch keinen Code. Und es geht weiter mit `Land.Schweiz:`. Hier gibt es einen ausführbaren Code und ein abschließendes `break;` das dafür sorgt, dass die Mehrfachverzweigung beendet und die Programmausführung nach der Mehrfachverzweigung fortgesetzt

wird. Jetzt verstehen wir auch, warum wir am Ende eines normalen Zweigs einer Mehrfachverzweigung ein abschließendes `break;` benötigen.



```
C:\Program Files\dotnet\dotnet.exe (Prozess "12840") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

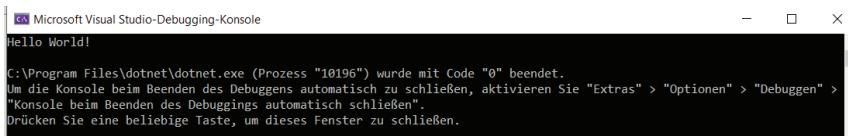
Abb. 5.4.3 Bildschirmausgabe für Land.Italien



```
Hallo Welt!
C:\Program Files\dotnet\dotnet.exe (Prozess "4240") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

5

Abb. 5.4.4 Bildschirmausgabe für Land.Deutschland



```
Hello World!
C:\Program Files\dotnet\dotnet.exe (Prozess "10196") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 5.4.5 Bildschirmausgabe für Land.USA

## 5.5 Die bedingte Zuweisung

Die bedingte Zuweisung ist eigentlich keine eigene Form einer Bedingung oder Zuweisung. Sie ist lediglich eine praktische, abgekürzte Schreibweise einer einfachen bedingten Verzweigung. Microsoft bezeichnet so etwas als syntactic sugar. Betrachten wir noch mal das Beispielprogramm, mit dem wir die einfache bedingte Verzweigung eingeführt haben:

```
1  using System;
2
3  namespace EinfacheBedingteVerzweigung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie Ihr Passwort ein:");
10             var password = Console.ReadLine();
11             if(password == "1234")
12             {
13                 Console.WriteLine("Zugriff autorisiert.");
14             }
15         else
```

## 5 Verzweigungen zur Steuerung des Programmablaufs

```

16         {
17             Console.WriteLine("Zugriff verweigert.");
18         }
19     }
20 }
21 }
```

Es wird die Bedingung: `passwort == „1234“` abgeprüft und, je nachdem ob die Bedingung erfüllt ist oder nicht, wird ein anderer Text ausgegeben. Solche Logiken lassen sich auch mit einer bedingten Zuweisung implementieren.

```

1 using System;
2
3 namespace BedingteZuweisung
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Geben Sie Ihr Passwort ein:");
10            var passwort = Console.ReadLine();
11
12            var ausgabe = passwort == "1234" ? "Zugriff
13                autorisiert." : "Zugriff verweigert.";
14
15            Console.WriteLine(ausgabe);
16        }
17    }
18 }
```

Wir haben wieder unsere Bedingung `passwort == „1234“` gefolgt von einem Fragezeichen und dann von einem Wert, der gilt, falls die Bedingung erfüllt ist und dann von einem Doppelpunkt und von einem Wert, der gilt, falls die Bedingung nicht erfüllt ist. Der Variablen `ausgabe` wird also einer der beiden Werte zugewiesen, je nachdem ob die Bedingung erfüllt ist oder nicht.

Für die Freunde des ultra-kompakten Programmcodes zeige ich hier noch die Variante, die Tastatureingabe, Bedingung, Zuweisung und Bildschirmausgabe in eine einzige Zeile packt.

```

1 Console.WriteLine(Console.ReadLine() == "1234" ? "Zugriff
2 autorisiert." : "Zugriff verweigert.");
```

Für die Lesbarkeit des Programmcodes, besonders wenn die Ausdrücke komplexer werden, empfehle ich diese Zeile umzubrechen und einzurücken:

```

1 Console.WriteLine(
2     Console.ReadLine() == "1234"
3         ? "Zugriff autorisiert."
4         : "Zugriff verweigert.");
```

Für den Compiler ist das immer noch eine einzige geschachtelte Anweisung. Ich habe lediglich Zeilenumbrüche und Einrückungen eingefügt. Derlei Umbrüche und Einrückungen werden als sogenannte Whitespaces bezeichnet und vom Compiler ignoriert, sie sind aber sehr hilfreich, wenn es darum geht, die Lesbarkeit eines Programms zu verbessern. Diese Methode des Umbrechens und Einrückens sollten Sie sich unbedingt beim Lernen von C# zu eigen machen.

Bedingte Zuweisungen können auch wieder geschachtelt werden. Sehen wir uns dazu ein Beispiel an.

```
1  using System;
2
3  namespace BedingteZuweisung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie Ihren Benutzernamen
10             ein:");
11             var benutzer = Console.ReadLine();
12             Console.WriteLine("Geben Sie Ihr Passwort ein:");
13             var passwort = Console.ReadLine();
14
15             var ausgabe = benutzer == "Robert"
16                 ? passwort == "1234"
17                     ? "Zugriff autorisiert."
18                         : "Das Passwort ist falsch."
19                     : "Der Benutzername ist unbekannt.";
20
21             Console.WriteLine(ausgabe);
22
23         }
24     }
25 }
```

5

Zuerst fragen wir Benutzername und Passwort vom Benutzer ab und lesen die Eingaben in die Variablen `benutzer` und `passwort` ein. Dann machen wir eine Zuweisung an die Variable `ausgabe`. Diese machen wir abhängig von der Bedingung `benutzer == „Robert“`: Ist diese Bedingung nicht erfüllt, weisen wir den Wert „Der Benutzername ist unbekannt.“ zu. Ist die Bedingung erfüllt, weisen wir - abhängig von der zweiten verschachtelten Bedingung `passwort == „1234“` - bei Erfüllung den Wert „Zugriff autorisiert.“ und bei Nicht-Erfüllung den Wert „Das Passwort ist falsch.“ zu. Zum Schluss geben wir den so ermittelten Wert der Variablen `ausgabe` am Bildschirm aus.

Wie bei der bedingten Verzweigung kann man bei der bedingten Zuweisung auch mehrere Bedingungen logisch miteinander verknüpfen, sodass die Entscheidung,

## 5 Verzweigungen zur Steuerung des Programmablaufs

welcher Wert am Ende einer Variablen zugewiesen wird, von mehreren Bedingungen abhängen kann.

```
1  using System;
2
3  namespace BedingteZuweisung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie Ihren Benutzernamen
10             ein:");
11             var benutzer = Console.ReadLine();
12             Console.WriteLine("Geben Sie Ihr Passwort ein:");
13             var passwort = Console.ReadLine();
14
15             var ausgabe = (benutzer == "Robert" && passwort ==
16             "1234")
17                 ? "Zugriff autorisiert."
18                 : "Zugriff verweigert.";
19
20             Console.WriteLine(ausgabe);
21
22         }
23     }
24 }
```

Der Variablen `ausgabe` kann entweder der Wert „Zugriff autorisiert.“ oder der Wert „Zugriff verweigert“ zugewiesen werden. Welcher Wert zugewiesen wird, hängt von den Bedingungen `benutzer == „Robert“` und `passwort == „1234“` ab. Die beiden Bedingungen werden mit dem logischen und-Operator „`&&`“ verknüpft. Das heißt, wenn beide Bedingungen wahr sind, wird der Wahr-Wert der bedingten Zuweisung verwendet, wenn eine der beiden Bedingungen falsch ist, wird der Falsch-Wert der bedingten Zuweisung verwendet.

### 5.6 Übungsaufgabe: Programmieren mit Verzweigungen

In dieser Übung werden Sie dreimal das gleiche Programm schreiben, aber mit drei verschiedenen Techniken.

Schreiben Sie ein Programm, das den Benutzer nach dem Namen eines Wochentags fragt und dann „Hurra Wochenende!“ ausgibt, falls der Benutzer „Samstag“ oder „Sonntag“ eingibt und „Oje, Sie müssen arbeiten!“, falls der Benutzer einen der Wochentage von „Montag“ bis „Freitag“ eingibt. Falls der Benutzer keinen Wochentag, sondern irgendetwas anderes eingibt, soll das Programm „Das ist kein Wochentag!“ ausgeben.

**5.6 Übungsaufgabe: Programmieren mit Verzweigungen**

Schreiben Sie das Programm zuerst so, dass die Entscheidungsfindung nur durch if-else-Verzweigungen realisiert wird. Dann schreiben Sie es ein zweites Mal und verwenden zur Entscheidungsfindung eine Switch-Case-Mehrfachverzweigung. Schreiben Sie das Programm ein drittes Mal und benutzen Sie dabei nur bedingte Zuweisungen zur Entscheidungsfindung.

## 5 Verzweigungen zur Steuerung des Programmablaufs

### Musterlösung mit if-else-Verzweigungen:

```
1  using System;
2
3  namespace Musterloesung_5_6_1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie einen Wochentag ein:");
10             var wochentag = Console.ReadLine();
11
12             if (wochentag == "Samstag" || wochentag == "Sonntag")
13             {
14                 Console.WriteLine("Hurra Wochenende!");
15             }
16             else
17             {
18                 if (wochentag == "Montag" ||
19                     wochentag == "Dienstag" ||
20                     wochentag == "Mittwoch" ||
21                     wochentag == "Donnerstag" ||
22                     wochentag == "Freitag")
23                 {
24                     Console.WriteLine("Oje, Sie müssen
25                         arbeiten!");
26                 }
27                 else
28                 {
29                     Console.WriteLine("Das ist kein Wochentag!");
30                 }
31             }
32         }
33     }
34 }
```

**Musterlösung mit einer Switch-Case-Mehrfachverzweigung:**

```
1  using System;
2
3  namespace Musterloesung_5_6_2
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie einen Wochentag ein:");
10             var wochentag = Console.ReadLine();
11
12             switch(wochentag)
13             {
14                 case "Samstag":
15                 case "Sonntag":
16                     Console.WriteLine("Hurra Wochenende!");
17                     break;
18                 case "Montag":
19                 case "Dienstag":
20                 case "Mittwoch":
21                 case "Donnerstag":
22                 case "Freitag":
23                     Console.WriteLine("Oje, Sie müssen
24                         arbeiten!");
25                     break;
26                 default:
27                     Console.WriteLine("Das ist kein Wochentag!");
28                     break;
29             }
30         }
31     }
32 }
```

5

## 5 Verzweigungen zur Steuerung des Programmablaufs

### Musterlösung mit bedingten Zuweisungen:

```
1  using System;
2
3  namespace Musterlösung_5_6_3
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie einen Wochentag ein:");
10             var wochentag = Console.ReadLine();
11
12             var ausgabe = (wochentag == "Samstag" || wochentag
13             == "Sonntag")
14             ? "Hurra Wochenende!"
15             : (
16                 wochentag == "Montag" ||
17                 wochentag == "Dienstag" ||
18                 wochentag == "Mittwoch" ||
19                 wochentag == "Donnerstag" ||
20                 wochentag == "Freitag"
21             )
22             ? "Oje, Sie müssen arbeiten!"
23             : "Das ist kein Wochentag!";
24
25             Console.WriteLine(ausgabe);
26         }
27     }
```

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 6

## Programmteile wiederholen mit Schleifen

Die Leistungsfähigkeit unserer ersten rein sequenziellen Programme konnten wir mit Verzweigungen, die flexibel auf Bedingungen reagieren, deutlich verbessern. Eine weitere Steigerung werden wir in den folgenden Kapiteln erreichen. Dort beschäftigen wir uns mit Schleifen. Das sind Kontrollstrukturen, die Programmteile wiederholen. Grundsätzlich unterscheiden wir zwischen zwei Arten von Schleifen.

Erstens Schleifen, die über eine festgelegte Anzahl von Wiederholungen verfügen, und zweitens sogenannte bedingte Schleifen, die über eine Abbruchbedingung entscheiden, ob sie weiterlaufen oder nicht.

### 6.1 Die while-Schleife: Erst prüfen, dann arbeiten

Die erste Schleife, die wir betrachten, ist die while-Schleife. Sie gehört zu den bedingten Schleifen, das heißt, sie verfügt über eine Abbruchbedingung, mit deren Hilfe sie entscheidet, wann die Schleife verlassen und die normale Programmausführung fortgesetzt wird. Bei der while-Schleife wird die Abbruchbedingung vor dem Eintritt in die Schleife geprüft, das heißt, wenn die Abbruchbedingung zu Beginn der Schleife schon nicht erfüllt ist, dann gibt es auch keinen einzigen Schleifendurchgang.

Formal sieht die while-Schleife so aus:

```
1 while (bedingung)
2 {
3     //Anweisungen, die in der Schleife wiederholt werden
4 }
```

Dabei ist `bedingung` ein Boole'scher Ausdruck. Wenn `bedingung` wahr ist, steigt die Programmausführung in die Schleife ein und wiederholt die Anweisungen in der Schleife so lange, bis `bedingung` falsch ist.

Betrachten wir nun ein erstes triviales Beispiel:

```
1 using System;
2
3 namespace WhileSchleife
4 {
5     class Program
6     {
7         static void Main(string[] args)
8     }
```

## 6.1 Die while-Schleife: Erst prüfen, dann arbeiten

```

9     int zaehler = 0;
10
11    bool machWeiter = true;
12    while(machWeiter)
13    {
14        zaehler = zaehler + 1;
15
16        Console.WriteLine($"Ich zähle: {zaehler}");
17
18        if (zaehler == 10)
19            machWeiter = false;
20    }
21}
22}
23}

```

Wie die Bildschirmausgabe zeigt, handelt es sich um ein Programm, das von 1-10 zählt.

6

```

Microsoft Visual Studio-Debugging-Konsole
Ich zähle: 1
Ich zähle: 2
Ich zähle: 3
Ich zähle: 4
Ich zähle: 5
Ich zähle: 6
Ich zähle: 7
Ich zähle: 8
Ich zähle: 9
Ich zähle: 10

C:\Program Files\dotnet\dotnet.exe (Prozess "18136") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 6.1.1** Ausgabe der while-Schleife

Zu Beginn des Programms wird die Variable `zaehler` mit dem Wert 0 initialisiert. Dann wird die Variable `machWeiter` mit dem Wert `true` initialisiert. Damit sorgen wir dafür, dass die Bedingung für die `while`-Schleife wahr ist und wir auf jeden Fall in die Schleife einsteigen. In der Schleife erhöhen wir die Variable `zaehler` um eins und geben dann den Stand der Variablen `zaehler` am Bildschirm aus. Am Ende der Schleife prüfen wir, ob `zaehler` schon den Wert 10 erreicht hat. Wenn nicht, springt die Programmausführung zum Beginn der Schleife zurück und falls `zaehler` den Wert 10 erreicht hat, wird die Variable `machWeiter` auf `false` gesetzt. Dadurch erreichen wir, dass die Schleife abbricht und der Programmcode, der auf die Schleife folgt, ausgeführt wird. In unserem Fall gibt es keinen Programmcode nach der Schleife, das heißt die Programmausführung wird beendet.

Das obige Programm lässt sich auch etwas kompakter schreiben:

```

1  using System;
2
3  namespace WhileSchleife
4  {
5      class Program

```

## 6 Programmteile wiederholen mit Schleifen

```

6   {
7     static void Main(string[] args)
8     {
9       int zaehler = 0;
10
11      while (zaehler < 10)
12      {
13        zaehler++;
14        Console.WriteLine($"Ich zähle: {zaehler}");
15      }
16    }
17  }
18 }
```

Die Bedingung, ob die Schleife weiterläuft oder nicht, steuern wir nicht mehr mit einer Boole'schen Variablen, sondern schreiben sie direkt in die while-Anweisung, dadurch sparen wir uns die if-Anweisung am Ende der Schleife. Auch das Hochzählen der Variablen zaehler können wir kompakter als zaehler++; schreiben. Die Anweisung zaehler--; würde die Variable um eins erniedrigen. Wollten wir zum Beispiel in Zehnerstufen zählen, könnten wir die kompakte Schreibweise zaehler += 10; verwenden beziehungsweise zaehler -= 10;, um in Zehnerschritten rückwärts zu zählen.

### 6.2 Die do-while-Schleife: Erst arbeiten, dann prüfen

Die do-while-Schleife gehört wie die while-Schleife zu den bedingten Schleifen. Allerdings prüft sie die Abbruchbedingung nicht zu Beginn der Schleife, sondern erst am Ende. Dadurch kommt es – unabhängig von der Abbruchbedingung – zu mindestens einem Schleifendurchgang.

Die do-while-Schleife wird wie folgt geschrieben:

```

1 do
2 {
3   //Anweisungen, die in der Schleife ausgeführt werden
4 } while (bedingung);
```

Die Schleife beginnt mit dem Schlüsselwort do, gefolgt von Anweisungen, die in geschweiften Klammern eingeschlossen sind. Danach folgt das Schlüsselwort while und ein Boole'scher Ausdruck als Abbruchbedingung in runden Klammern.

Das Programm aus dem vorherigen Kapitel, das von eins bis zehn zählt, würde mit einer do-while-Schleife formuliert so aussehen:

```

1 using System;
2
3 namespace DoWhileSchleife
4 {
```

```

5   class Program
6   {
7     static void Main(string[] args)
8     {
9       var zaehler = 0;
10      do
11      {
12        zaehler++;
13        Console.WriteLine($"Ich zähle: {zaehler}");
14      } while (zaehler < 10);
15    }
16  }
17 }
```

Im Prinzip funktioniert es genauso wie das Beispielprogramm für die `while`-Schleife. Es sieht auch fast genauso aus. Der einzige Unterschied ist, dass die Abbruchbedingung am Ende der Schleife und nicht am Anfang geprüft wird.

Wenn wir jetzt die Zeile `var zaehler = 0;` durch die Zeile `var zaehler = 1000;` ersetzen, sehen wir den Unterschied zwischen den beiden Schleifen. Die `while`-Schleife des vorherigen Kapitels würde nichts am Bildschirm ausgeben, da die Bedingung `zaehler < 10` nicht erfüllt ist und die Programmausführung somit erst gar nicht in die Schleife einsteigen würde.

Aber führen wir das Programm mit der `do-while`-Schleife und einem Anfangswert von 1000 für die Variable `zaehler` aus, sieht die Ausgabe wie folgt aus:

The screenshot shows a black terminal window titled "Microsoft Visual Studio-Debugging-Konsole". It displays the following text:  
 Ich zähle: 1001  
 C:\Program Files\dotnet\dotnet.exe (Prozess "5812") wurde mit Code "0" beendet.  
 Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
 Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

**Abb. 6.2.1** Ausgabe der do-while-Schleife für `zaehler = 1000`

Zum Einstieg in die Schleife gibt es keine Prüfung. Daher wird die Variable `zaehler` von 1000 auf 1001 erhöht und am Bildschirm ausgegeben. Am Ende der Schleife wird die Abbruchbedingung geprüft und da 1001 nicht kleiner als 10 ist, wird die Schleife verlassen und das Programm beendet.

Am Ende dieses Kapitels möchte ich noch das Thema der unendlichen Schleife besprechen. Eine unendliche Schleife läuft, wie der Name schon sagt, unendlich. Sie wird nie beendet. Das ist vom Programmierer in den allermeisten Fällen auch nicht beabsichtigt. Aber bei einer komplizierten Logik zur Bestimmung der Abbruchbedingung einer Schleife kann es schon mal vorkommen, dass ein Programm ungewollt in eine unendliche Schleife läuft.

Zu Demonstrationszwecken machen wir das mal mit Absicht:

## 6 Programmteile wiederholen mit Schleifen

```
1 using System;
2
3 namespace DieUnendlicheSchleife
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Die unendliche Geschichte...");
10            while(1==1)
11            {
12
13            }
14        }
15    }
16 }
17 }
```

Unsere Bedingung für die `while`-Schleife heißt `1 == 1` und da eins immer gleich eins ist, ist die Bedingung immer wahr und unsere Schleife bricht nie ab.

Starten wir nun das Programm:



Abb. 6.2.2 Ausgabe der Konsolen-App mit einer unendlichen Schleife

Unser Programm läuft und lässt sich nicht so ohne weiteres beenden. Wenn Windows jetzt nur ein Programm gleichzeitig ausführen könnte, wäre unser Computer jetzt blockiert. Aber Windows kann mehrere Programme gleichzeitig ausführen und hat für solche Fälle ein paar Notbremsen eingebaut, die ich hier im Folgenden vorstellen möchte:

### Notbremse 1: Alt F4

Wenn ihr verunglücktes Programm in einem Fenster läuft, dann stellen Sie mit einem Mausklick auf das Fenster sicher, dass es den Fokus hat und drücken Sie die Tastenkombination „Alt F4“, dann beendet Windows das Programm für Sie.

Übrigens schreiben Sie doch mal im Chat eines Onlinespiels „Press Alt F4 to receive a secret item“. Sie werden nicht glauben, wie viele immer noch auf diesen alten Trick hereinfallen.

### Notbremse 2: Das kleine Kreuz rechts oben

Gilt wie „Alt F4“ für Programme, die in einem Fenster laufen. Die meisten Fenster unter Windows haben rechts oben ein kleines Kreuz. Wenn Sie auf dieses Kreuz klicken, wird das Programm beendet.

**Notbremse 3: Task beenden**

Es gibt spezielle Fehler, die man als Programmierer machen kann, die dafür sorgen, dass „Alt F4“ und ein Klick auf das kleine Kreuz rechts oben das Programm nicht beenden. Dann drücken Sie die Tastenkombination „Strg Alt Entf“ und danach auf „Task Manager“. Windows öffnet dann den sog. Task Manager:

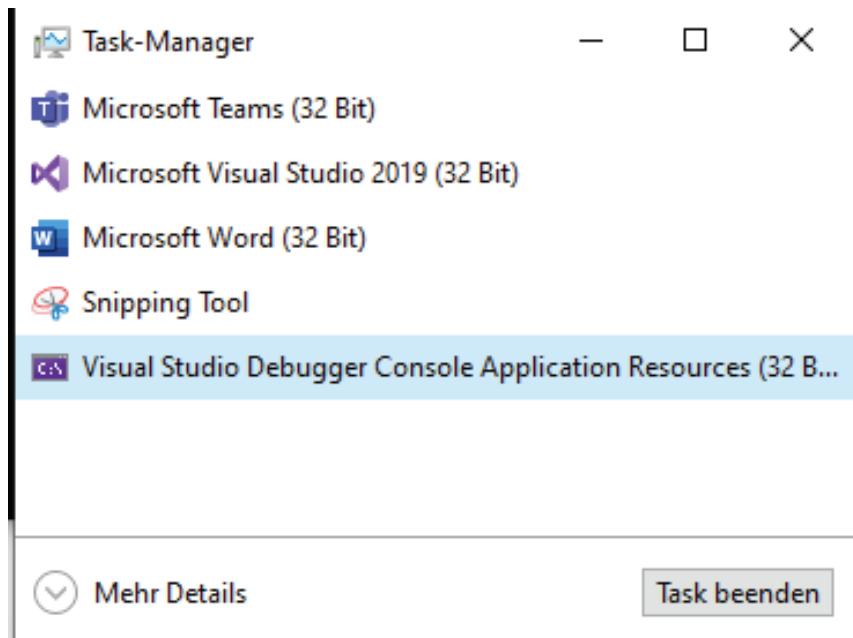


Abb. 6.2.3 Ein Programm mit dem Task-Manager beenden

Markieren Sie Ihr verunglücktes Programm und klicken Sie auf die Schaltfläche „Task beenden“.

**Notbremse 4: Visual Studio**

Wenn Sie, wie für Programmierer üblich, Ihr Programm aus Visual Studio heraus gestartet haben. Können Sie das Programm auch mit Visual Studio wieder beenden. Klicken Sie dazu einfach auf den kleinen roten Knopf in der oberen Toolbar.

## 6 Programmteile wiederholen mit Schleifen

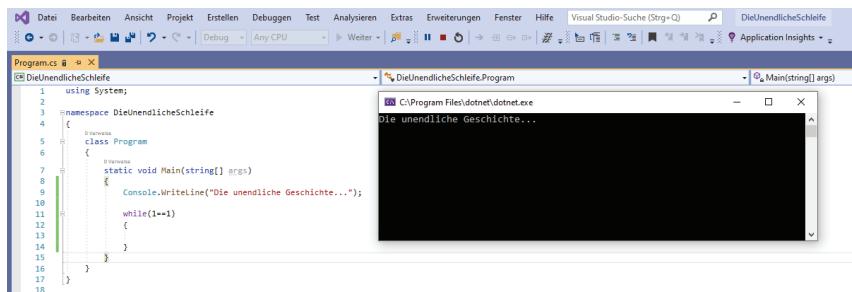


Abb. 6.2.4 Ein Programm mit Visual Studio beenden

### 6.3 Die for-Schleife: Eine feste Anzahl von Wiederholungen

Die while-Schleife und die do-while-Schleife wurden mit einem Programm vorgestellt, das von eins bis zehn zählt. So ein Programm kann man auch mit einer for-Schleife schreiben, bei der man gleich zu Beginn festlegt, dass es exakt 10 Schleifendurchgänge gibt.

```

1  using System;
2
3  namespace ForSchleife
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              for (var i = 1; i <= 10; i++)
10             {
11                 Console.WriteLine($"Ich zähle: {i}");
12             }
13         }
14     }
15 }
```

Die **for**-Schleife beginnt mit dem Schlüsselwort **for**, dann folgen in runden Klammern und mit Semikolons getrennt die drei Parameter der **for**-Schleife. Danach in geschweiften Klammern die Anweisungen, die in der Schleife wiederholt werden.

Der erste Parameter der Schleife ist die Deklaration der sogenannten Laufvariablen. In unserem Fall `var i = 1`. Üblicherweise heißt die Laufvariable `i`, aber das ist nur eine alte Tradition unter Programmierern, Sie können einer Laufvariablen jeden in C# erlaubten Namen geben. Da der Laufvariablen gleich bei der Deklaration ein Wert zugewiesen wird, können wir sie mit `var` deklarieren. In unserem Fall ist sie vom Typ `int`. Es sind aber auch andere Typen erlaubt.

### 6.3 Die for-Schleife: Eine feste Anzahl von Wiederholungen

Der zweite Parameter der `for`-Schleife ist die Bedingung, in unserem Fall `i <= 10`. Solange die Bedingung wahr ist, läuft die Schleife weiter.

Der dritte und letzte Parameter der Schleife ist die Manipulation der Laufvariablen, in unserem Beispiel `i++`. Diese Manipulation wird am Ende eines jeden Schleifendurchgangs wiederholt. Bei unserem Beispiel wird die Laufvariable nach jedem Durchgang um eins erhöht. In der Schleife geben wir die Laufvariable am Bildschirm aus. Damit haben wir wieder ein Programm, das von eins bis zehn zählt, diesmal aber mit einer `for`-Schleife realisiert.

Schleifen können auch ineinander geschachtelt werden. Dazu möchte ich Ihnen ein kleines Beispiel vorstellen. Wir wollen ein Programm schreiben, das am Bildschirm 10 Zeilen ausgibt und jede Zeile soll zehnmal das Zeichen „\*“ enthalten. Eine simple Lösung wäre ein Programm mit 10 `Console.WriteLine()`-Anweisungen, die jeweils einen String mit zehnmal dem Zeichen „\*“ ausgeben. Stellen Sie sich jetzt mal vor, sie wollen das Programm so verändern, dass es nicht zehnmal 10 Sternchen ausgibt, sondern 15- oder 18-mal oder Sie wollen statt dem Zeichen „\*“ das Zeichen „@“ ausgeben. Da wird die Programmänderung etwas mühsam. In der Softwareentwicklung nennt man dieses Thema „Die Wartbarkeit eines Programms“. Ein Computerprogramm sollte gut wartbar sein. Neue Anforderungen und Änderungen sollten mit möglichst wenig Arbeitsaufwand realisiert werden können. Wenn wir unser obiges Beispiel mit 10 `Console.WriteLine()`-Anweisungen realisieren, dann ist das Programm sehr schlecht wartbar. Mit zwei geschachtelten `for`-Schleifen können wir das Problem wesentlich eleganter und deutlich wartbarer lösen.

```
1  using System;
2
3  namespace ZweiGeschachtelteForSchleifen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var hoehe = 10;
10             var breite = 10;
11             var zeichen = "*";
12             for(var i=1; i<=hoehe; i++)
13             {
14                 for(var j=1;j<=breite;j++)
15                 {
16                     Console.Write(zeichen);
17                 }
18                 Console.WriteLine();
19             }
20         }
21     }
22 }
```

## 6 Programmteile wiederholen mit Schleifen

Zuerst deklarieren wir die Variablen `hoehe`, `breite` und `zeichen` und initialisieren sie mit den Werten 10, 10 und „\*“. Dann beginnen wir die äußere Schleife, wobei die Laufvariable von 1 bis 10 läuft. Die äußere Schleife führt zehnmal die innere Schleife aus und bei jedem Durchgang wird nach der inneren Schleife `Console.WriteLine()` ausgeführt, diesmal übergeben wir an die Methode `Console.WriteLine` keinen `string`, daher wird an dieser Stelle nur ein Zeilenumbruch erzeugt.

Die innere Schleife läuft ebenfalls von 1 bis 10 und gibt bei jedem Durchgang einmal unsere Variable `Zeichen`, also „\*“, aus. Die Ausgabe erfolgt mit der Methode `Console.Write()` die nur den übergebenen String und im Gegensatz zu `Console.WriteLine()` keinen Zeilenumbruch am Ende ausgibt.

Wenn wir unser Sternenviereck breiter oder höher ausgeben wollen, genügt es, der Variablen `breite` oder der Variablen `hoehe` einen anderen Wert zu zuweisen. Wollen wir anstatt des Sterns ein anderes Zeichen verwenden, können wir einfach die Zuweisung der Variablen `zeichen` anpassen. Damit ist unser Programm sehr einfach wartbar.

### 6.4 Die `foreach`-Schleife läuft über alles

Die letzte Schleife, die ich vorstellen möchte, ist die `foreach`-Schleife. Allerdings kann man die `foreach`-Schleife nur vernünftig erklären, wenn man dabei sogenannte Listentypen zur Hilfe nimmt. Deshalb werde ich den einfachsten Listentyp, das Array an dieser Stelle, kurz vorstellen und in einem späteren Kapitel detaillierter erläutern.

Stellen Sie sich vor, Sie wollen die Namen der Wochentage in einem Programm speichern. Natürlich könnten Sie das mit sieben String-Variablen erledigen. Für jeden Wochentag eine. Ein Array ist eine Möglichkeit, mit einer Deklaration mehrere Variablen des gleichen Typs zu deklarieren. Für unsere Wochentage sieht das wie folgt aus:

```
1 string[] wochentage = { "Montag", "Dienstag", "Mittwoch",
2 "Donnerstag", "Freitag", "Samstag", "Sonntag"};
```

Nach der Angabe des Variablentyps `string` folgen zwei eckige Klammern, dann der Name der Variablen und als Wert weisen wir durch Kommata getrennte String-Literale, die in geschweiften Klammern eingeschlossen werden, zu. Auf die einzelnen Elemente unseres Arrays können wir mit einem numerischen Index zugreifen. Zum Beispiel können wir „Montag“ mit der folgenden Anweisung am Bildschirm ausgeben.

```
1 Console.WriteLine(wochentage[0]);
```

Wir greifen auf unser Array mit seinem Namen, gefolgt von einem Index in eckigen Klammern, zu. In C# hat das erste Element eines Arrays immer den Index 0.

Der Index muss nicht zwingend in Form eines Integer-Literals angegeben werden, sondern kann auch in Form einer anderen Variablen angegeben werden. Für den Index kann zum Beispiel auch die Laufvariable einer `for`-Schleife verwendet werden. Damit können wir unser vollständiges Array mit Hilfe einer `for`-Schleife am Bildschirm ausgeben.

```
1  using System;
2
3  namespace Arrays
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              string[] wochentage = { "Montag", "Dienstag",
10                         "Mittwoch", "Donnerstag", "Freitag", "Samstag",
11                         "Sonntag"};
12              for (var i = 0; i < wochentage.Length; i++)
13              {
14                  Console.WriteLine(wochentage[i]);
15              }
16          }
17      }
18 }
```

Die `for`-Schleife läuft solange die Laufvariable `i` kleiner als `wochentage.Length` ist. `wochentage.Length` gibt die Anzahl der Elemente des Arrays wieder. Bei unserem Array ist das die Sieben. Beachten Sie bitte, dass wir damit unsere Laufvariable von 0 bis 6 laufen lassen. Das ist nötig, da wir die Laufvariable direkt als Index zum Zugriff auf das Array verwenden und die Zählung bei Arrays bei 0 beginnt.

Jetzt haben wir genug Rüstzeug, um die `foreach`-Schleife zu erklären. Mit der `foreach`-Schleife können wir direkt über ein Array laufen und die Laufvariable enthält bei jedem Durchgang das nächste Element des Arrays. Das obige Programm sieht mit einer `foreach`-Schleife formuliert so aus:

```
1  using System;
2
3  namespace Arrays
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              string[] wochentage = { "Montag", "Dienstag",
10                         "Mittwoch", "Donnerstag", "Freitag", "Samstag",
11                         "Sonntag"};
12              foreach(var wochentag in wochentage)
13              {
14                  Console.WriteLine(wochentag);
15              }
16      }
17 }
```

## 6 Programmteile wiederholen mit Schleifen

```
16     }
17 }
18 }
```

Die Schleife beginnt mit Schlüsselwort „foreach“, dann folgen in runden Klammern die Parameter der Schleife. Der erste Parameter ist die Deklaration der Laufvariablen `var wochentag`, dann folgt das Schlüsselwort `in` und danach der zweite Parameter, also die Struktur, die durchlaufen werden soll, nämlich das Array `wochentage`. Beim ersten Schleifendurchgang hat die Laufvariable `wochentag` den Wert „Montag“, beim zweiten Durchgang den Wert „Dienstag“ und so weiter. Im Prinzip ist die `foreach`-Schleife ein Stück syntactic sugar, welches das Durchlaufen einer Struktur mit einer `for`-Schleife erleichtert.

Was passiert eigentlich, wenn wir in einer Schleife die Elemente eines Arrays überschreiben wollen? Probieren wir es doch einfach mal mit einer `foreach`-Schleife aus. Das folgende Programm läuft mit einer `foreach`-Schleife über ein Array von Wochentagen und schreibt die Nummer des Wochentags vor den Namen des Wochentags im jeweiligen Element des Arrays.

```
1 using System;
2
3 namespace ForeachSchleife
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var wochentage = new string[] { "Montag", "Dienstag",
10             "Mittwoch", "Donnerstag", "Freitag", "Samstag",
11             "Sonntag" };
12             var i = 0;
13             foreach (var wochentag in wochentage)
14             {
15                 i++;
16                 wochentag = $"{i} {wochentag}";
17                 Console.WriteLine(wochentag);
18             }
19         }
20     }
21 }
```

Dieses Programm können wir nicht einmal starten, denn der Compiler erklärt uns schon, dass Laufvariablen von `foreach`-Schleifen nicht schreibbar sind. Mit einer `for`-Schleife dagegen sollte es funktionieren.

```
1 using System;
2
3 namespace ForeachSchleife
4 {
5     class Program
```

## 6.5 Feintuning von Schleifen mit break und continue

```

6      {
7          static void Main(string[] args)
8          {
9              var wochentage = new string[] { "Montag", "Dienstag",
10                 "Mittwoch", "Donnerstag", "Freitag", "Samstag",
11                 "Sonntag" };
12
13             for (var i=0; i<wochentage.Length; i++)
14             {
15                 wochentage[i] = ${i+1} {wochentage[i]};
16                 Console.WriteLine(wochentage[i]);
17             }
18         }
19     }

```

```

Microsoft Visual Studio-Debugging-Konsole
1 Montag
2 Dienstag
3 Mittwoch
4 Donnerstag
5 Freitag
6 Samstag
7 Sonntag
C:\Users\rober\source\repos\VSharpBuch\ForeachSchleife\ForeachSchleife\bin\Debug\netcoreapp3.1\ForeachSchleife.exe (Prozess "8644") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

6

Abb. 6.4.1 Ausgabe des überschriebenen Arrays

## 6.5 Feintuning von Schleifen mit break und continue

Bei allen in C# unterstützten Schleifenarten kann mit den Anweisungen `break` und `continue` ein Feintuning vorgenommen werden, das meistens zur Verbesserung der Ausführungsgeschwindigkeit von Programmen benutzt wird. Stellen Sie sich ein Programm vor, das eine Textdatei nach einem bestimmten Wort durchsucht. So ein Programm liest die Datei, zerlegt die Datei in Zeilen und Wörter und speichert die einzelnen Wörter in einem `string`-Array. Wie man das Einlesen der Datei programmiert, werden wir in einem späteren Kapitel lernen. Zum Zerlegen der Datei in Zeilen und Wörter liefert uns die Klasse `String` eine einfache, aber sehr praktische Methode. Das Durchsuchen erledigen wir mit zwei ineinander geschachtelten `for`-Schleifen. Ohne Feintuning sieht das Programm so aus:

```

1 using System;
2
3 namespace FeinTuningVonSchleifen
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var text = @"Das Buch C# Kompendium ist ein Buch
10                zum Erlernen von C#. Geschrieben hat

```

## 6 Programmteile wiederholen mit Schleifen

```

11         das Buch Robert Schiefele.";
12
13     var suchbegriff = "Buch";
14
15     var wortGefunden = false;
16     var wortIstInZeile = 0;
17     var wortIstAnPosition = 0;
18
19     var zeilen = text.Split("\r\n");
20
21     for(var i = 0; i<zeilen.Length; i++)
22     {
23         var woerter = zeilen[i].Split(" ");
24         for(var j = 0; j<woerter.Length; j++)
25         {
26             if (wortGefunden == false)
27             {
28                 if (woerter[j] == suchbegriff)
29                 {
30                     wortIstInZeile = i + 1;
31                     wortIstAnPosition = j + 1;
32                     wortGefunden = true;
33                 }
34             }
35         }
36     }
37
38     Console.WriteLine($"Das Wort {suchbegriff} befindet
39     sich in Zeile: {wortIstInZeile} an Position:
40     {wortIstAnPosition}");
41 }
42 }
43 }
```

In unserem Beispielprogramm gehen wir davon aus, dass wir die eingelesene Textdatei in der String-Variablen `text` und den vom Benutzer eingegebenen Suchbegriff in der String-Variablen `suchbegriff` finden.

Des Weiteren verwenden wir die Hilfsvariablen `wortGefunden`, `wortIstInZeile` und `wortIstAnPosition`. Die Variable `wortGefunden` ist ein sogenanntes „Flag“. Dieses ist vom Typ `bool` und wir speichern darin den Zustand, ob wir den Suchbegriff bereits gefunden haben oder noch nicht. Die int-Variablen `wortIstInZeile` wird mit dem Wert 0 initialisiert und wir speichern in ihr die Zeilennummer der Zeile, in der wir den Suchbegriff zuerst finden. In der int-Variablen `wortIstAnPosition`, die ebenfalls mit dem Wert 0 initialisiert wird, speichern wir die Positionsnummer innerhalb der Zeile, in der der Suchbegriff gefunden wird.

Dann zerlegen wir den eingelesenen Text mit der Anweisung:

```
1 var zeilen = text.Split("\r\n");
```

Da die Variable `text` vom Typ `String` ist, können wir die Methode `Split` verwenden. Dieser Methode übergeben wir das Literal „\r\n“, welches einen Zeilenumbruch darstellt. Damit weisen wir die Methode an, die String-Variablen `text` immer dann, wenn ein Zeilenumbruch vorkommt, aufzuspalten. Die Methode `Split` gibt ihr Ergebnis in Form eines String-Arrays zurück. Die einzelnen Elemente des Arrays enthalten dann die einzelnen Zeilen der Variablen `text`.

Jetzt können wir mit der äußeren for-Schleife unseres Programms über das String-Array `zeilen` laufen. Für die Laufvariable vergeben wir den Namen `i`.

Die äußere Schleife beginnt mit der Anweisung

```
1 var woerter = zeilen[i].Split(" ");
```

Mit dem Ausdruck `zeilen[i]` greifen wir auf das jeweilige Element des String-Arrays `zeilen` zu und da wir auf ein String-Array zugreifen, hat ein Element des Arrays den Typ `String` und wir können hier wieder die Methode `Split` verwenden. Diesmal spalten wir den String an seinen Leerzeichen auf und wir erhalten als Ergebnis das String-Array `woerter`, welches die einzelnen Wörter einer Zeile enthält.

Damit können wir in die innere Schleife einsteigen. Sie läuft über das Array `woerter` und verwendet die Laufvariable `j`. In der inneren Schleife überprüfen wir mit dem Flag `wortGefunden`, ob wir den Suchbegriff schon gefunden haben. Ist der Suchbegriff noch nicht gefunden, überprüfen wir mit dem Ausdruck `woerter[j] == suchbegriff`, ob das Element des Arrays `woerter`, das dem aktuellen Durchgang der inneren Schleife entspricht, der Suchbegriff ist. Wenn ja, dann haben wir das gesuchte Wort gefunden und setzen die Variable `wortIstInZeile` auf `i+1`, damit die Zeilenzählung bei 1 und nicht bei 0 beginnt. Die Variable `wortIstAnPosition` setzen wir auf `j+1` und speichern so die Positionsnummer des gesuchten Worts. Auch hier beginnt die Zählung der Positionsnummer mit 1.

Wenn unsere Schleifenkonstruktion fertig gelaufen ist, geben wir das Ergebnis mit folgender Anweisung am Bildschirm aus:

```
1 Console.WriteLine($"Das Wort {suchbegriff} befindet sich in Zeile:
2 {wortIstInZeile} an Position: {wortIstAnPosition}");
```

Unser Beispielprogramm durchsucht nur eine Datei und diese Datei ist auch noch sehr klein, daher liefert es auch das Ergebnis sofort. Stellen wir uns jetzt mal vor, unsere Wortsuche ist ein Teil eines größeren Programms, das nicht nur einzelne Textdateien, sondern eine ganze Verzeichnisstruktur mit mehreren Tausend Textdateien durchsucht. Und stellen wir uns zudem vor, dass eine Textdatei im Schnitt aus 10.000 Wörtern besteht. Da kann es dann schon eine Weile dauern, bis wir ein Ergebnis bekommen.

## 6 Programmteile wiederholen mit Schleifen

Unser Programm hat unter diesem Gesichtspunkt einen kleinen Schönheitsfehler. Es läuft nämlich weiter über alle Wörter und alle Zeilen, auch wenn es das gesuchte Wort schon gefunden hat, und wir uns nur für das erste Vorkommen des gesuchten Worts interessieren. Das heißt, unser Programm macht sich jede Menge unnötige Arbeit. Je mehr Textdateien wir durchsuchen, desto mehr summiert sich diese unnötige Arbeit zu unnötiger Wartezeit für den Benutzer.

Am einfachsten wäre es, wenn wir - sobald wir das gesuchte Wort gefunden haben - unsere Schleifenkonstruktion einfach verlassen und das Ergebnis ausgeben könnten. Genau das können wir mit der Anweisung `break` erreichen. Daher ändern wir unser Suchprogramm etwas ab:

```
1  using System;
2
3  namespace FeinTuningVonSchleifen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var text = @"Das Buch C# Das Kompendium ist ein Buch
10             zum Erlernen von C#. Geschrieben hat
11             das Buch Robert Schiefele.";
12
13             var suchbegriff = "Buch";
14
15             var wortGefunden = false;
16             var wortIstInZeile = 0;
17             var wortIstAnPosition = 0;
18             var zeilen = text.Split("\r\n");
19
20             for(var i = 0; i<zeilen.Length; i++)
21             {
22                 var woerter = zeilen[i].Split(" ");
23                 for(var j = 0; j<woerter.Length; j++)
24                 {
25                     if (wortGefunden == false)
26                     {
27                         if (woerter[j] == suchbegriff)
28                         {
29                             wortIstInZeile = i + 1;
30                             wortIstAnPosition = j + 1;
31                             wortGefunden = true;
32                             break;
33                         }
34                     }
35                 }
36                 if (wortGefunden == true)
37                     break;
38             }
39
40             Console.WriteLine($"Das Wort {suchbegriff} befindet
41             sich in Zeile: {wortIstInZeile} an Position:
```

## 6.5 Feintuning von Schleifen mit break und continue

```
42         {wortIstAnPosition}");  
43     }  
44 }  
45 }
```

In der inneren Schleife führen wir, sobald wir den Suchbegriff gefunden haben, die Anweisung „break“ durch, damit beenden wir die innere Schleife und unser Programm muss die aktuelle Zeile nicht mehr fertig durchsuchen. Wenn die innere Schleife beendet ist, überprüfen wir mit der Bedingung `wortGefunden == true`, ob wir den Suchbegriff bereits gefunden haben. Wenn ja, dann beenden wir auch die äußere Schleife mit `break` und geben das Ergebnis am Bildschirm aus.

Jetzt ändern wir den Zweck unseres Programms etwas ab. Diesmal wollen wir nicht das erste Vorkommen eines Wortes suchen, sondern wir wollen die Anzahl der Wörter in einem Text zählen. Zuerst betrachten wir das Programm wieder ohne Feintuning.

```
1  using System;  
2  
3 namespace FeinTuningVonSchleifen2  
4 {  
5     class Program  
6     {  
7         static void Main(string[] args)  
8         {  
9             var text = @"Die Woche hat sieben Tage.  
10            Die Tage der Woche heißen:  
11            Montag  
12            Dienstag  
13            Mittwoch  
14            Donnerstag  
15            Freitag  
16            Samstag  
17            und Sonntag.";  
18  
19             var anzahlWorter = 0;  
20  
21             var zeilen = text.Split("\r\n");  
22             foreach(var zeile in zeilen)  
23             {  
24                 var woerter = zeile.Split(" ");  
25                 foreach(var wort in woerter)  
26                 {  
27                     anzahlWorter++;  
28                 }  
29             }  
30  
31             Console.WriteLine($"Der Text besteht aus  
32             {anzahlWorter} Wörtern.");  
33         }  
34     }  
35 }
```

## 6 Programmteile wiederholen mit Schleifen

Die eingelesene Textdatei erwarten wir wieder in der String-Variablen `text`. Dann deklarieren wir die Hilfsvariable `anzahlWorter` vom Typ `int` und initialisieren sie mit dem Wert 0. Mit der Methode `Split` zerlegen wir dann den Text wieder in Zeilen. Diesmal verwenden wir eine `foreach`-Schleife, um über alle Zeilen zu laufen. In der inneren Schleife zerlegen wir zunächst die aktuelle Zeile in Wörter und laufen dann ebenfalls mit einer `foreach`-Schleife über alle Wörter. In der inneren Schleife erhöhen wir bei jedem Durchgang die Hilfsvariable `anzahlWorter` um eins. Damit erledigen wir das eigentliche Zählen. Wenn alle Schleifendurchgänge erledigt sind, können wir das Ergebnis am Bildschirm ausgeben.

Aber auch hier können wir mit etwas Feintuning die Laufzeit des Programms wieder etwas optimieren:

```
1  using System;
2
3  namespace FeinTuningVonSchleifen2
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var text = @"Die Woche hat sieben Tage.
10             Die Tage der Woche heißen:
11             Montag
12             Dienstag
13             Mittwoch
14             Donnerstag
15             Freitag
16             Samstag
17             und Sonntag.";
18
19             var anzahlWorter = 0
20             var zeilen = text.Split("\r\n");
21
22             foreach(var zeile in zeilen)
23             {
24                 if (zeile.IndexOf(" ") < 0)
25                 {
26                     anzahlWorter++;
27                     continue;
28                 }
29                 var woerter = zeile.Split(" ");
30                 foreach(var wort in woerter)
31                 {
32                     anzahlWorter++;
33                 }
34             }
35             Console.WriteLine($"Der Text besteht aus
36             {anzahlWorter} Wörtern.");
37         }
38     }
39 }
```

## 6.6 Übungsaufgabe: Programmsteuerung mit Schleifen

In der inneren Schleife verwenden wir die Methode `IndexOf()` der Klasse `String`. Ihr übergeben wir ein Leerzeichen als Literal. Die Methode `IndexOf()` gibt uns die Position des übergebenen Strings innerhalb eines anderen Strings zurück. Auch hier beginnt die Zählung wieder bei 0. Enthält unser String den übergebenen String gar nicht, so gibt `IndexOf()` den Wert -1 zurück. Damit können wir mit der Bedingung `zeile.IndexOf(" ") < 0` überprüfen, ob die aktuelle Zeile ein Leerzeichen enthält. Enthält sie kein Leerzeichen, besteht die Zeile nur aus einem einzigen Wort. Das heißt wir können die int-Variable `anzahlWorter` sofort um eins erhöhen und mit der Anweisung `continue` lassen wir den Rest des Schleifendurchgangs aus und beginnen gleich den nächsten Durchgang.

## 6.6 Übungsaufgabe: Programmsteuerung mit Schleifen

Schleifen gehören in der Programmierung mit zu den wichtigsten Techniken. Daher ist es für Programmierer essenziell, dieses Thema zu beherrschen. Deswegen machen wir hier auch zum Thema „Schleifen“ eine längere Übung.

### Übungsaufgabe 1:

Erstellen Sie in Visual Studio eine neue Konsolen-App und mit folgendem Programmcode für die Datei „Program.cs“.

```
1  using System;
2
3  namespace Uebung_6_6_Aufgabe1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int zahl = 0;
10             Console.WriteLine("Geben Sie eine ganze Zahl ein:");
11             do
12             {
13                 Console.WriteLine($"5 geteilt durch {zahl} = {(5
14                     / zahl)} Rest {5 % zahl}");
15
16                 Console.WriteLine("Geben Sie eine ganze Zahl
17                     ein:");
18             }
19             while ( (zahl = int.Parse(Console.ReadLine())) !=
20                     0);
21             Console.WriteLine("Durch 0 kann nicht geteilt
22                     werden!");
23         }
24     }
25 }
```

## 6 Programmteile wiederholen mit Schleifen

Wenn Sie das Programm starten, erhalten Sie die Fehlermeldung „System.DivideByZeroException“.

The screenshot shows the Visual Studio IDE with the code editor displaying a C# program named 'Uebung\_6\_6\_Aufgabe1'. The code contains a main method that reads user input for a number, divides 5 by that number, and prints the result. A 'try' block is present, but it does not handle the division by zero case. A tooltip from the debugger highlights the line 'int zahl = 0;' with the note 'Unbehandelte Ausnahme' (Unhandled Exception) and 'System.DivideByZeroException: "Attempted to divide by zero."'. The status bar at the bottom indicates 'Keine Probleme gefunden' (No problems found).

```

1 using System;
2
3 namespace Uebung_6_6_Aufgabe1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             int zahl = 0;
10            Console.WriteLine("Geben Sie eine ganze Zahl ein:");
11            do
12            {
13                Console.WriteLine($"5 geteilt durch {zahl} = ({5 / zahl}) Rest {5 % zahl}");
14                Console.WriteLine("Geben Sie eine ganze Zahl ein:");
15            }
16            while ((zahl = int.Parse(Console.ReadLine())) != 0);
17            Console.WriteLine("Durch 0 kann nicht geteilt werden!");
18        }
19    }
20}
21
22

```

**Abb. 6.6.1** DivideByZeroException nach Programmstart

Korrigieren Sie das Programm, in dem Sie nur den Schleifentyp wechseln.

### Übungsaufgabe 2:

Schreiben Sie ein Programm, das eine Textzeile nach der anderen von der Tastatur einliest. Wenn der Benutzer die Textzeile „Fertig!“ eingibt, soll das Programm eine Zeile mit 10 Sternchen ausgeben und dann den gesamten eingegebenen Text, aber ohne die Zeile „Fertig!“.

### Übungsaufgabe 3

Geben Sie für die jeweilige Anwendung den geeigneten Schleifentyp an.

1. Eine Schleife, die über ein ganzes Array laufen soll.
2. Eine Schleife, die mindestens einmal durchlaufen werden soll, unabhängig von der Abbruchbedingung.
3. Eine Schleife, die genau zehnmal durchlaufen werden soll.
4. Eine Schleife, die gar nicht durchlaufen werden soll, wenn Ihre Bedingung nicht erfüllt ist.

**Übungsaufgabe 4:**

Ein Array ist wie folgt deklariert:

```
1 int[] zahlen = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Schreiben Sie ein Programm, das mit Hilfe dieses Arrays ein Dreieck aus „\*“-Zeichen am Bildschirm ausgibt.

## 6 Programmteile wiederholen mit Schleifen

### Musterlösung für Übungsaufgabe 1:

```
1  namespace Uebung_6_6_Aufgabe1
2  {
3      class Program
4      {
5          static void Main(string[] args)
6          {
7              int zahl = 0;
8              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
9              while ((zahl = int.Parse(Console.ReadLine())) != 0)
10             {
11                 Console.WriteLine($"5 geteilt durch {zahl} = {(5
12 / zahl)} Rest {5 % zahl}");
13                 Console.WriteLine("Geben Sie eine ganze Zahl
14 ein:");
15             }
16             Console.WriteLine("Durch 0 kann nicht geteilt
17 werden!");
18         }
19     }
20 }
```

**Musterlösung für Übungsaufgabe 2:**

```
1  using System;
2
3  namespace Uebung_6_6_Aufgabe2
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              //Hier speichern wir den vollständigen
10             //eingegebenen Text
11             var text = "";
12             //Das Fertig-Kommando legen wir uns auch in eine
13             //Variable, damit wir es nur an einer Stelle
14             //ändern müssen falls nötig
15             var fertig = "Fertig!";
16             //Hier speichern wir die aktuelle eingegebene
17             //Zeile
18             var eingabe = "";
19
20             //Die Schleife läuft, bis der Benutzer
21             //das Fertig-Kommando eingibt
22             while((eingabe = Console.ReadLine()) != fertig)
23             {
24                 //Die neue Zeile an den Text anhängen
25                 text += eingabe;
26
27                 //Einen Zeilenumbruch an den Text anhängen
28                 text += "\r\n";
29             }
30
31             Console.WriteLine("*****");
32
33             //Text in Zeilen splitten
34             var zeilen = text.Split("\r\n");
35
36             //Die Schleife läuft über alle Zeilen
37             foreach(var zeile in zeilen)
38             {
39                 //Wenn wir die Zeile mit dem Fertig-Kommando
40                 //noch nicht erreicht haben
41                 if(zeile != fertig)
42                 {
43                     //dann geben wir die Zeile aus
44                     Console.WriteLine(zeile);
45                 }
46             }
47         }
48     }
49 }
```

## 6 Programmteile wiederholen mit Schleifen

### Musterlösung für Übungsaufgabe 3:

1. Die `foreach`-Schleife
2. Die `do-while`-Schleife
3. Die `for`-Schleife
4. Die `while`-Schleife

**Musterlösung für Übungsaufgabe 4:**

```
1 using System;
2
3 namespace Uebung_6_6_Aufgabe4
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             //Ein Array mit Zahlen von 1-9
10            int[] zahlen = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
11
12            //Schleife läuft über das gesamte Array
13            foreach(var zahl in zahlen)
14            {
15                //Die Schleife hat „zahl“ Durchgänge
16                for(var i = 0; i < zahl; i++)
17                {
18                    Console.Write("*");
19                }
20
21                //Ein Zeilenumbruch am Ende von jedem Durchgang
22                //der äußeren Schleife
23                Console.WriteLine();
24            }
25        }
26    }
27 }
```

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 7

# Strukturierte Daten in C#

In diesem Kapitel geht es um Datenstrukturen. Die Datentypen, die ich Ihnen bereits vorgestellt habe, sind sogenannte primitive Datentypen. Eine Variable mit einem primitiven Datentyp enthält nur einen Wert: eine ganze Zahl, eine Gleitkommazahl, einen Text, ein Zeichen oder einen Boole'schen Wert. Jetzt werden wir Datentypen kennenlernen, bei denen mehrere Werte in einer Variablen gespeichert werden.

### 7.1 Arrays: Eine Variable für viele Werte

Die erste Datenstruktur, das Array, haben wir schon eingeführt, um die foreach-Schleife zu behandeln. Jetzt werden wir das Array detaillierter betrachten. Ein Array beschreibt eine endliche Menge von Variablen eines bestimmten Variablen Typs. Das muss nicht zwangsläufig ein primitiver Typ sein. Aber da wir bisher nur primitive Typen kennen, beschränken wir uns vorerst auf Arrays mit primitiven Typen. Ein Array wird deklariert, indem man dem Typ zwei eckige Klammern anhängt, dem das Array zugrunde liegt.

```
1 int[] zahlen;
```

Mit einem so deklarierten Array können Sie allerdings noch nichts anfangen. Damit Sie es verwenden können, müssen Sie es zuerst initialisieren. Das heißt, Sie müssen festlegen, wie viele Elemente es umfassen soll.

```
1 zahlen = new int[6];
```

Damit haben Sie dem Array `zahlen` sechs Speicherplätze für sechs `int`-Variablen zugewiesen. Jede dieser `int`-Variablen hat zunächst den Standardwert für `int`-Variablen, also 0. Natürlich können Sie das Array auch schon bei der Deklaration initialisieren.

```
1 int[] zahlen = new int[6];
```

Sie können bei der Deklaration auch gleich alle sechs `int`-Variablen mit Werten belegen.

```
1 int[] zahlen = { 1, 2, 3, 4, 5, 6 };
```

Falls sich jetzt denken, Sie könnten die obige Programmzeile mit dem Schlüsselwort `var` vereinfachen, muss ich Sie leider enttäuschen. Wenn Sie `int []` durch `var` ersetzen,

## 7 Strukturierte Daten in C#

zen, erhalten Sie einen Fehler. Allerdings gibt es doch eine Möglichkeit, wie Sie Arrays mit var deklarieren können. Dazu müssen Sie die Zuweisung geringfügig anpassen.

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

Der Zugriff auf ein Array erfolgt mit einem sogenannten Indexer. Das ist eine Zahl, die Sie nach dem Variablenamen in eckige Klammern schreiben.

```
1 var zahl = zahlen[1];
```

Damit weisen Sie der int-Variablen zahl das zweite Element des Arrays zu. Sie erinnern sich, die Zählung bei Arrays beginnt bei 0. Das erste Element wäre somit zahlen[0].

Analog zu einem lesenden Zugriff funktioniert ein schreibender Zugriff auf ein Array.

```
1 zahlen[0] = 28;
```

Hier wird dem ersten Element des Arrays die Zahl 28 zugewiesen.

Wie Sie schon bemerkt haben, muss man bei einem Array vor der Deklaration wissen, wie viele Elemente es enthalten soll. Das funktioniert auch dynamisch.

```
1 var anzahl = 5;
2 var zahlen = new int[anzahl];
```

Diese beiden Programmzeilen stellen für C# kein Problem dar. Wie gesagt, die Anzahl der Elemente muss vor der Deklaration bekannt sein, nicht sofort beim Programmstart. Was aber, wenn sich die Anzahl der Elemente während eines Programms dynamisch verändert? Auch dafür gibt es in C# eine Lösung.

```
1 Array.Resize(ref zahlen, 9);
```

Die Methode Resize der Klasse Array kann die Größe eines Arrays verändern. Dazu werden der Methode das zu vergrößernde Array und ein Wert für die neue Größe des Arrays übergeben. Das Array muss als sogenannter reference-Parameter übergeben werden. Das heißt, vor den Parameter müssen Sie das Schlüsselwort ref stellen. Was es mit reference-Parametern auf sich hat, erfahren Sie in einem späteren Kapitel. Beachten Sie dabei: Wenn Sie ein Array verkleinern, sind die Werte der oberen Elemente verloren. Wenn Sie diese in Ihrem Programm später noch benötigen müssen Sie sie in anderen Variablen zwischenspeichern. Allerdings kann ich Ihnen die Verwendung der Methode „Resize“ nur im Notfall empfehlen. Wenn Sie in Ihrem Programm eine Struktur benötigen, die dynamisch anwächst, empfehle ich Ihnen die Verwendung der List-Struktur, welche wir in einem späteren Kapitel kennenlernen.

## 7.2 Mehrdimensionale Arrays

Im vorigen Kapitel habe ich erwähnt, dass die Elemente eines Arrays nicht zwingend aus primitiven Variablen bestehen müssen. Man kann zum Beispiel auch ein Array aus Arrays deklarieren.

```
1 string[,] spielfeld;
```

Wir setzen einfach ein Komma zwischen die eckigen Klammern. Aber auch Arrays aus Arrays beziehungsweise zweidimensionale Arrays müssen zuerst initialisiert werden.

```
1 string[,] spielfeld = new string[3,3];
```

Damit haben wir unserem Array-Speicher für dreimal drei `string`-Variablen zugewiesen. Zweidimensionale Arrays werden wie folgt bei der Deklaration mit Werten belegt:

```
1 var spielfeld = new string[3, 3] { { "#", "#", "#" }, { "#", "#", "#" }, { "#", "#", "#" } };
```

7

In dieser Variablen könnten wir den Zustand eines Tic-Tac-Toe-Spiels speichern. Tic-Tac-Toe ist ein simples Brettspiel mit einem Spielfeld, das aus dreimal drei Feldern besteht. Für ein Schachspiel bräuchten wir ein Array mit acht mal acht Elementen.

Betrachten wir nun ein Beispielprogramm, mit dem wir unser Tic-Tac-Toe-Spielfeld am Bildschirm anzeigen.

```
1 using System;
2
3 namespace Arrays
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9
10             var spielfeld = new string[4, 4] {
11                 { " ", "A", "B", "C" },
12                 { "1", "#", "#", "#" },
13                 { "2", "#", "#", "#" },
14                 { "3", "#", "#", "#" }
15             };
16
17             for(var i=0; i<4; i++)
18             {
19                 for(var j=0; j<4; j++)
20                 {
21                     Console.Write(spielfeld[i, j]);
22                 }
23                 Console.WriteLine();
24             }
25         }
26     }
27 }
```

## 7 Strukturierte Daten in C#

```
25 }  
26 }  
27 }
```

Die Variable `spielfeld` enthält für jedes Feld unseres Spielbretts ein Zeichen, zudem in der ersten Reihe die Buchstaben A-C und in der ersten Spalte die Zahlen 1-3, wobei das Feld links oben ein Leerzeichen enthält. Damit bilden wir - ähnlich wie bei einem Schachbrett - ein Koordinatensystem. Um einen Spielzug zu definieren, zum Beispiel „Spielstein auf B2“, würde man so einen Spielstein in die Mitte des Spielfelds setzen. Zum Start des Spiels enthält jedes Feld das Zeichen „#“. Damit symbolisieren wir, dass kein Spielstein auf das Feld gesetzt wurde. Die Bildschirmausgabe realisieren wir mit zwei ineinander geschachtelten for-Schleifen. Die äußere Schleife läuft mit der Laufvariablen `i` über die erste Dimension des zweidimensionalen Arrays. Die innere Schleife läuft mit der Laufvariablen `j` über die zweite Dimension des Arrays. In der inneren Schleife greifen wir mit dem Ausdruck `spielfeld[i, j]` auf das Spielfeld zu und geben den Zustand des jeweiligen Feldes aus. Nach der inneren Schleife geben wir noch einen Zeilenumbruch aus, um mit der nächsten Zeile beginnen zu können.

Theoretisch haben wir jetzt genug C#-Wissen angesammelt, um ein vollständiges Tic-Tac-Toe-Spiel programmieren zu können. Ich möchte es aber trotzdem an dieser Stelle nicht tun, da am Ende nur Spaghetti-Code herauskommen würde, der als Beispiel für einen schlechten Programmierstil herhalten könnte. Aber nach dem Kapitel über Methoden werden wir ein Tic-Tac-Toe-Spiel Schritt für Schritt in einer Übungsaufgabe durchexerzieren.

### 7.3 Typisierte Listen: Arrays mit Komfort

Die nächste Struktur, die wir betrachten, ist die typisierte Liste. Damit wir eine typisierte Liste verwenden können, müssen wir zu Beginn unseres Programms eine weitere Klassenbibliothek einbinden. Das erledigen wir mit der folgenden `using`-Anweisung.

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

Jetzt können wir unsere erste typisierte Liste deklarieren.

```
1 List<string> wochentage;
```

Die Liste wird mit dem Schlüsselwort `List` deklariert, danach folgt der Basis-Typ der Liste in spitzen Klammern und dann der Variablenname. Wie das Array basiert die Liste auf einem Basis-Typ, in diesem Beispiel ist das der Typ `string`. Es könnte aber jeder andere Typ sein und es muss auch kein primitiver Typ sein. Auch exotische Varianten wie:

```
1 List<string[]> listeAusStringArrays;
```

oder

```
1 List<List<string>> listeAusListenAusStrings;
```

sind erlaubt. Eine Liste muss vor ihrer Verwendung zuerst initialisiert werden:

```
1 List<string> wochentage = new List<string>();
```

Das wäre dann eine leere Liste, also eine Liste mit null Elementen. Eine Liste kann auch bei der Deklaration gleiche Werte erhalten.

```
1 List<string> wochentage = new List<string> { "Montag",
2 "Dienstag", "Mittwoch", "Donnerstag", "Freitag" };
```

Der Zugriff auf eine Liste erfolgt genauso wie auf ein Array.

```
1 var tag = wochentage[0];
```

7

Auch der schreibende Zugriff funktioniert analog zu einem Array.

```
1 wochentage[0] = "Der erste Tag der Woche";
```

Eine Liste können wir - genauso wie ein Array - mit einer for-Schleife oder mit einer foreach-Schleife durchlaufen.

```
1 using System;
2 using System.Collections.Generic;
3
4 namespace TypisierteListe
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             List<string> wochentage = new List<string> {
11                 "Montag", "Dienstag", "Mittwoch", "Donnerstag",
12                 "Freitag" };
13             foreach(var tag in wochentage)
14             {
15                 Console.WriteLine(tag);
16             }
17         }
18     }
19 }
```

Bis auf die Tatsache, dass man bei der Liste keine Größenangabe machen muss, ist noch kein nennenswerter Vorteil gegenüber einem Array zu sehen.

## 7 Strukturierte Daten in C#

Der große Vorteil einer Liste liegt in Ihrer einfachen dynamischen Erweiterbarkeit.

```
1 List<string> wochentage = new List<string> { "Montag",
2 "Dienstag", "Mittwoch", "Donnerstag", "Freitag" };
3 wochentage.Add("Samstag");
```

Mit der Methode „Add“, mit der wir ein Element hinzufügen, hängen wir ein neues Element an die Liste an. Aber die Liste kann noch mehr. Man kann auch mehrere Elemente auf einmal anhängen.

```
1 var arbeitsWoche = new List<string> { "Montag", "Dienstag",
2 "Mittwoch", "Donnerstag", "Freitag" };
3 var volleWoche = new List<string>();
4 volleWoche.AddRange(arbeitsWoche);
5 volleWoche.Add("Samstag");
6 volleWoche.Add("Sonntag");
```

Der Methode AddRange () kann wieder eine Liste übergeben werden, die dann an die ursprüngliche Liste angehängt wird. Die Variable arbeitsWoche enthält dann die Wochentage von „Montag“ bis „Freitag“ und die Variable volleWoche enthält die Wochentage von „Montag“ bis „Sonntag“. Wie Sie sehen, ist die Liste deutlich eleganter als das Array, wenn Sie Strukturen benötigen, die laufend erweitert werden müssen.

Ein absolutes Highlight der Liste ist die Möglichkeit, einzelne Element wieder zu entfernen.

```
1 volleWoche.Remove("Montag");
```

Man muss nur der Methode Remove das Element, das man entfernen möchte, übergeben. In unserem Beispiel bekommen wir dann eine Woche ohne Montag.

Eine Liste kann das gleiche Element mehrfach enthalten. Mit dem Codefragment:

```
1 var arbeitsWoche = new List<string> { "Montag", "Dienstag",
2 "Mittwoch", "Donnerstag", "Freitag" };
3 var volleWoche = new List<string>();
4 volleWoche.AddRange(arbeitsWoche);
5 volleWoche.Add("Samstag");
6 volleWoche.Add("Sonntag");
7 volleWoche.Add("Sonntag");
```

erhalten wir eine Woche mit zwei Sonntagen. Mit der Anweisung:

```
1 volleWoche.Remove("Sonntag");
```

entfernen wir einen Sonntag wieder aus der Liste. Ob damit der erste oder zweite Sonntag in unserer Woche entfernt wird, können wir von außen nicht feststellen. Das hängt davon ab, wie die Entwickler von Microsoft die Liste implementiert haben. Aus

der Sicht der Programmlogik von C# ist es auch unerheblich, welcher der beiden Einträge gelöscht wird. Enthält die Liste nur einen Eintrag mit dem Wert „Sonntag“, so enthält sie nach dem Löschen keinen mehr. Enthält sie mehrere Einträge, so enthält sie nach dem Löschen einen Eintrag weniger.

## 7.4 Das Dictionary, die elegante Listenverwaltung

Ein Dictionary können Sie sich wie eine Liste vorstellen, auf die Sie nicht mit einem Index, also einer ganzen Zahl, zugreifen können, sondern mit einem Schlüssel, der wiederum ein beliebiger Variabtentyp sein kann. Damit hat ein Dictionary zwei Basistypen: einen für den Schlüssel und einen für den im jeweiligen Element zu speichern- dem Wert. Um ein Dictionary zu verwenden, benötigen Sie die gleiche `using`-Anweisung wie für eine Liste.

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

7

Die Deklaration des Dictionary erfolgt mit dem Schlüsselwort „Dictionary“.

```
1 Dictionary<string, string> telefonVerzeichnis;
```

Nach dem Schlüsselwort `Dictionary` folgen in spitzen Klammern und durch ein Komma getrennt die beiden Basistypen des `Dictionary`. Zuerst der Typ für den Schlüssel und dann der Typ für den Wert. Wir benutzen hier für beide den Typ `string`. Das `Dictionary` muss vor seiner Verwendung zuerst initialisiert werden.

```
1 telefonVerzeichnis = new Dictionary<string, string>();
```

Damit hätten wir ein verwendbares `Dictionary`, das leer ist, also null Elemente enthält.

Dem `Dictionary` kann man bei der Deklaration auch gleich einige Elemente zuweisen.

```
1 var telefonVerzeichnis = new Dictionary<string, string>
2 {
3     { "Robert", "12345678" },
4     { "Klaus", "86754321" },
5     { "Dieter", "42548678" }
6 };
```

In diesem `Dictionary` haben wir jetzt die Telefonnummern für die Schlüssel „Robert“, „Klaus“ und „Dieter“ gespeichert. Benötigen wir in einem Programm die Telefonnummer von Robert können wir einfach mit dem Schlüssel Robert auf das `Dictionary` zu- greifen.

```
1 var robertsNummer = telefonVerzeichnis["Robert"];
```

## 7 Strukturierte Daten in C#

Wir greifen - genau wie bei einem Array oder einer Liste – einfach mit einem Schlüssel zu, aber wir verwenden keinen Index. Der Schreibzugriff erfolgt genauso.

```
1 telefonVerzeichnis["Robert"] = "34345656";
```

Mit dieser Zuweisung bekommt Robert eine neue Telefonnummer. Das Dictionary verfügt auch über eine Add-Methode, mit der wir ihm einen neuen Eintrag hinzufügen können.

```
1 telefonVerzeichnis.Add("Thea", "3456453");
```

Der Methode „Add“ übergeben wir den Schlüssel („Thea“) und den Wert („3456453“) für den neuen Eintrag. Übrigens können Sie einem Eintrag, der im Dictionary gar nicht existiert, auch einen Wert zuweisen. C# legt den Eintrag einfach an.

```
1 telefonVerzeichnis["Katia"] = "34534525";
```

Diese Zuweisung legt einem neuen Eintrag für den Schlüssel „Katia“ an und weist ihm gleich den Wert „34534525“ zu. Ein Schlüssel muss in einem Dictionary eindeutig sein, das heißt, es darf keine zwei Einträge mit demselben Schlüssel geben.

```
1 telefonVerzeichnis.Add("Thea", "3456453");
2 telefonVerzeichnis.Add("Thea", "34564456");
```

Das obige Code-Fragment würde zu einem Fehler führen. Allerdings ist der gleiche Wert mit verschiedenen Schlüsseln wiederum erlaubt.

```
1 telefonVerzeichnis.Add("Thea", "3456453");
2 telefonVerzeichnis.Add("Karl", "3456453");
```

Diese beiden Zeilen führen zu keinem Fehler. Wenn Sie wissen wollen, ob ein Schlüssel in einem Dictionary bereits existiert, können Sie das Dictionary mit der Methode ContainsKey() abfragen.

```
1 if (telefonVerzeichnis.ContainsKey("Katia"))
2 {
3     //Code der ausgeführt wird, falls der Schlüssel
4     //Katia im Dictionary existiert.
5 }
```

Der Methode ContainsKey() übergeben Sie den abzufragenden Schlüssel und erhalten den Boole'schen Wert „true“, wenn der Schlüssel existiert, beziehungsweise den Boole'schen Wert „false“, wenn der Schlüssel nicht existiert.

Auf die gleiche Art können Sie ein Dictionary mit der Methode ContainsValue() abfragen, ob ein Wert existiert.

**7.5 Das Tuple: Mehrere verschiedene Variablen in einer Struktur**

```

1 if (telefonVerzeichnis.ContainsValue("1234"))
2 {
3     //Code der ausgeführt wird, falls der Wert
4     //1234 im Dictionary existiert.
5 }
```

Mit der Methode „Remove“ können Sie einen Eintrag aus einem Dictionary entfernen.

```
1 telefonVerzeichnis.Remove("Karl");
```

Wir können auch mit einer `foreach`-Schleife über Dictionary laufen.

```

1 foreach(var element in telefonVerzeichnis)
2 {
3     Console.WriteLine(${element.Key}: ${element.Value});
4 }
```

Das einzelne Element deklarieren hier mit `var`, da wir eigentlich nicht genau wissen, was es für einen Typ hat. Ein Dictionary-Eintrag besteht bekanntermaßen aus einem Schlüssel und einem Wert und die beiden müssen auch nicht zwangsläufig vom Typ `string` sein, das haben wir bei der Deklaration der Variablen `telefonVerzeichnis` so festgelegt. Aber Visual Studio kann uns hier weiterhelfen. Wenn wir in Visual Studio die Maus über das Wort `element` in der `foreach`-Schleife bewegen, zeigt uns Visual Studio in einem Tooltip den Typ von `element` an:

```
1 KeyValuePair<string, string> element
```

Innerhalb der `foreach`-Schleife können wir mit `element.Key` auf den Schlüssel und mit `element.Value` auf den Wert zugreifen.

7

**7.5 Das Tuple: Mehrere verschiedene Variablen in einer Struktur**

In der Mathematik wird eine Kombination von zwei Werten aus der gleichen Grundmenge als Doppel bezeichnet. Wenn es sich um eine Kombination aus drei Werten handelt, nennt man sie Tripel. Bei vier Werten spricht man von einem Quadrupel und bei fünf Werten von einem Quintupel. Wer auf lateinisch zählen kann, der kann die Reihe noch weiterführen. Allgemein spricht man von einem  $n$ -Tupel, wobei  $n$  die Anzahl der kombinierten Werte ist. In C# gibt es einen Variablentyp, der genau dieses mathematische Konstrukt abbildet.

```
1 Tuple<int, int, int> zahlenTripel;
```

In der Variablen `zahlenTripel` können wir jetzt drei Ganzzahlen in einer Variablen speichern. Allerdings können wir die Zahlen nicht einzeln zuweisen. Wir können alle drei Zahlen auf einmal zuweisen. Das wird mit dem sogenannten Konstruktor von

## 7 Strukturierte Daten in C#

Tuple erledigt. Ein Konstruktor ist eine spezielle Methode. Details über den Konstruktor lernen Sie im Kapitel über die Objektorientierte Programmierung.

```
1 zahlenTripel = new Tuple<int, int, int>(1, 2, 3);
```

Der Konstruktor heißt genauso wie unser `Tuple<int, int, int>`. Um einen Konstruktor aufzurufen, müssen wir vor den Konstruktor das Schlüsselwort `new` setzen. Ein Konstruktor bekommt, wie jede andere Methode, seine Parameter in runden Klammern übergeben. In unserem Fall die Werte für die einzelnen Elemente des Tupels. Der Rückgabewert des Konstruktors ist das Tupel selbst.

Der Zugriff auf die einzelnen Elemente des Tupels ist wie folgt möglich:

```
1 var eins = zahlenTripel.Item1;
2 var zwei = zahlenTripel.Item2;
3 var drei = zahlenTripel.Item3;
```

Mit den Namen der Tupel-Variablen - gefolgt von einem Punkt und den Schlüsselwörtern `Item1`, `Item2`, ... bis `Itemn` - können die Elemente eines Tupels gelesen werden. Allerdings kann man auf die Elemente eines Tupels nicht schreibend zugreifen. Dieser Versuch würde zu einem Fehler führen. Man könnte die Variable `zahlenTripel` aber neu zuweisen.

```
1 zahlenTripel = new Tuple<int, int, int>(4, 5, 6);
```

Jetzt hat unser Tupel neue Werte. Ein Tupel kann auch mehr als drei Elemente haben und jedes Element kann auch wieder einen anderen Typ haben.

```
1 Tuple<int, double, string, bool> meinTuple;
```

Die obige Zeile ist zum Beispiel eine gültige Deklaration für ein Tupel.

Seit C# 7.0 gibt es eine wesentlich elegantere Art von Tupel - sogenannte „named tuples“. Wenn wir ein Tupel wie folgt deklarieren:

```
1 (int eineZahl, string einString) modernesTupel;
```

dann liefert uns diese Schreibweise alles, was wir mit Variablen vom Typ `Tuple<...>` bisher nicht tun konnten. Die Elemente des Tupels heißen nicht mehr `Item1`, `Item2` und so weiter, sondern wir können ihnen eigene Namen geben, wie hier `eineZahl` und `einString`. Wir müssen das Tupel auch nicht mehr über einen Konstruktor erzeugen. Das Tupel ist bei der Deklaration schon initialisiert. Wir können auch schreibend und einzeln auf die Elemente des Tupels zugreifen.

```
1 modernesTupel.eineZahl = 42;
2 modernesTupel.einString = "Per Anhalter durch die Galaxis";
```

## 7.6 Übungsaufgaben: Arbeiten mit Arrays, Listen, Dictionaries und Tuples

Man kann so ein benanntes Tupel auch einem Literal zuweisen.

```
1 modernesTupel = (5, "Hallo");
```

Auch eine Deklaration und mit var und einem Literal ist möglich.

```
1 var modernesTupel = (eineZahl: 5, einString: "Hallo");
```

Wenn Sie ein benanntes Tupel mit var deklarieren, können Sie die Namen für die Elemente gefolgt von einem Doppelpunkt im Literal direkt angeben. Wenn Sie bei der Deklaration mit var auf Namen für die Elemente des Tupels verzichten, heißen die Elemente wieder Item1, Item2 und so weiter.

Durch diese Neuerung in C# 7.0 hat das früher eher selten verwendete Tupel bei C#-Programmierern sehr stark an Beliebtheit gewonnen.

7

## 7.6 Übungsaufgaben: Arbeiten mit Arrays, Listen, Dictionaries und Tuples

In dieser Übungsaufgabe werden wir schrittweise ein kleines Programm zur Abfrage eines Telefonverzeichnisses entwerfen.

### Teilaufgabe 1:

Deklarieren Sie eine zusammengesetzte Struktur zur Speicherung von Festnetznummer, Mobilnummer und E-Mail-Adresse für mehrere Personen. Für eine Person speichern Sie dabei nur den Vornamen. Befüllen Sie diese Struktur beim Start des Programms mit ein paar Beispieldaten.

### Teilaufgabe 2:

Erweitern Sie das Programm aus der Teilaufgabe 1 so, dass das Programm in einer Schleife ständig einen Namen vom Benutzer abfragt und in einer Variablen speichert. Wenn der Benutzer „Fertig!“ eingibt, soll sich das Programm beenden.

### Teilaufgabe 3:

Erweitern Sie das Programm aus Teilaufgabe 2. Nachdem das Programm einen Namen vom Benutzer abgefragt hat, soll es überprüfen, ob es für diesen Namen einen Eintrag im Telefonverzeichnis gibt und wenn nicht, soll es den Text „Der Name *name* ist mir nicht bekannt.“ ausgeben, wobei *name* der eingegebene Name ist. Danach soll das Programm fortfahren, Namen abzufragen.

### Teilaufgabe 4

Erweitern Sie das Programm aus Teilaufgabe 3, sodass es den Text „Festnetznummer; Mobil: mobilfunknummer; E-Mail: email“ ausgibt, wenn der eingegebene Name im Telefonverzeichnis gefunden wird. Wobei *festnetznummer* die Festnetznummer, *mobilfunknummer* die Mobilfunknummer und *email* die E-Mail-Adresse des im Telefonverzeichnis gefundenen Namens ist.

### Musterlösung zu Teilaufgabe 1

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TelefonVerzeichnis
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              var telefonVerzeichnis = new Dictionary<string,
11                  (string FestNetz, string Mobilfunk, string Email)>
12              {
13                  {"Robert", ("089 12345678", "0171 12345678",
14                  "robert@email.de") },
15                  {"Katia", ("089 244345636", "0171 4534564",
16                  "katia@email.de") },
17                  {"Karl", ("089 2434545636", "0171 4294564", "karl@"
18                  "email.de") },
19                  {"Thea", ("089 238545736", "0171 246732589",
20                  "thea@email.de") }
21              };
22          }
23      }
24 }
```

## Musterlösung für Teilaufgabe 2

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TelefonVerzeichnis
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var telefonVerzeichnis = new Dictionary<string,
11                                         (string FestNetz, string Mobilfunk, string Email)>
12             {
13                 {"Robert", ("089 12345678","0171 12345678",
14                  "robert@email.de") },
15                 {"Katia", ("089 244345636","0171 4534564",
16                  "katia@email.de") },
17                 {"Karl", ("089 2434545636","0171 4294564", "karl@"
18                  "email.de") },
19                 {"Thea", ("089 238545736","0171 246732589",
20                  "thea@email.de") }
21             };
22
23             var name = "";
24
25             while(name != "Fertig!")
26             {
27                 Console.WriteLine("Geben Sie bitte einen Namen
28                           ein:");
29                 name = Console.ReadLine();
30             }
31         }
32     }
33 }
34 }
```

7



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the output of a C# program that reads names from the user until "Fertig!" is entered. The console window has a dark background with white text. The output is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie bitte einen Namen ein:
Robert
Geben Sie bitte einen Namen ein:
Katia
Geben Sie bitte einen Namen ein:
Thea
Geben Sie bitte einen Namen ein:
Frieda
Geben Sie bitte einen Namen ein:
Fertig!
C:\Program Files\dotnet\dotnet.exe (Prozess "10052") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 7.6.1 Ausgabe der Musterlösung für Teilaufgabe 2

**Musterlösung zu Teilaufgabe 3**

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TelefonVerzeichnis
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              var telefonVerzeichnis = new Dictionary<string,
11                  (string FestNetz, string Mobilfunk, string Email)>
12              {
13                  {"Robert", ("089 12345678", "0171 12345678",
14                      "robert@email.de") },
15                  {"Katia", ("089 244345636", "0171 4534564",
16                      "katia@email.de") },
17                  {"Karl", ("089 2434545636", "0171 4294564", "karl@"
18                      "email.de") },
19                  {"Thea", ("089 238545736", "0171 246732589",
20                      "thea@email.de") }
21              };
22
23              var name = "";
24
25              while(name != "Fertig!")
26              {
27                  Console.WriteLine("Geben Sie bitte einen Namen
28                      ein:");
29
30                  name = Console.ReadLine();
31
32                  if(name != "Fertig!" && telefonVerzeichnis.
33                      ContainsKey(name) == false)
34                  {
35                      Console.WriteLine($"Der Name {name} ist mir
36                          nicht bekannt.");
37                  }
38              }
39          }
40      }
41  }
```

## 7.6 Übungsaufgaben: Arbeiten mit Arrays, Listen, Dictionaries und Tuples



The screenshot shows a Microsoft Visual Studio Debugging Console window. The console output is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie bitte einen Namen ein:
Robert
Geben Sie bitte einen Namen ein:
Katia
Geben Sie bitte einen Namen ein:
Thea
Geben Sie bitte einen Namen ein:
Frieda
Der Name Frieda ist mir nicht bekannt.
Geben Sie bitte einen Namen ein:
Fertig!

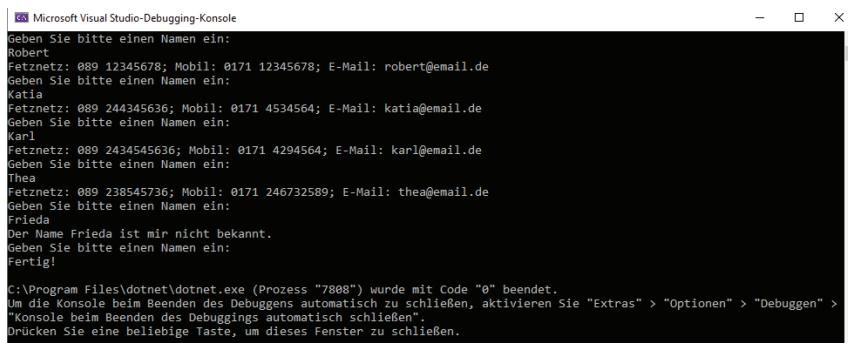
C:\Program Files\dotnet\dotnet.exe (Prozess "8492") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 7.6.2 Ausgabe der Musterlösung für Teilaufgabe 3

**Musterlösung zu Teilaufgabe 4**

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TelefonVerzeichnis
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var telefonVerzeichnis = new Dictionary<string,
11                                         string FestNetz, string Mobilfunk, string Email>
12             {
13                 {"Robert", ("089 12345678", "0171 12345678",
14                  "robert@email.de") },
15                 {"Katia", ("089 244345636", "0171 4534564",
16                  "katia@email.de") },
17                 {"Karl", ("089 2434545636", "0171 4294564", "karl@"
18                  "email.de") },
19                 {"Thea", ("089 238545736", "0171 246732589",
20                  "thea@email.de") }
21             };
22
23             var fertig = "Fertig!";
24             var name = "";
25
26             while(name != fertig)
27             {
28                 Console.WriteLine("Geben Sie bitte einen Namen"
29                               "ein:");
30
31                 name = Console.ReadLine();
32
33                 if(telefonVerzeichnis.ContainsKey(name))
34                 {
35                     var eintrag = telefonVerzeichnis[name];
36                     Console.WriteLine($"Festnetz: {eintrag."
37                                     "FestNetz}; Mobil: {eintrag.Mobilfunk};"
38                                     "E-Mail: {eintrag.Email}");
39                 }
40                 else
41                 {
42                     if(name != fertig)
43                     {
44                         Console.WriteLine($"Der Name {name} ist"
45                                       "mir nicht bekannt.");
46                     }
47                 }
48             }
49         }
50     }
51 }
```

## 7.6 Übungsaufgaben: Arbeiten mit Arrays, Listen, Dictionaries und Tuples



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays a series of user inputs and corresponding system responses. The user repeatedly enters names (Robert, Katia, Karl, Thea, Frieda) followed by their phone number, mobile number, and email address. The program then prints a message indicating that Frieda's name is unknown. Finally, it prompts the user to press any key to close the window.

```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie bitte einen Namen ein:
Robert
Fetzenetz: 089 12345678; Mobil: 0171 12345678; E-Mail: robert@email.de
Geben Sie bitte einen Namen ein:
Katia
Fetzenetz: 089 244345636; Mobil: 0171 4534564; E-Mail: katia@email.de
Geben Sie bitte einen Namen ein:
Karl
Fetzenetz: 089 2434545636; Mobil: 0171 4294564; E-Mail: karl@email.de
Geben Sie bitte einen Namen ein:
Thea
Fetzenetz: 089 238545736; Mobil: 0171 246732589; E-Mail: thea@email.de
Geben Sie bitte einen Namen ein:
Frieda
Der Name Frieda ist mir nicht bekannt.
Geben Sie bitte einen Namen ein:
Fertig!

C:\Program Files\dotnet\dotnet.exe (Prozess "7808") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 7.6.3 Ausgabe der Musterlösung für Teilaufgabe 4

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 8

# Methoden schaffen Ordnung

Bisher haben all unsere Programme nur eine einzige Methode: die Methode `Main`. Wir haben sie bisher immer nur „die Hauptmethode“ genannt. Aber warum Hauptmethode, warum nicht einfach nur „die Methode“ oder den Startpunkt des Programms? Und warum steht `static void` vor der Methode `Main`?

Ein Programm kann beliebig viele Methoden haben, aber nur eine Hauptmethode, die den Startpunkt des Programms darstellt. Wie man in einem Programm weitere Methoden erstellt, wie man sie verwendet und wozu sie gut sind, wird in den folgenden Kapiteln behandelt. Dort wird auch erklärt, was die Schlüsselwörter `static` und `void` vor der Methode `Main` bedeuten. Und am Ende der Kapitel über Methoden gibt es das versprochene Tic-Tac-Toe-Spiel als Übungsaufgabe.

8

### 8.1 Wozu benötigt man Methoden?

Methoden benötigt man, um seine Programme besser zu strukturieren. Stellen Sie sich vor, Sie würden ein größeres und komplexeres Programm schreiben, für das Sie etwa 10.000 Zeilen benötigen. Und Sie können mir glauben, in der professionellen Software-Entwicklung gehören Programme mit 10.000 Zeilen zu den kleineren Programmen. Wenn Sie für Ihr Programm mit 10.000 Zeilen nur das verwenden dürften, was ich Ihnen bisher in diesem Buch vorgestellt habe, dann hätten Sie eine Hauptmethode mit ca. 10.000 Zeilen. Das wäre unübersichtlich und Wartungsarbeiten an diesem Programm wären mühselig und zeitaufwendig. Mit Methoden können Sie sich Ihre eigenen kleinen Unterprogramme schreiben, die sie in Ihrem Programm mehrfach verwenden können.

### 8.2 Einfache Methoden

Weitere Methoden - neben der Hauptmethode - werden innerhalb der Klasse `Program`, aber außerhalb der Hauptmethode `Main` deklariert. Betrachten wir aber zunächst das „Hello Word!“-Programm mit einer eigenen Methode:

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
```

## 8 Methoden schaffen Ordnung

```
8     {
9         Hallo();
10    }
11
12    static void Hallo()
13    {
14        Console.WriteLine("Hello World!");
15    }
16}
17 }
```

Unsere erste Methode heißt `Hallo()`, sie steht außerhalb der Hauptmethode und - wie bei der Hauptmethode - stehen vor dem Methodennamen die Schlüsselwörter `static` und `void`. Das Schlüsselwort `static` bedeutet, dass es sich um eine sogenannte statische Methode handelt. Das ist nötig, damit wir sie direkt aus der Hauptmethode aufrufen können. Das Programmiermodell von C# sieht vor, dass ein C# Programm, nachdem es geladen wurde, die statische Methode `Program.Main()` aufruft. Warum ist das so? Ganz einfach: Die Entwickler des .NET-Frameworks bei Microsoft haben beschlossen, dass sie das nun mal so implementieren. Wenn statische Methoden andere Methoden aus derselben Klasse aufrufen wollen, müssen diese anderen Methoden ebenfalls statisch sein. Das heißt unsere Hauptmethode muss statisch sein, weil der .NET-Framework nun mal so gebaut ist. Daher muss unsere zweite Methode „Hallo“ auch statisch sein, weil wir sie sonst nicht aufrufen können. Da stellt sich sofort die Frage, welchen Sinn ergibt es dann, eine nicht-statische Methode zu deklarieren. Nach all dem, was wir bisher gelernt haben, ergibt es tatsächlich keinen Sinn. Aber wenn wir unser Wissen in einem späteren Kapitel um die Objektorientierte Programmierung erweitern, ergeben nicht-statische Methoden sogar sehr viel Sinn.

Das zweite Schlüsselwort, das vor der Methode `Hallo` steht, heißt `void`. Im Englischen heißt das „die Leere“ oder „die Leerstelle“. In C# bedeutet das einfach, dass unsere Methode keinen Wert zurückgibt. In der Hauptmethode rufen wir die Methode `Hallo` mit dem Namen der Methode, gefolgt von zwei runden Klammern, auf – natürlich mit einem Semikolon am Ende, was für C# „Anweisung beendet“ bedeutet.

Innerhalb der Methode steht das, was die Methode tun soll. In unserem Fall ist das einfach die Anweisung:

```
1 Console.WriteLine("Hello World!");
```

denn schließlich handelt es um unser „Hello World!“-Programm für Methoden.

### 8.3 Methoden mit Übergabeparametern

Die Hauptmethode hat einen Übergabeparameter, das string-Array `args`. Es enthält die Aufrufargumente des Programms. Bei unseren selbstgeschriebenen Methoden

können wir auch Übergabeparameter verwenden. Dazu betrachten wir das folgende Beispiel:

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Hallo("Hello World!");
10             Hallo("Hallo Welt!");
11             Hallo("Hola Mundo!");
12         }
13
14         static void Hallo(string ausgabe)
15         {
16             Console.WriteLine(ausgabe);
17         }
18     }
19 }
```

8

Wie bei der Hauptmethode steht der Übergabeparameter in den Klammern nach dem Methodennamen. Im obigen Beispiel ist das der Übergabeparameter `ausgabe`. Er hat den Typ `string`. Die Methode `Hallo` gibt den Wert des Übergabeparameters `ausgabe` am Bildschirm aus. In der Hauptmethode rufen wir `Hallo` dreimal auf und übergeben ihr den Text „Hello World!“ jeweils in einer anderen Sprache. Angenommen, die Methode `Hallo` würde nicht nur eine simple Bildschirmausgabe machen, sondern wäre komplizierter und hätte beispielsweise 20 Zeilen Code, dann wäre die Verwendung einer eigenen Methode für diese 20 Zeilen ein echter Gewinn für unser Programm. Eine Methode kann auch mehrere Übergabeparameter verarbeiten:

```
1  static void Addiere(int a, int b)
2  {
3      Console.WriteLine($"{a} + {b} = {a + b}");
4 }
```

Die Methode „Addiere“ bekommt zwei Übergabeparameter vom Typ `int` und gibt die Summe dieser beiden Zahlen am Bildschirm aus.

Methoden können auch optionale Parameter haben. Optionale Parameter müssen am Ende der Parameter-Liste stehen und benötigen einen Standardwert, der verwendet wird, falls der Parameter nicht übergeben wird.

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
```

## 8 Methoden schaffen Ordnung

```
6      {
7          static void Main(string[] args)
8          {
9              Hallo("Robert");
10             Hallo("Katia", "Spanisch");
11         }
12
13         static void Hallo(string vorname, string
14             sprache="Deutsch")
15         {
16             switch(sprache)
17             {
18                 case "Deutsch":
19                     Console.WriteLine($"Hallo {vorname}!");
20                     break;
21                 case "Englisch":
22                     Console.WriteLine($"Hello {vorname}!");
23                     break;
24                 case "Spanisch":
25                     Console.WriteLine($"Hola {vorname}!");
26                     break;
27                 default:
28                     Console.WriteLine($"Die Sprache {sprache}
29                         spreche ich nicht!");
30                     break;
31             }
32         }
33     }
32 }
```

Die Methode „Hallo“ hat zwei Übergabeparameter, vom Typ `string`, wobei dem letzten der beiden Parameter der Wert „Deutsch“ zugewiesen wird. Damit können wir die Methode `Hallo` aufrufen und ihr nur einen Wert für den Parameter `vorname` geben. Die erste Anweisung der Hauptmethode macht genau das. Die Methode `Hallo` verhält sich nun genauso, als hätte man für den Parameter `sprache` den Wert „Deutsch“ übergeben und zeigt eine Begrüßung auf Deutsch an. Die zweite Anweisung der Hauptmethode ruft die Methode „Hallo“ mit einem Wert für den Parameter `vorname` auf und setzt auch den Parameter `sprache` auf den Wert „Spanisch“. Die Methode `Hallo` gibt dann eine Begrüßung auf Spanisch aus. Die Bildschirmausgabe des Beispielprogramms sieht dann so aus:

Hallo Robert!

Hola Katia!

Eine weitere Möglichkeit der Parameterübergabe ist das „params“-Array. Hierbei handelt es sich wieder mal um „syntactic sugar“. Intern ist das „params“-Array, aber ein ganz normales Array. Wenn wir einer Methode eine pro Aufruf unterschiedliche Anzahl an Parametern des gleichen Typs übergeben wollen, dann denken wir natürlich

zuerst an ein Array, aber mit einem „params“-Array geht das etwas bequemer und eleganter.

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine(Summe(1, 2, 3, 4, 5));
10             Console.WriteLine(Summe(1, 2, 3));
11         }
12
13         static int Summe(params int[] summanden)
14         {
15             var summe = 0;
16             foreach(var summand in summanden)
17             {
18                 summe += summand;
19             }
20             return summe;
21         }
22
23     }
24 }
```

8

Der Übergabeparameter `summanden` ist ein ganz normales `int`-Array und wird innerhalb der Methode `Summe` auch so verwendet. Wenn wir allerdings vor den Parameter das Schlüsselwort `params` schreiben, benötigen wir kein Array mehr, um die Methode `Summe` aufzurufen. Es genügt, die Werte des Arrays als normale Parameter an die Methode zu übergeben.

## 8.4 Methoden mit Rückgabewerten

Methoden können nicht nur Werte beim Aufruf entgegennehmen, sie können auch Werte zurückgeben. Damit eine Methode einen Wert zurückgeben kann, schreiben wir - anstelle des Schlüsselworts `void` - den Typ des Werts, den die Methode zurückgeben soll. Als letzte Anweisung innerhalb der Methode schreiben wir das Schlüsselwort `return`, gefolgt von einem Ausdruck, der den Wert bestimmt, den die Methode zurückgeben soll.

```
1  static int addiere(int a, int b)
2  {
3      var c = a + b;
4      return c;
5 }
```

## 8 Methoden schaffen Ordnung

Den Rückgabewert einer Methode erhält man über eine Zuweisung des Methodenaufrufs an eine Variable.

```
1 var ergebnis = addiere(2, 3);
```

Damit erhält die Variable `ergebnis` den Wert 5. Sie können eine Methode, die einen Wert zurückgibt, auch als Teil eines Ausdrucks verwenden.

```
1 var ergebnis = 5 + addiere(2, 3);
```

Jetzt bekommt die Variable `ergebnis` den Wert 10.

Stellen wir uns nun vor, wir würden gerne eine Methode schreiben, die nicht nur einen Wert, sondern zwei oder mehr Werte zurückgibt. Dafür definieren wir einen so genannten out-Parameter als Übergabeparameter.

```
1 using System;
2
3 namespace Methoden
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var summe = SummeUndDifferenz(5, 2, out int differenz);
10            Console.WriteLine($"Die Summe von 5 und 2 =
11            {summe}");
12            Console.WriteLine($"Die Differenz zwischen 5 und 2 =
13            {differenz}");
14        }
15
16        static int SummeUndDifferenz(int a, int b, out int
17        differenz)
18        {
19            differenz = a - b;
20            return a + b;
21        }
22
23    }
24 }
```

Wir benötigen die Summe und die Differenz von zwei Zahlen. Dazu lassen wir bei den Berechnungen von der Methode `SummeUndDifferenz()` erledigen. Die Summe geben wir mit dem normalen Rückgabewert der Methode zurück. Für die Differenz definieren wir den out-Parameter `differenz` als Übergabeparameter. Dazu schreiben wir das Schlüsselwort `out` vor den Übergabeparameter. In der Methode weisen wir dem Parameter `differenz` einfach den gewünschten Wert zu. Beim Aufruf der Methode `SummeUndDifferenz()` übergeben wir für den Parameter `differenz` eine Variablendeclaration, vor die wir das Schlüsselwort `out` schreiben. Auf die Variable `differenz` kann in der Hauptmethode erst nach dem Aufruf der Methode

SummeUndDifferenz() zugegriffen werden. Ein out-Parameter kann beim Aufruf und in seiner Methodendeklaration den gleichen Namen haben. Das muss aber nicht sein.

Erinnern Sie sich noch an die Methode „int.Parse“, mit der ein String in einen Integer-Wert konvertiert werden kann?

```
1 var zahl = int.Parse(eingabe);
```

Die Variable `eingabe` ist ein String, der vom Benutzer eingegeben wurde, und er wird mit Hilfe der Methode `int.Parse()` in einen Integer-Wert konvertiert und der Integer-Variablen `zahl` zugewiesen. Allerdings sollte jeder Programmierer damit rechnen, dass ein Benutzer nicht unbedingt das tut, was das Programm von ihm erwartet. Wenn `eingabe` ein vom Benutzer eingegebener String ist, kann da alles Mögliche drinstehen. Aber wenn der Benutzer keine Zahl eingibt und wir versuchen, den String `eingabe` mit der Methode `int.Parse()` zu konvertieren, dann erhalten wir einen Fehler. Die Klasse `int` stellt für diese Fälle die Methode `TryParse()` bereit, die mit Hilfe eines `out`-Parameters das Problem elegant umgeht.

8

```
1 using System;
2
3 namespace Methoden
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Geben Sie eine ganze Zahl:");
10
11            var eingabe = Console.ReadLine();
12
13            if (int.TryParse(eingabe, out int zahl))
14            {
15                Console.WriteLine($"Das doppelte von {zahl} ist
16                {zahl * 2}.");
17            }
18            else
19            {
20                Console.WriteLine($"{eingabe} ist keine ganze
21                Zahl.");
22            }
23        }
24    }
25}
26}
```

An Stelle von `int.Parse()` verwenden wir `int.TryParse()` zur Konvertierung. `int.TryParse()` versucht, wie der Name schon sagt, eine Konvertierung und gibt einen Boole'schen Wert zurück, der `true` ist, wenn die Konvertierung funktioniert und `false`, wenn die Konvertierung fehlschlägt. Damit wir aber auch auf das Ergebnis

## 8 Methoden schaffen Ordnung

nis der Konvertierung zugreifen können, müssen wir der Methode `int.TryParse()` einen `out`-Parameter für das Resultat der Konvertierung übergeben.

Die letzte Möglichkeit, um für Methoden einen Wert zurückzugeben, die ich vorstellen möchte, ist der Einsatz von „ref-Parametern“. Dazu schauen wir uns zunächst das folgende kleine experimentelle Programm an:

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var a = 2;
10             var b = 3;
11             var c = VerdoppleUndAddiere(a, b);
12             Console.WriteLine($"a = {a}");
13             Console.WriteLine($"b = {b}");
14             Console.WriteLine($"c = {c}");
15         }
16
17         static int VerdoppleUndAddiere(int a, int b)
18         {
19             a *= 2;
20             b *= 2;
21             return a + b;
22         }
23
24     }
25 }
```

Eigentlich ist die Methode `VerdoppleUndAddiere()` ganz simpel. Wir übergeben zwei Zahlen, verdoppeln beide Zahlen und zählen danach die verdoppelten Zahlen zusammen und geben das Ergebnis zurück. Die Bildschirmausgabe des Programms sieht so aus:

a = 2

b = 3

c = 10

Den Wert 10 für die Variable `c` hätte wohl jeder erwartet. Aber warum haben die Variablen `a` und `b` die Werte 2 und 3? Wir haben Sie doch in unserer Methode verdoppelt: Sollten sie nicht die Werte 4 und 6 haben?

Die Erklärung ist relativ einfach: C# über gibt primitive Datentypen standardmäßig als Kopie an Methoden. Das heißt, innerhalb der Methode arbeiten wir mit einer Kopie der Variablen `a` und `b`. Das erklärt, warum die Variablen `a` und `b` der Hauptmethode sich nach dem Aufruf der Methode `VerdoppleUndAddiere()` nicht verändert haben.

Variablen als Kopie zu übergeben, kann man machen, muss man aber nicht. Wir verändern das Experimentierprogramm etwas und sehen, was passiert.

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var a = 2;
10             var b = 3;
11             var c = VerdoppleUndAddiere(ref a, ref b);
12             Console.WriteLine($"a = {a}");
13             Console.WriteLine($"b = {b}");
14             Console.WriteLine($"c = {c}");
15         }
16
17         static int VerdoppleUndAddiere(ref int a, ref int b)
18         {
19             a *= 2;
20             b *= 2;
21             return a + b;
22         }
23     }
24 }
25 }
```

8

Wir haben in der Deklaration der Methode vor die beiden Variablen `a` und `b` das Schlüsselwort `ref` geschrieben und auch beim Aufruf der Methode steht `ref` vor den übergebenen Variablen.

Die Bildschirmausgabe des Programms ist:

`a = 4;`

`b = 6;`

`c = 10;`

Jetzt werden durch die Methode `VerdoppleUndAddiere` auch die Übergabeparameter `a` und `b` verändert. Das Schlüsselwort `ref` weist den Compiler an, die Parame-

## 8 Methoden schaffen Ordnung

ter als Referenz auf die übergebene Variable zu übertragen. Das heißt, in der Methode wird jetzt auf den Originalvariablen gearbeitet.

### 8.5 Methoden überladen

Um das Überladen von Methoden zu verstehen, betrachten wir folgende Anforderung: Wir benötigen eine Methode mit dem String-Übergabeparameter `vorname`, die für den übergebenen Vornamen eine Begrüßung ausgibt. Des Weiteren benötigen wir eine zweite Methode, die das String-Array `vornamen` entgegennimmt und für jeden String in diesem Array eine Begrüßung am Bildschirm ausgibt. Eine mögliche Lösung für diese Anforderung könnte so aussehen:

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              HalloEinzelperson("Robert");
10             string[] vornamen = { "Katia", "Karl", "Thea" };
11             HalloPersonenGruppe(vornamen);
12         }
13
14         static void HalloEinzelperson(string vorname)
15         {
16             Console.WriteLine($"Hallo {vorname}");
17         }
18
19         static void HalloPersonenGruppe(string[] vornamen)
20         {
21             foreach(var vorname in vornamen)
22             {
23                 HalloEinzelperson(vorname);
24             }
25         }
26     }
27 }
28 }
```

Die Methodennamen `HalloEinzelperson()` und `HalloPersonenGruppe()` sind etwas sperrig. Wäre es da nicht viel eleganter, wenn beide Methoden einfach `Hallo` heißen würden. Beim Aufruf der Methoden würde es so aussehen, als ob es nur eine Methode `Hallo` gibt, der man wahlweise einen einzelnen String oder ein String-Array übergeben könnte und die Methode wüsste, was sie jeweils mit den unterschiedlichen Parametertypen anfangen soll. Und genau diese Technik nennt man Methodenüberladung. Wenn wir jetzt unsere beiden Methoden `HalloEinzelperson()` und `HalloPersonenGruppe()` einfach nur `Hallo` nennen, dann funktioniert unser Programm genauso wie vorher.

```
1  using System;
2
3  namespace Methoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Hallo("Robert");
10             string[] vornamen = { "Katia", "Karl", "Thea" };
11             Hallo(vornamen);
12         }
13
14         static void Hallo( string vorname)
15         {
16             Console.WriteLine($"Hallo {vorname}");
17         }
18
19         static void Hallo(string[] vornamen)
20         {
21             foreach(var vorname in vornamen)
22             {
23                 Hallo(vorname);
24             }
25         }
26     }
27 }
28 }
```

8

## 8.6 Übungsaufgaben: Programmieren mit Methoden

Wie versprochen, programmieren wir jetzt schrittweise ein Tic-Tac-Toe-Spiel. Die Aufgabe ist aufgeteilt in sieben Teilaufgaben. Anschließend an die Teilaufgaben finden Sie Musterlösungen für jede Teilaufgabe, die das vollständige Programm bis einschließlich der Lösung der jeweiligen Teilaufgabe zeigt.

### Teilaufgabe 1

Deklarieren Sie in der Hauptmethode geeignete Variablen für Zeichen, die jeweils ein leeres Feld, ein Feld mit einem Kreuz und ein Feld mit einem Kreis symbolisieren. Deklarieren Sie eine Struktur, die den Zustand eines Tic-Tac-Toe-Spielfelds speichern kann. Die Struktur soll auch ein Koordinatensystem enthalten, das in horizontaler Richtung von 1 bis 3 läuft und in vertikaler Richtung von A bis C. Belegen Sie alle Variablen mit Startwerten.

### Teilaufgabe 2:

Legen Sie in der Hauptmethode zwei geeignete Variablen an, um die Namen von zwei Spielern zu speichern. Schreiben Sie eine Methode, die den Namen für einen Spieler einliest und rufen Sie diese Methode zweimal in der Hauptmethode auf, um die Namen von zwei Spielern abzufragen.

## 8 Methoden schaffen Ordnung

### Teilaufgabe 3:

Schreiben Sie die Methode Spielzug() mit dem Rückgabetyp void und folgenden Übergabeparametern:

```
1 string name, string spielstein, string[,] spielfeld, string  
2 leeresFeld
```

Der Übergabeparameter name erwartet den Namen des Spielers, der Parameter spielstein soll das Zeichen für den Spielstein (Kreuz oder Kreis) des Spielers enthalten. Das Array spielfeld soll die in der Hauptmethode definierte Struktur erhalten, die den Zustand des Spielfelds speichert. Der Parameter leeresFeld soll das Zeichen enthalten, das ein leeres Feld symbolisiert. Die Methode Spielzug soll einen Spielzug machen. Dazu soll sie den im Parameter name übergebenen Benutzer nach den Koordinaten seines Zuges fragen. Die Koordinaten sollen in der Form Buchstabe,Zahl eingegeben werden: zum Beispiel A,1 oder C,3. Bei ungültigen Eingaben oder wenn das Feld an den vom Spieler eingegebenen Koordinaten schon besetzt ist, soll der Spieler auf seinen Fehler aufmerksam gemacht werden und das Programm soll nochmal die Koordinaten für den Zug des Spielers abfragen. Bei gültigen Koordinaten soll die Methode an die eingegebenen Koordinaten den Spielstein (Kreuz oder Kreis) setzen und die Methode endet.

**ACHTUNG:** Beachten Sie, dass die Variablen name, spielstein und leeresFeld einen primitiven Datentyp haben. Sie werden als Kopie an die Methode übergeben. Die Variable spielfeld dagegen ist ein Array. Arrays werden standardmäßig nicht als Kopie, sondern als ref-Parameter übergeben, auch wenn das Schlüsselwort ref nicht vor dem Parameter steht. Das heißt, wenn die Methode Spielzug einen Wert in den Array-Parameter spielfeld schreibt, dann ist dieser neue Wert auch außerhalb der Methode Spielzug verfügbar.

### Teilaufgabe 4:

Schreiben Sie die Methode ZeigeSpielfeld(). Die Methode ZeigeSpielfeld() hat den Rückgabetyp void und erhält als Übergabeparameter das zweidimensionale Array spielfeld, das in der Hauptmethode deklariert wurde. Die Methode ZeigeSpielfeld() soll das Tic-Tac-Toe-Spielfeld als Quadrat von 4 mal 4 Zeichen am Bildschirm ausgeben.

### Teilaufgabe 5:

Schreiben Sie die Methode HatGewonnen(), die einen Boole'schen Wert zurückgibt und die Übergabeparameter spielstein vom Typ string und das zweidimensionale Array spielfeld erhält. Der Parameter spielstein erwartet das Zeichen, das den Spielstein des jeweiligen Spielers symbolisiert, für den die Methode überprüfen

soll, ob er mit seinem Zug gewonnen hat oder nicht. Die Methode soll `true` zurückgeben, wenn der Spieler drei Steine in einer Reihe hat, sonst `false`. Drei Steine in einer diagonalen Reihe gelten auch als Sieg.

**Teilaufgabe 6:**

Schreiben Sie die Methode `SpieldFeldVoll()`, die einen Boole'schen Wert zurückgibt und als Übergabeparameter das zweidimensionale Array `spielfeld` und den String-Parameter `leeresFeld` erhält. Der Parameter `leeresFeld` erwartet das Zeichen, das ein leeres Feld auf dem Spielfeld symbolisiert. Wenn das Spielfeld voll ist und somit kein weiterer Stein gesetzt werden kann, gibt die Methode `true` aus, sonst `false`.

**Teilaufgabe 7:**

Schreiben Sie das Spiel fertig. Implementieren Sie in der Hauptmethode eine Schleife, die abwechselnd einen Zug von Spieler 1 und Spieler 2 abfragt und dann das Spielfeld anzeigt. Wenn ein Spieler gewonnen hat oder wenn kein weiterer Zug mehr möglich ist, brechen Sie die Schleife mit einer entsprechenden Meldung ab.

## 8 Methoden schaffen Ordnung

### Musterlösung für Teilaufgabe 1

```
1  using System;
2
3  namespace TicTacToe
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var leeresFeld = "#";
10             var kreuz = "X";
11             var kreis = "O";
12
13             var spielFeld = new string[4, 4]
14             {
15                 {" ", "1", "2", "3"},  

16                 {"A",leeresFeld, leeresFeld, leeresFeld},  

17                 {"B",leeresFeld, leeresFeld, leeresFeld},  

18                 {"C",leeresFeld, leeresFeld, leeresFeld}
19             };
20
21         }
22
23     }
24 }
```

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2
3  namespace TicTacToe
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var leeresFeld = "#";
10             var kreuz = "X";
11             var kreis = "O";
12
13             string spieler1, spieler2;
14
15             var spielFeld = new string[4, 4]
16             {
17                 {" ", "1", "2", "3"},
18                 {"A", leeresFeld, leeresFeld, leeresFeld},
19                 {"B", leeresFeld, leeresFeld, leeresFeld},
20                 {"C", leeresFeld, leeresFeld, leeresFeld}
21             };
22
23             spieler1 = FrageSpielerNameAb(1);
24             spieler2 = FrageSpielerNameAb(2);
25         }
26
27         static string FrageSpielerNameAb(int spielerNummer)
28         {
29             Console.WriteLine($"Geben Sie einen Namen für Spieler
30             {spielerNummer} ein:");
31             return Console.ReadLine();
32         }
33
34     }
35 }
```

## 8 Methoden schaffen Ordnung

### Musterlösung für Teilaufgabe 3:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TicTacToe
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var leeresFeld = "#";
11             var kreuz = "X";
12             var kreis = "O";
13
14             string spieler1, spieler2;
15
16             var spielFeld = new string[4, 4]
17             {
18                 {" ", "1", "2", "3"},
19                 {"A", leeresFeld, leeresFeld, leeresFeld},
20                 {"B", leeresFeld, leeresFeld, leeresFeld},
21                 {"C", leeresFeld, leeresFeld, leeresFeld}
22             };
23
24             spieler1 = FrageSpielerNameAb(1);
25             spieler2 = FrageSpielerNameAb(2);
26
27             Spielzug(spieler1, kreuz, spielFeld, leeresFeld);
28         }
29
30         static string FrageSpielerNameAb(int spielerNummer)
31         {
32             Console.WriteLine($"Geben Sie einen Namen für Spieler
33             {spielerNummer} ein:");
34             return Console.ReadLine();
35         }
36
37         static void Spielzug(string name, string spielstein,
38         string[,] spielFeld, string leeresFeld)
39         {
40
41             var eingabeOk = false;
42             List<string> buchstaben = new List<string> { "A",
43                 "B", "C" };
44             List<string> zahlen = new List<string> { "1", "2",
45                 "3" };
46             string fehler = "Ungültiger Zug!";
47             string feldBesetzt = "Dieses Feld ist schon
48             besetzt!";
49
50             while(!eingabeOk)
51             {
52                 Console.WriteLine($"{name}, geben Sie die
53                 Koordinaten für Ihren Zug ein (Buchstabe,Zahl");
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

```
55     var eingabe = Console.ReadLine();  
56  
57     if (eingabe.Length != 3)  
58     {  
59         Console.WriteLine(fehler);  
60         continue;  
61     }  
62  
63     var koordinaten = eingabe.Split(",");  
64  
65     if (koordinaten.Length != 2)  
66     {  
67         Console.WriteLine(fehler);  
68     }  
69     if (!buchstaben.Contains(koordinaten[0]))  
70     {  
71         Console.WriteLine(fehler);  
72         continue;  
73     }  
74     if (!zahlen.Contains(koordinaten[1]))  
75     {  
76         Console.WriteLine(fehler);  
77         continue;  
78     }  
79  
80     var i = buchstaben.IndexOf(koordinaten[0]) + 1;  
81     var j = int.Parse(koordinaten[1]);  
82  
83     if (spielFeld[i, j] != leeresFeld)  
84     {  
85         Console.WriteLine(feldBesetzt);  
86         continue;  
87     }  
88  
89     spielFeld[i, j] = spielstein;  
90  
91     eingabeOk = true;  
92 }  
93 }  
94 }  
95 }
```

8

## 8 Methoden schaffen Ordnung

### Musterlösung für Teilaufgabe 4:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TicTacToe
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var leeresFeld = "#";
11             var kreuz = "X";
12             var kreis = "O";
13
14             string spieler1, spieler2;
15
16             var spielFeld = new string[4, 4]
17             {
18                 {" ", "1", "2", "3"},
19                 {"A", leeresFeld, leeresFeld, leeresFeld},
20                 {"B", leeresFeld, leeresFeld, leeresFeld},
21                 {"C", leeresFeld, leeresFeld, leeresFeld}
22             };
23
24             spieler1 = FrageSpielerNameAb(1);
25             spieler2 = FrageSpielerNameAb(2);
26
27         }
28
29         static string FrageSpielerNameAb(int spielerNummer)
30         {
31             Console.WriteLine($"Geben Sie einen Namen für Spieler
32             {spielerNummer} ein:");
33             return Console.ReadLine();
34         }
35
36         static void Spielzug(string name, string spielstein,
37             string[,] spielFeld, string leeresFeld)
38         {
39
40             var eingabeOk = false;
41             List<string> buchstaben = new List<string> { "A",
42                 "B", "C" };
43             List<string> zahlen = new List<string> { "1", "2",
44                 "3" };
45             string fehler = "Ungültiger Zug!";
46             string feldBesetzt = "Dieses Feld ist schon
47             besetzt!";
48
49             while (!eingabeOk)
50             {
51                 Console.WriteLine($"{name}, geben Sie die
52                     Koordinaten für Ihren Zug ein (Buchstabe,Zahl");
53
54                 var eingabe = Console.ReadLine();
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

```
55         if (eingabe.Length != 3)
56     {
57         Console.WriteLine(fehler);
58         continue;
59     }
60
61     var koordinaten = eingabe.Split(",");
62
63     if(koordinaten.Length !=2)
64     {
65         Console.WriteLine(fehler);
66     }
67
68     if(!buchstaben.Contains(koordinaten[0]))
69     {
70         Console.WriteLine(fehler);
71         continue;
72     }
73
74     if(!zahlen.Contains(koordinaten[1]))
75     {
76         Console.WriteLine(fehler);
77         continue;
78     }
79
80     var i = buchstaben.IndexOf(koordinaten[0]) + 1;
81     var j = int.Parse(koordinaten[1]);
82
83     if(spielFeld[i,j] != leeresFeld)
84     {
85         Console.WriteLine(feldBesetzt);
86         continue;
87     }
88
89     spielFeld[i, j] = spielstein;
90
91     eingabeOk = true;
92 }
93
94
95     static void ZeigeSpielFeld(string[,] spielFeld)
96     {
97         for(var i = 0; i < 4; i++)
98     {
99         for(var j = 0; j < 4; j++)
100         {
101             Console.Write(spielFeld[i, j]);
102         }
103         Console.WriteLine();
104     }
105 }
106 }
107 }
108 }
```

## 8 Methoden schaffen Ordnung

### Musterlösung für Teilaufgabe 5:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TicTacToe
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var leeresFeld = "#";
11             var kreuz = "X";
12             var kreis = "O";
13
14             string spieler1, spieler2;
15
16             var spielFeld = new string[4, 4]
17             {
18                 {" ", "1", "2", "3"},
19                 {"A", leeresFeld, leeresFeld, leeresFeld},
20                 {"B", leeresFeld, leeresFeld, leeresFeld},
21                 {"C", leeresFeld, leeresFeld, leeresFeld}
22             };
23
24             spieler1 = FrageSpielerNameAb(1);
25             spieler2 = FrageSpielerNameAb(2);
26         }
27
28         static string FrageSpielerNameAb(int spielerNummer)
29         {
30             Console.WriteLine($"Geben Sie einen Namen für Spieler
{spielerNummer} ein:");
31             return Console.ReadLine();
32         }
33
34
35         static void Spielzug(string name, string spielstein,
36         string[,] spielFeld, string leeresFeld)
37         {
38
39             var eingabeOk = false;
40             List<string> buchstaben = new List<string> { "A",
41                 "B", "C" };
42             List<string> zahlen = new List<string> { "1", "2",
43                 "3" };
44             string fehler = "Ungültiger Zug!";
45             string feldBesetzt = "Dieses Feld ist schon
besetzt!";
46
47             while(!eingabeOk)
48             {
49                 Console.WriteLine($"{name}, geben Sie die
50                 Koordinaten für Ihren Zug ein (Buchstabe,Zahl");
51
52                 var eingabe = Console.ReadLine();
53             }
54         }
55     }
56 }
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

```
55         if (eingabe.Length != 3)
56         {
57             Console.WriteLine(fehler);
58             continue;
59         }
60
61         var koordinaten = eingabe.Split(",");
62
63         if(koordinaten.Length !=2)
64         {
65             Console.WriteLine(fehler);
66         }
67
68         if(!buchstaben.Contains(koordinaten[0]))
69         {
70             Console.WriteLine(fehler);
71             continue;
72         }
73
74         if(!zahlen.Contains(koordinaten[1]))
75         {
76             Console.WriteLine(fehler);
77             continue;
78         }
79
80         var i = buchstaben.IndexOf(koordinaten[0]) + 1;
81         var j = int.Parse(koordinaten[1]);
82
83         if(spielFeld[i,j] != leeresFeld)
84         {
85             Console.WriteLine(feldBesetzt);
86             continue;
87         }
88
89         spielFeld[i, j] = spielstein;
90
91         eingabeOk = true;
92     }
93 }
94
95 static void ZeigeSpielFeld(string[,] spielFeld)
96 {
97     for(var i = 0; i < 4; i++)
98     {
99         for(var j = 0; j < 4; j++)
100         {
101             Console.Write(spielFeld[i, j]);
102         }
103         Console.WriteLine();
104     }
105 }
106
107 static bool HatGewonnen(string spielStein, string[,]
108 spielFeld)
109 {
110     // 3 Steine in der 1. Reihe?
```

## 8 Methoden schaffen Ordnung

```
111     if (spielFeld[1, 1] == spielStein &&
112         spielFeld[1, 2] == spielStein &&
113         spielFeld[1, 3] == spielStein)
114     return true;
115
116 // 3 Steine in der 2. Reihe?
117 if (spielFeld[2, 1] == spielStein &&
118     spielFeld[2, 2] == spielStein &&
119     spielFeld[2, 3] == spielStein)
120 return true;
121
122 // 3 Steine in der 3. Reihe?
123 if (spielFeld[3, 1] == spielStein &&
124     spielFeld[3, 2] == spielStein &&
125     spielFeld[3, 3] == spielStein)
126 return true;
127
128 // 3 Steine in der 1. Spalte
129 if (spielFeld[1, 1] == spielStein &&
130     spielFeld[2, 1] == spielStein &&
131     spielFeld[3, 1] == spielStein)
132 return true;
133
134 // 3 Steine in der 2. Spalte
135 if (spielFeld[1, 2] == spielStein &&
136     spielFeld[2, 2] == spielStein &&
137     spielFeld[3, 2] == spielStein)
138 return true;
139
140 // 3 Steine in der 3. Spalte
141 if (spielFeld[1, 3] == spielStein &&
142     spielFeld[2, 3] == spielStein &&
143     spielFeld[3, 3] == spielStein)
144 return true;
145
146 // 3 Steine diagonal von links oben
147 // nach rechts unten
148 if (spielFeld[1, 1] == spielStein &&
149     spielFeld[2, 2] == spielStein &&
150     spielFeld[3, 3] == spielStein)
151 return true;
152
153 // 3 Steine diagonal von rechts oben
154 // nach links unten
155 if (spielFeld[1, 3] == spielStein &&
156     spielFeld[2, 2] == spielStein &&
157     spielFeld[3, 1] == spielStein)
158 return true;
159
160 return false;
161 }
162 }
163 }
```

## Musterlösung für Teilaufgabe 6

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TicTacToe
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var leeresFeld = "#";
11             var kreuz = "X";
12             var kreis = "O";
13
14             string spieler1, spieler2;
15
16             var spielFeld = new string[4, 4]
17             {
18                 {" ", "1", "2", "3"}, 
19                 {"A", leeresFeld, leeresFeld, leeresFeld}, 
20                 {"B", leeresFeld, leeresFeld, leeresFeld}, 
21                 {"C", leeresFeld, leeresFeld, leeresFeld}
22             };
23
24             spieler1 = FrageSpielerNameAb(1);
25             spieler2 = FrageSpielerNameAb(2);
26         }
27
28         static string FrageSpielerNameAb(int spielerNummer)
29         {
30             Console.WriteLine($"Geben Sie einen Namen für Spieler
{spielerNummer} ein:");
31             return Console.ReadLine();
32         }
33
34
35         static void Spielzug(string name, string spielstein,
36         string[,] spielFeld, string leeresFeld)
37         {
38
39             var eingabeOk = false;
40             List<string> buchstaben = new List<string> { "A",
41                 "B", "C" };
42             List<string> zahlen = new List<string> { "1", "2",
43                 "3" };
44             string fehler = "Ungültiger Zug!";
45             string feldBesetzt = "Dieses Feld ist schon
besetzt!";
46
47             while(!eingabeOk)
48             {
49                 Console.WriteLine($"{name}, geben Sie die
Koordinaten für Ihren Zug ein (Buchstabe,Zahl)");
50
51                 var eingabe = Console.ReadLine();
52
53             }
54 }
```

## 8 Methoden schaffen Ordnung

```
55             if (eingabe.Length != 3)
56             {
57                 Console.WriteLine(fehler);
58                 continue;
59             }
60
61             var koordinaten = eingabe.Split(",");
62
63             if(koordinaten.Length !=2)
64             {
65                 Console.WriteLine(fehler);
66             }
67
68             if(!buchstaben.Contains(koordinaten[0]))
69             {
70                 Console.WriteLine(fehler);
71                 continue;
72             }
73
74             if(!zahlen.Contains(koordinaten[1]))
75             {
76                 Console.WriteLine(fehler);
77                 continue;
78             }
79
80             var i = buchstaben.IndexOf(koordinaten[0]) + 1;
81             var j = int.Parse(koordinaten[1]);
82
83             if(spielFeld[i,j] != leeresFeld)
84             {
85                 Console.WriteLine(feldBesetzt);
86                 continue;
87             }
88
89             spielFeld[i, j] = spielstein;
90
91             eingabeOk = true;
92         }
93     }
94
95     static void ZeigeSpielFeld(string[,] spielFeld)
96     {
97         for(var i = 0; i < 4; i++)
98         {
99             for(var j = 0; j < 4; j++)
100             {
101                 Console.Write(spielFeld[i, j]);
102             }
103             Console.WriteLine();
104         }
105     }
106
107     static bool HatGewonnen(string spielStein, string[,]
108     spielFeld)
109     {
110         // 3 Steine in der 1. Reihe?
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

```
111     if (spielFeld[1, 1] == spielStein &&
112         spielFeld[1, 2] == spielStein &&
113         spielFeld[1, 3] == spielStein)
114     return true;
115
116 // 3 Steine in der 2. Reihe?
117 if (spielFeld[2, 1] == spielStein &&
118     spielFeld[2, 2] == spielStein &&
119     spielFeld[2, 3] == spielStein)
120 return true;
121
122 // 3 Steine in der 3. Reihe?
123 if (spielFeld[3, 1] == spielStein &&
124     spielFeld[3, 2] == spielStein &&
125     spielFeld[3, 3] == spielStein)
126 return true;
127
128 // 3 Steine in der 1. Spalte
129 if (spielFeld[1, 1] == spielStein &&
130     spielFeld[2, 1] == spielStein &&
131     spielFeld[3, 1] == spielStein)
132 return true;
133
134 // 3 Steine in der 2. Spalte
135 if (spielFeld[1, 2] == spielStein &&
136     spielFeld[2, 2] == spielStein &&
137     spielFeld[3, 2] == spielStein)
138 return true;
139
140 // 3 Steine in der 3. Spalte
141 if (spielFeld[1, 3] == spielStein &&
142     spielFeld[2, 3] == spielStein &&
143     spielFeld[3, 3] == spielStein)
144 return true;
145
146 // 3 Steine diagonal von links oben
147 // nach rechts unten
148 if (spielFeld[1, 1] == spielStein &&
149     spielFeld[2, 2] == spielStein &&
150     spielFeld[3, 3] == spielStein)
151 return true;
152
153 // 3 Steine diagonal von rechts oben
154 // nach links unten
155 if (spielFeld[1, 3] == spielStein &&
156     spielFeld[2, 2] == spielStein &&
157     spielFeld[3, 1] == spielStein)
158 return true;
159
160 return false;
161 }
162
163 static bool SpielFeldVoll(string[,] spielFeld, string
164 leeresFeld)
165 {
166     var spielFeldVoll = true;
```

## 8 Methoden schaffen Ordnung

```
167         for(var i = 1; i <= 3; i++)
168         {
169             for(var j = 1; j <= 3; j++ )
170             {
171                 if(spielFeld[i,j] == leeresFeld)
172                 {
173                     return false;
174                 }
175             }
176         }
177         return spielFeldVoll;
178     }
179 }
180 }
```

**Musterlösung für Teilaufgabe 7:**

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace TicTacToe
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var leeresFeld = "#";
11             var kreuz = "X";
12             var kreis = "O";
13
14             string spieler1, spieler2;
15
16             var spielFeld = new string[4, 4]
17             {
18                 {" ", "1", "2", "3"}, 
19                 {"A", leeresFeld, leeresFeld, leeresFeld}, 
20                 {"B", leeresFeld, leeresFeld, leeresFeld}, 
21                 {"C", leeresFeld, leeresFeld, leeresFeld}
22             };
23
24             spieler1 = FrageSpielerNameAb(1);
25             spieler2 = FrageSpielerNameAb(2);
26
27             ZeigeSpielFeld(spielFeld);
28
29             while (true)
30             {
31                 Spielzug(spieler1, kreuz, spielFeld, leeresFeld);
32
33                 ZeigeSpielFeld(spielFeld);
34
35                 if(SpielFeldVoll(spielFeld, leeresFeld))
36                 {
37                     Console.WriteLine("Das Spiel endet
38                     unentschieden!");
39                     break;
40                 }
41                 if(HatGewonnen(kreuz,spielFeld))
42                 {
43                     Console.WriteLine($" {spieler1} hat
44                     gewonnen!");
45                     break;
46                 }
47
48                 Spielzug(spieler2, kreis, spielFeld, leeresFeld);
49
50                 ZeigeSpielFeld(spielFeld);
51
52                 if (SpielFeldVoll(spielFeld, leeresFeld))
53                 {
54                     Console.WriteLine("Das Spiel endet
```

## 8 Methoden schaffen Ordnung

```
55             unentschieden!");
56         break;
57     }
58     if (HatGewonnen(kreis, spielFeld))
59     {
60         Console.WriteLine(${spieler2} hat
61         gewonnen!");
62         break;
63     }
64 }
65
66 static string FrageSpielerNameAb(int spielerNummer)
67 {
68     Console.WriteLine($"Geben Sie einen Namen für Spieler
69     {spielerNummer} ein:");
70     return Console.ReadLine();
71 }
72
73
74 static void Spielzug(string name, string spielstein,
75 string[,] spielFeld, string leeresFeld)
76 {
77
78     var eingabeOk = false;
79     List<string> buchstaben = new List<string> { "A",
80         "B", "C" };
81     List<string> zahlen = new List<string> { "1", "2",
82         "3" };
83     string fehler = "Ungültiger Zug!";
84     string feldBesetzt = "Dieses Feld ist schon
85     besetzt!";
86
87     while(!eingabeOk)
88     {
89         Console.WriteLine($"{name}, geben Sie die
90         Koordinaten für Ihren Zug ein (Buchstabe,Zahl");
91
92     eingabe.Length != 3)
93     {
94         Console.WriteLine(fehler);
95         continue;
96     }
97
98     var koordinaten = eingabe.Split(",");
99
100    if(koordinaten.Length !=2)
101    {
102        Console.WriteLine(fehler);
103    }
104
105    if(!buchstaben.Contains(koordinaten[0]))
106    {
107        Console.WriteLine(fehler);
108        continue;
109    }
110 }
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

```
111         if(!zahlen.Contains(koordinaten[1]))
112     {
113         Console.WriteLine(fehler);
114         continue;
115     }
116
117     var i = buchstaben.IndexOf(koordinaten[0]) + 1;
118     var j = int.Parse(koordinaten[1]);
119
120     if(spielFeld[i,j] != leeresFeld)
121     {
122         Console.WriteLine(feldBesetzt);
123         continue;
124     }
125
126     spielFeld[i, j] = spielstein;
127
128     eingabeOk = true;
129 }
130
131
132 static void ZeigeSpielFeld(string[,] spielFeld)
133 {
134     for(var i = 0; i < 4; i++)
135     {
136         for(var j = 0; j < 4; j++)
137         {
138             Console.Write(spielFeld[i, j]);
139         }
140         Console.WriteLine();
141     }
142 }
143
144 static bool HatGewonnen(string spielStein, string[,]
145 spielFeld)
146 {
147     // 3 Steine in der 1. Reihe?
148     if (spielFeld[1, 1] == spielStein &&
149         spielFeld[1, 2] == spielStein &&
150         spielFeld[1, 3] == spielStein)
151         return true;
152
153     // 3 Steine in der 2. Reihe?
154     if (spielFeld[2, 1] == spielStein &&
155         spielFeld[2, 2] == spielStein &&
156         spielFeld[2, 3] == spielStein)
157         return true;
158
159     // 3 Steine in der 3. Reihe?
160     if (spielFeld[3, 1] == spielStein &&
161         spielFeld[3, 2] == spielStein &&
162         spielFeld[3, 3] == spielStein)
163         return true;
164
165     // 3 Steine in der 1. Spalte
166     if (spielFeld[1, 1] == spielStein &&
```

## 8 Methoden schaffen Ordnung

```
167         spielFeld[2, 1] == spielStein &&
168         spielFeld[3, 1] == spielStein)
169         return true;
170
171     // 3 Steine in der 2. Spalte
172     if (spielFeld[1, 2] == spielStein &&
173         spielFeld[2, 2] == spielStein &&
174         spielFeld[3, 2] == spielStein)
175         return true;
176
177     // 3 Steine in der 3. Spalte
178     if (spielFeld[1, 3] == spielStein &&
179         spielFeld[2, 3] == spielStein &&
180         spielFeld[3, 3] == spielStein)
181         return true;
182
183     // 3 Steine diagonal von links oben
184     // nach rechts unten
185     if (spielFeld[1, 1] == spielStein &&
186         spielFeld[2, 2] == spielStein &&
187         spielFeld[3, 3] == spielStein)
188         return true;
189
190     // 3 Steine diagonal von rechts oben
191     // nach links unten
192     if (spielFeld[1, 3] == spielStein &&
193         spielFeld[2, 2] == spielStein &&
194         spielFeld[3, 1] == spielStein)
195         return true;
196
197     return false;
198 }
199
200 static bool SpielFeldVoll(string[,] spielFeld, string
201 leeresFeld)
202 {
203     var spielFeldVoll = true;
204     for(var i = 1; i <= 3; i++)
205     {
206         for(var j = 1; j <= 3; j++ )
207         {
208             if(spielFeld[i,j] == leeresFeld)
209             {
210                 return false;
211             }
212         }
213     }
214     return spielFeldVoll;
215 }
216 }
217 }
```

## 8.6 Übungsaufgaben: Programmieren mit Methoden

The screenshot shows the Microsoft Visual Studio Debugging Console window. The console output is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie einen Namen für Spieler 1 ein:
Robert
Geben Sie einen Namen für Spieler 2 ein:
Katia
123
A###
B###
C###
Robert, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
x2
Ungültiger Zug!
Robert, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
A,1
123
AX##
B###
C###
Katia, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
A,2
123
AX##
B###
C###
Robert, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
C,3
123
AX##
B###
C##X
Katia, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
C,2
123
AX##
B###
C#OX
Robert, geben Sie die Koordinaten für Ihren Zug ein (Buchstabe,Zahl
B,2
123
AX##
B#X#
C#OX
Robert hat gewonnen!

C:\Program Files\dotnet\dotnet.exe (Prozess "6680") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
```

8

Abb. 8.6.1 Ausgabe des TIC-TAC-TOE-Spiels

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 9

# Grundlagen der Objektorientierten Programmierung

In diesem Kapitel lernen wir die Grundlagen der Objektorientierten Programmierung kennen. Lesern, die bereits Programmiererfahrung haben und nun zum ersten Mal mit Objektorientierung konfrontiert werden, geht es wahrscheinlich wie mir früher: Ich habe 1989 als Student neben dem Studium ein Projekt mit Turbo Pascal implementiert. Dann erschien Turbo Pascal zum ersten Mal in einer objektorientierten Version und ich habe begonnen, mich mit Objektorientierung zu beschäftigen. Bei den Handbüchern der neuen Version befand sich ein kleiner Band, der sich speziell nur mit der Objektorientierung beschäftigt hat. Ich hatte schnell verstanden, wie das formal funktionierte, aber mir drängte sich immer wieder die Frage auf: Wozu das Ganze? Was habe ich als Programmierer davon? Zu diesem Thema gab es in diesem Buch einen kleinen Abschnitt. Die Autoren rieten dazu, sich nicht um solche Fragestellungen zu kümmern, sondern die objektorientierte Programmierung einfach mal auszuprobieren. Die Objektorientierung würde zu spürbar weniger Seiteneffekten im Code und zu deutlich einfacherer Wiederverwendbarkeit von bestehenden Codes führen. Also habe ich es ausprobiert und mir eine einfache Klassenbibliothek zur Verwaltung von Programmfenstern geschrieben und musste feststellen: Es stimmt! Diese Klassenbibliothek konnte ich noch in einigen weiteren Projekten ohne nennenswerte Änderungen verwenden.

9

Die Objektorientierte Programmierung beruht auf den folgenden Konzepten, die dem Programmierer das Leben erleichtern.

### **Organisation in Objekten:**

Der Programmcode wird in Objekten organisiert und das kommt der menschlichen Denkweise entgegen. Wir Menschen denken in Objekten, wie zum Beispiel Fahrrad, Auto und Motorrad. Wir sehen auch Gemeinsamkeiten dieser Objekte: Alle der genannten Objekte sind Fahrzeuge. Derartige Sichtweisen lassen sich in der Objektorientierten Programmierung sehr treffend abbilden.

### **Konzept der Vererbung**

Ein Objekt kann von einem Objekt erben. Das entspricht dem Prinzip, etwas Neues zu erschaffen, indem man auf etwas Bestehendem aufbaut. Wenn wir einem bestehenden Fahrrad einen Motor hinzufügen, erhalten wir ein neues Motorrad. Das Konzept

## 9 Grundlagen der Objektorientierten Programmierung

der Vererbung ist ein wichtiges Hilfsmittel, um den Programmcode weiterverwendbar zu machen.

### Datenkapselung

Die Datenkapselung ist ein Konzept, um Programmcodes vor „unsachgemäßem“ Gebrauch zu schützen. Der Programmierer kann detailliert einstellen, welche Funktionalitäten seiner Objekte zugreifbar sind und welche nicht.

### Polymorphie

Objekte können vielgestaltig sein. Wenn man aus einem Fahrrad ein Motorrad machen will, genügt es nicht, einfach nur einen Motor einzubauen, sondern man muss auch Funktionalitäten des Fahrrads ändern, zum Beispiel muss der Fahrradsattel durch einen größeren und anders geformten Sattel ersetzt werden, damit das Motorrad bequem gefahren werden kann. In der Objektorientierten Programmierung können wir derartige Modifikationen durch Überschreiben von Funktionalitäten abbilden.

In diesem Sinne: Lernen Sie Objektorientierung kennen und machen Sie ein Projekt damit - Sie werden nie wieder anders programmieren wollen.

#### 9.1 Was ist Objektorientierte Programmierung?

Bisher haben wir die Beispielprogramme in diesem Buch unter dem Paradigma der prozeduralen Programmierung erstellt. Ab jetzt wechseln wir zum Objektorientierten Paradigma. Erinnern Sie sich: Im Kapitel über Verzweigungen habe ich Ihnen die Bedingung

```
1 alter >= 67 || beitragsJahre >= 45
```

vorgestellt und erklärt, dass wir damit eine verwaltungstechnische Vorschrift in einen mathematischen Ausdruck gegossen haben. Genau darum geht es bei der Softwareentwicklung: Ein Programmierer muss mit seinem Programm eine Realität abbilden. Eine Realität besteht aus Objekten, die miteinander interagieren. Ein Objekt kann alles Mögliche sein, wie zum Beispiel ein Bankkonto in einem Buchhaltungsprogramm, ein Avatar in einem Computerspiel oder ein Termin in einem Kalenderprogramm. Informationen, die ein Objekt und seinen Zustand beschreiben, nennt man Eigenschaften des Objekts. Ein Objekt mit dem Namen „Auto“ kann die Eigenschaft „Farbe“ haben und die Eigenschaft „Farbe“ kann dann wieder einen Wert haben wie zum Beispiel „Blau“. Eigenschaften stellen in der Objektorientierten Programmierung eine spezielle Form von Variablen dar. Eine Eigenschaft muss keinen primitiven Datentyp haben, sie kann auch eine Liste sein oder wieder ein Objekt oder eine Liste von Objek-

ten. So kann ein Objekt „Firma“ einfache Eigenschaften wie „FirmenName“ haben und auch komplizierte Eigenschaften wie „Belegschaft“, wobei „Belegschaft“ eine Liste von „Mitarbeiter“-Objekten ist.

Objekte werden mit Hilfe von Klassen erzeugt. Eine Klasse ist eine Blaupause, die beschreibt, wie ein Objekt aussieht, reserviert aber noch keinen Speicher für ihre Eigenschaften. Erst wenn man aus einer Klasse ein Objekt erzeugt, wird Speicher reserviert. Ein Objekt wird auch als Instanz einer Klasse bezeichnet. Das Erzeugen eines Objektes beschreiben wir auch als instanzieren. Zum Beispiel sind die Objekte „meinAuto“, „deinAuto“ und „nachbarsAuto“ Instanzen der Klasse „Auto“, sie haben alle die gleichen Eigenschaften, aber die Eigenschaften können unterschiedliche Werte haben. Objekte sind aber nicht nur elegant strukturierte Datenspeicher. Objekte können auch etwas tun – und dafür haben Objekte Methoden. Eine Klasse „Auto“ könnte Methoden wie „Fahren“, „Blinken“ oder „Hupen“ bereitstellen.

Bei diesem ersten kleinen Exkurs in die Theorie möchte ich es vorerst belassen. Im Kapitel „Objektorientierung für Fortgeschrittene“ werden wir einen weiteren Ausflug in die Theorie unternehmen. Jetzt kehren wir wieder in die Praxis zurück und betrachten noch einmal unser erstes „Hello World!“-Programm.

```
1  using System;
2
3  namespace HalloObjekt
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
```

9

Eigentlich haben wir hier schon eine Klasse: Unser Programm, das „Program“ heißt, ist eine Klasse. Das können wir am vorangestellten Schlüsselwort `class` erkennen. Allerdings werden in diesem Beispiel die Möglichkeiten der Objektorientierung nicht wirklich verwendet. Es gibt keinen Code, der eine Instanz der Klasse `Program` erstellt. Es gibt nur eine Methode `Main`, die statisch ist und daher direkt gerufen werden kann, ohne dass eine Instanz der Klasse erstellt wird. Das werden wir jetzt ändern.

```
1  using System;
2
3  namespace HalloObjekt
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
```

## 9 Grundlagen der Objektorientierten Programmierung

```
9         var meinProgramm = new Program();
10        meinProgramm.Hallo();
11    }
12
13    void Hallo()
14    {
15        Console.WriteLine("Hello Objektorientierung!");
16    }
17}
18}
```

Damit haben wir eine Objektorientierte Version des klassischen „Hello World!“-Programms erstellt. Die Ausgabe des Programms haben wir in die Methode `Hallo` ausgelagert. Bei der Methode `Hallo` fehlt jetzt das Schlüsselwort `static`. Das heißt, sie ist nicht mehr statisch und kann auch nicht mehr mit der Anweisung:

```
1 Hallo();
```

in der Hauptmethode gerufen werden. Wenn Sie es versuchen, meldet Ihnen bereits der Compiler einen Fehler. Stattdessen erzeugen wir eine Instanz der Klasse `Program` mit der Anweisung:

```
1 var meinProgramm = new Program();
```

Die Variable `meinProgramm` ist jetzt ein Objekt. Aber welchen Typ hat die Variable? Wenn Sie den Mauszeiger über den Variablennamen bewegen, zeigt Ihnen Visual Studio im Tooltip an, dass die Variable den Typ `Program` hat. In C# ist eine Klasse ein Variablentyp, den der Programmierer kreativ und frei gestalten kann. Erzeugt wird die Instanz mit dem Schlüsselwort `new`, gefolgt vom Namen der Klasse und einem Paar runder Klammern. Nachdem wir eine Instanz der Klasse `Program` erzeugt haben, rufen wir die Methode `Hallo` der Klasse über den Namen der Instanzvariablen auf.

```
1 meinProgramm.Hallo();
```

Sie können den Aufruf der Methode `Hallo` auch wie folgt verkürzen.

```
1 new Program().Hallo();
```

Da wir die Instanz der Klasse `Program` hier nur zum Aufruf der Methode `Hallo` benötigen, müssen wir sie nicht unbedingt einer Variablen zuweisen. Man nennt so etwas ein anonymes Objekt.

Rekapitulieren wir noch einmal die grundlegende Programmarchitektur einer Konsole-App:

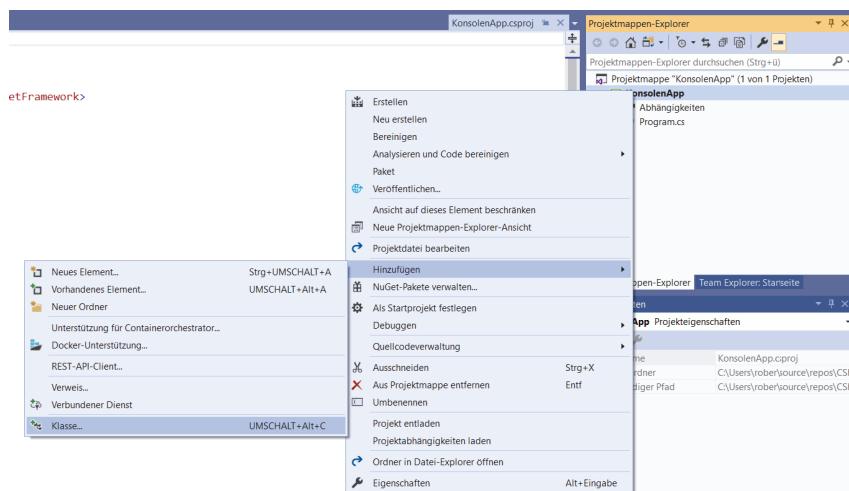
Eine Konsolen-App wird durch die Klasse `Program` abgebildet. Zum Start der Konsolen-App wird die statische Methode `Main` aufgerufen, der die Kommandozeilenargumente in der Form eines String-Arrays übergeben werden.

Diese grundlegende Architektur ist von den Entwicklern des .NET-Frameworks so festgelegt. Als Programmierer können wir diese Architektur verwenden, aber nicht verändern. Wie wir uns von dieser Architektur entkoppeln können und unsere eigene Architektur entwerfen können, wird im nächsten Kapitel vorgestellt.

## 9.2 Eine Klasse erstellen

Bisher haben wir immer nur mit einer Klasse gearbeitet. Ab jetzt wird es mehrere Klassen in einem Programm geben. Es ist zwar möglich, weitere Klassen in der Datei `Program.cs` unterzubringen, aber wir werden jede Klasse in einer eigenen Datei speichern. Dieses Verfahren sorgt für mehr Übersicht in unseren Programmen und hat auch entscheidende Vorteile, wenn wir Software gemeinsam mit anderen Programmierern im Team entwickeln.

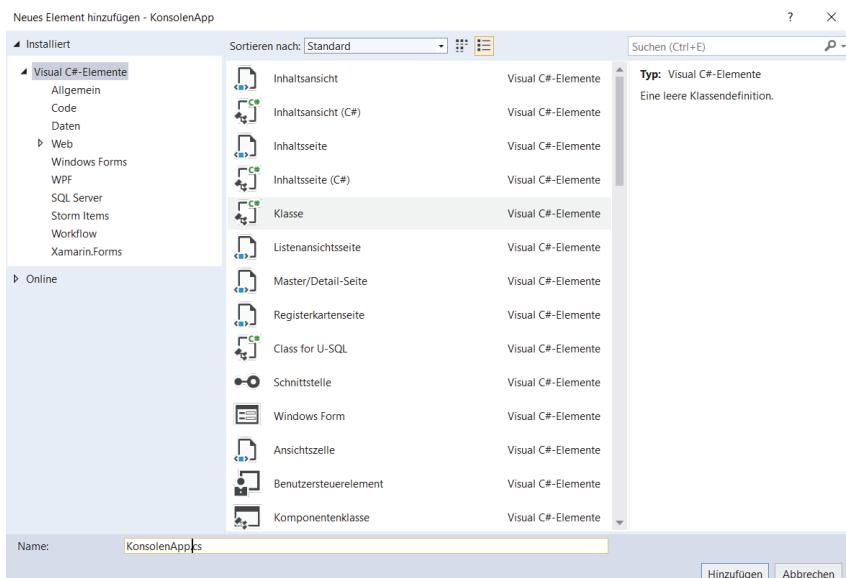
Zum Erstellen unserer eigenen Klasse erstellen wir zunächst eine neue Konsolen-App. Dann klicken wir mit der rechten Maustaste im Projektmappen-Explorer auf unser Projekt und wählen im Kontextmenü „Hinzufügen/Klasse“ aus.



**Abb. 9.2.1** Eine neue Klasse erstellen

Das Fenster „Neues Element hinzufügen“ erscheint.

## 9 Grundlagen der Objektorientierten Programmierung



**Abb. 9.2.2** Der Dialog „Neues Element hinzufügen“

Im Feld „Name“ vergeben wir den Namen „KonsolenApp.cs“ und klicken auf die Schaltfläche „Hinzufügen“. Visual Studio erstellt uns jetzt die Klasse KonsolenApp.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace KonsolenApp
6  {
7      class KonsolenApp
8      {
9      }
10 }
```

Zu Beginn werden die Klassenbibliotheken System, System.Collections.Generic und System.Text eingebunden. Die werden zunächst nicht benötigt. Es handelt sich hier um vorauselgenden Gehorsam von Visual Studio, da wir wahrscheinlich diese Klassenbibliotheken bei unserer weiteren Programmierung benötigen werden. Jetzt haben wir eine neue Klasse, die wir mit Leben füllen werden. Mit dieser Klasse können wir die Architektur unserer Konsolen-App frei gestalten.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
```

```
4 namespace KonsolenApp
5 {
6     class KonsolenApp
7     {
8         string begruessung = "Hallo Welt!";
9
10        public void Start()
11        {
12            Console.WriteLine(begruessung);
13        }
14    }
15}
16}
```

Bisher hatten wir Variablen nur innerhalb von Methoden deklariert. Man nennt derartige Variablen auch lokale Variablen, das heißt, sie sind nur innerhalb der Methode gültig, in der sie deklariert werden. Außerhalb der Methode existieren sie nicht. Die Klasse `KonsolenApp` deklariert nun eine Variable außerhalb ihrer Methoden, aber innerhalb der Klasse. Diese Variablen werden Member-Variablen genannt, sie sind innerhalb der ganzen Klasse, in der sie deklariert werden, gültig. Jede Methode der Klasse kann auf sie zugreifen. In der Klasse `KonsolenApp` deklarieren wir die Member-Variable `begruessung` als `string` und weisen ihr den Wert „Hallo Welt!“ zu. In der Methode `Start` greifen wir auf die Variable zu und geben sie aus. Der Methode `Start` ist das Schlüsselwort `public` vorangestellt. Das benötigen wir, damit die Methode `Start` von außerhalb der Klasse `KonsolenApp` gerufen werden kann. Wir können die Klasse `KonsolenApp` zwar völlig frei gestalten, aber zur Ausführung bringen können wir sie nur durch Aufrufen der Methode `Start` außerhalb der Klasse. Das erledigen wir in der Klasse `Program` in der Methode `Main`.

```
1 using System;
2
3 namespace KonsolenApp
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             new KonsolenApp().Start();
10        }
11    }
12}
```

Ohne das Schlüsselwort `public` vor der Methode „`Start`“ würde uns der Compiler für die Anweisung:

```
1 new KonsolenApp().Start();
```

einen Fehler melden. Damit haben wir die grundlegende Architektur für unsere Konsole-App von der Klasse `Program` entkoppelt. Wir können die Architektur unserer

## 9 Grundlagen der Objektorientierten Programmierung

Konsolen-Apps nun frei gestalten. In der Klasse `Program` erzeugen wir lediglich eine Instanz der Klasse `KonsolenApp` und rufen die Methode `Start` der Instanz auf.

### 9.3 Klassen mit Konstruktoren

Klassen haben spezielle Methoden, die nur dazu dienen, die Klasse zu erzeugen. Diese Methoden heißen Konstruktoren. Eine Klasse kann mehrere Konstruktoren haben, sie hat aber mindestens einen Konstruktor. Auch wenn wir für unsere Klasse „`KonsolenApp`“ keinen Konstruktor deklariert haben, so besitzt sie doch einen Konstruktor, den sogenannten Standard-Konstruktor. Diesen Konstruktor hat eine Klasse automatisch, wenn kein Konstruktor deklariert wird. Deklarieren wir jetzt einen Konstruktor für die Klasse „`KonsolenApp`“.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace KonsolenApp
6  {
7      class KonsolenApp
8      {
9          string begruessung = "Hallo Welt!";
10
11         public KonsolenApp()
12         {
13
14         }
15
16         public void Start()
17         {
18             Console.WriteLine(begruessung);
19         }
20     }
21 }
```

Der Konstruktor wird deklariert wie eine Methode, die den gleichen Namen hat wie die Klasse. Allerdings wird für einen Konstruktor kein Rückgabetyp angegeben. Der Rückgabetyp eines Konstruktors ist seine Klasse. Das Schlüsselwort `public` vor dem Konstruktor ist nicht zwingend vorgeschrieben. In unserem Fall benötigen wir es, weil wir den Konstruktor von außerhalb der Klasse `KonsolenApp` rufen, wenn wir eine Instanz der Klasse in der Methode `Main` der Klasse `Program` erzeugen. Sobald eine Klasse einen eigenen Konstruktor definiert, steht der Standard-Konstruktor nicht mehr zur Verfügung. Allerdings funktioniert unser Konstruktor genauso wie der Standard, er nimmt keine Parameter entgegen und tut nichts, außer die Klasse zu instanziieren. Füllen wir unseren Konstruktor mal mit Leben.

```
1  using System;
2  using System.Collections.Generic;
```

```
3  using System.Text;
4
5  namespace KonsolenApp
6  {
7      class KonsolenApp
8      {
9          string begruessung = "Hallo Welt!";
10
11         public KonsolenApp(string begruessung)
12         {
13             this.begruessung = begruessung;
14         }
15
16         public void Start()
17         {
18             Console.WriteLine(begruessung);
19         }
20     }
21 }
```

Da Konstruktoren Methoden sind, können sie auch Übergabeparameter entgegennehmen. Unser Konstruktor deklariert den String-Parameter `begruessung`. Der Konstruktor nimmt den Parameter entgegen und weist ihn der Member-Variablen `begruessung` zu und überschreibt damit deren Inhalt. Da der Übergabeparameter genauso heißt wie die Member-Variable, müssen wir beim Zugriff der Member-Variablen das Schlüsselwort `this` und einen Punkt voranstellen, damit der Compiler weiß, was die Member-Variable und was der Übergabeparameter ist.

9

Genauso wie Methoden können auch Konstruktoren überladen werden.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace KonsolenApp
6  {
7      class KonsolenApp
8      {
9          string begruessung = "Hallo Welt!";
10
11         public KonsolenApp()
12         {
13
14         }
15
16         public KonsolenApp(string begruessung)
17         {
18             this.begruessung = begruessung;
19         }
20
21         public void Start()
22         {
23             Console.WriteLine(begruessung);
24         }
25     }
26 }
```

## 9 Grundlagen der Objektorientierten Programmierung

```
24     }
25 }
26 }
```

Jetzt verfügt unsere Klasse über zwei Konstruktoren. Einen parameterlosen Konstruktor und einen Konstruktor mit Übergabeparameter. Wird KonsolenApp mit dem parameterlosen Konstruktor erzeugt, so gibt die Methode Start den Text „Hallo Welt!“ am Bildschirm aus. Verwenden wir dagegen den Konstruktor mit Übergabeparameter, so wird nach dem Aufruf der Methode Start der Wert des Übergabeparameters ausgegeben.

```
1 using System;
2
3 namespace KonsolenApp
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             new KonsolenApp().Start();
10            new KonsolenApp("Hello World!").Start();
11        }
12    }
13 }
```

In dieser Variante erstellen wir in der Main-Methode zwei Instanzen der Klasse „Konsolen-App“, jeweils mit einem anderen Konstruktor. Zuerst verwenden wir den parameterlosen Konstruktor und für das zweite Objekt verwenden wir den Konstruktor mit Übergabeparameter. Die Ausgabe am Bildschirm sieht dann wie folgt aus:

Hello Welt!

Hello World!

### 9.4 Eine Instanz einer Klasse erzeugen und verwenden

In diesem Kapitel möchte ich die verschiedenen Varianten vorstellen, mit denen man eine Klasse erzeugen und verwenden kann. Als Beispiel benutzen wir eine leicht modifizierte Variante der bereits bekannten Klasse KonsolenApp:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonsolenApp
6 {
7     class KonsolenApp
8     {
9         public string Begruebung = "Hallo Welt!";
10    }
```

## 9.4 Eine Instanz einer Klasse erzeugen und verwenden

```
11     public KonsolenApp()
12     {
13     }
14
15
16     public KonsolenApp(string begruessung)
17     {
18         this.Begruebung = begruessung;
19     }
20
21     public void Start()
22     {
23         Console.WriteLine(Begruebung);
24     }
25 }
26 }
```

Es gibt zwei Änderungen an der Klasse. Der Member-Variablen `Begruebung` ist jetzt das Schlüsselwort `public` vorangestellt und zudem wurde der Variablenname großgeschrieben. Das Schlüsselwort `public` benötigen wir, damit auf die Membervariable auch von außen zugegriffen werden kann. Die Großschreibung hat technisch keine Auswirkungen, allerdings ist es eine Konvention unter C#-Programmierern, dass Member-Variablen großgeschrieben werden, wenn sie `public` deklariert sind. Die erste Variante, ein Objekt zu erzeugen, ist ein dreistufiger Vorgang:

```
1 var konsole = new KonsolenApp();
2 konsole.Begruebung = "Hola Mundo!";
3 konsole.Start();
```

9

In der Stufe eins erzeugen wir das Objekt mit einem parameterlosen Konstruktor und weisen es der Variablen `konsole` zu. In der zweiten Stufe weisen wir dem Objekt die für seine Funktion notwendigen Daten zu. Bei unserem Objekt ist das ein Wert für die Member-Variable `Begruebung`. In der dritten Stufe lassen wir das Objekt aktiv werden. In diesem Beispiel rufen wir die Methode `Start` des Objekts auf.

```
1 Für die zweite Variante benötigen wir nur zwei Stufen:
2 var konsole = new KonsolenApp("Hello World");
3 konsole.Start();
```

Die erste Stufe erzeugt das Objekt mit einem Konstruktor, der die Daten entgegen nimmt, die für das Objekt benötigt werden. Die zweite Stufe ruft die Methode `Start` auf.

Für die dritte Variante müssen wir die Klasse `KonsolenApp` etwas erweitern. Wir fügen ihr eine sogenannte Fabrik-Methode hinzu. Das ist eine statische Methode, die ein Objekt der Klasse `KonsolenApp` erzeugt. Die Methode muss statisch sein, damit wir sie direkt über die Klasse aufrufen können, ohne vorher ein Objekt zu erzeugen. Das Erzeugen des Objekts soll die Fabrik-Methode übernehmen.

## 9 Grundlagen der Objektorientierten Programmierung

```
1 public static KonsolenApp InSpanisch()
2 {
3     var instanz = new KonsolenApp("Hola Mundo!");
4     return instanz;
5 }
```

Der Rückgabewert der Fabrik-Methode ist natürlich `KonsolenApp`. Wir wollen das Objekt ja schließlich erzeugen. In der Fabrik-Methode erzeugen wir das Objekt mit dem einem Konstruktor, der den Wert „Hola Mundo!“ entgegennimmt und dann geben wir das Objekt zurück. Damit haben wir eine Fabrik-Methode, die ein `KonsolenApp`-Objekt erzeugt, das mit einer spanischen Begrüßung initialisiert ist. Bei der Verwendung des Objekts müssen keine Daten zur Initialisierung zugewiesen werden.

```
1 var konsole = KonsolenApp.InSpanisch();
2 konsole.Start();
```

Natürlich können wir den Aufruf auch auf eine Zeile verkürzen:

```
1 KonsolenApp.InSpanisch().Start();
```

Auch wenn wir hier eine statische Methode aufrufen, wird ein Objekt erzeugt, welches uns ermöglicht, die Methode `Start` aufzurufen. Da das Objekt keinen Namen im Programm hat, nennt man es ein anonymes Objekt.

Für die letzte Variante, die ich hier vorstellen möchte, ist unsere Klasse `KonsolenApp` zu trivial. In dieser Variante erzeugen wir ein Objekt, indem wir zuerst eine Fabrik-Klasse erzeugen, die uns eine Methode bereitstellt, mit der wir unser eigentliches Objekt erzeugen können. Unser eigentliches Objekt definiert die Klasse „`Begruesser`“.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ObjektFabrik
6 {
7     public class Begruesser
8     {
9         string begruessung;
10
11         public Begruesser(string begruessung)
12         {
13             this.begruessung = begruessung;
14         }
15
16         public void Hallo()
17         {
18             Console.WriteLine(begruessung);
19         }
20     }
21 }
```

## 9.4 Eine Instanz einer Klasse erzeugen und verwenden

Die Klasse `Begruesser` stellt einen Konstruktor zur Verfügung, der einen String entgegennimmt. In diesem String wird eine Begrüßung erwartet, die von der Methode `Hallo` am Bildschirm ausgegeben wird. Aber die Klasse `Begruesser` ist so geduldig, wie das Papier im Sprichwort. Sie können dem Konstruktor eine Begrüßung in jeder beliebigen Sprache mitgeben oder noch schlimmer: einen Text wie „Ich bin ein Biba-Butzemann“, der gar keine Begrüßung ist. Unsere Beispielklasse `Begruesser` ist sehr einfach. Und jeder Kollege kann sich sofort erschließen, wie er sie im Sinne des Erfinders verwenden soll. Aber bei Klassen, die komplizierter konstruiert werden, ist das nicht sofort ersichtlich und es gibt viele Möglichkeiten des versehentlichen Missbrauchs. Davor können wir uns mit einer Fabrikklasse schützen.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace ObjektFabrik
6  {
7      public enum Sprache
8      {
9          Deutsch,
10         Englisch,
11         Spanisch
12     }
13
14     public class BegruesserFabrik
15     {
16         Sprache sprache;
17
18         public BegruesserFabrik(Sprache sprache)
19         {
20             this.sprache = sprache;
21         }
22
23         public Begruesser ErzeugeBegruesser()
24         {
25             switch(sprache)
26             {
27                 case Sprache.Englisch:
28                     return new Begruesser("Hello World!");
29                 case Sprache.Deutsch:
30                     return new Begruesser("Hallo Welt!");
31                 case Sprache.Spanisch:
32                     return new Begruesser("Holla Mundo!");
33                 default:
34                     return new Begruesser($"Ich kann kein
35                     {sprache}!");
36             }
37         }
38     }
39 }
```

## 9 Grundlagen der Objektorientierten Programmierung

Als Unterstützung für die Klasse `BegruesserFabrik` deklarieren wir die Aufzählung `Sprache` mit den Elementen `Deutsch`, `Englisch` und `Spanisch`. Diese drei Sprachen wollen wir unterstützen. Der Konstruktor der Klasse nimmt einen Parameter vom Typ `Sprache` entgegen, damit können wir sicherstellen, dass die Klasse `BegruesserFabrik` nur `Begruesser`-Objekte in einer der drei unterstützten Sprachen erzeugt. Das Erzeugen der `Begruesser`-Objekte übernimmt die Fabrikmethode `ErzeugeBegruesser()`. Je nachdem, welche Sprache beim Erzeugen von `BegruesserFabrik` festgelegt wurde, erzeugt die Methode `ErzeugeBegruesser()` ein `Begruesser`-Objekt mit einer Begrüßung in einer anderen Sprache. Falls jemand in der Aufzählung `Sprache` eine weitere Sprache hinzufügt, und vergisst, diese Sprache in der Fabrikmethode zu berücksichtigen, so erzeugt die Fabrikmethode ein `Begruesser`-Objekt, das eine entsprechende Meldung am Bildschirm ausgibt.

Wenn wir jetzt ein `Begruesser`-Objekt benötigen, erzeugen wir zuerst ein `BegruesserFabrik`-Objekt mit der gewünschten Sprache und rufen dann die Fabrikmethode des Fabrik-Objekts auf, um ein `Begruesser`-Objekt zu erhalten.

```
1  using System;
2
3  namespace ObjektFabrik
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var fabrik = new BegruesserFabrik(Sprache.Deutsch);
10             var begruesser = fabrik.ErzeugeBegruesser();
11             begruesser.Hallo();
12         }
13     }
14 }
```

Natürlich kann man auch hier wieder mit anonymen Objekten das Erzeugen der Objekte und den Aufruf der Methode `Hallo` auf eine Zeile verkürzen.

```
1  new BegruesserFabrik(Sprache.Englisch).ErzeugeBegruesser().
2  Hallo();
```

### 9.5 Zugriffsmodifizierer für Klassen, Eigenschaften und Methoden

Wie wir schon gesehen haben, müssen wir vor ein Element einer Klasse das Schlüsselwort `public` schreiben, wenn wir es außerhalb der Klasse verwenden wollen. `public` ist ein sogenannter Zugriffsmodifizierer. Jede Klasse, jede Member-Variable, jeder Konstruktor und jede Methode hat so einen Zugriffsmodifizierer. Wenn wir keinen Zugriffsmodifizierer angeben, verwendet C# einen Standard-Zugriffsmodifizierer. Allerdings ist dieser Standard-Zugriffsmodifizierer nicht in allen Situationen der

gleiche. Daher ist es empfehlenswert, immer einen Zugriffsmodifizierer anzugeben. Ab jetzt werden wir bei allen Beispielen immer Zugriffsmodifizierer verwenden.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace ObjektFabrik
6  {
7      public class Begruesser
8      {
9          private string begruessung;
10
11         public Begruesser(string begruessung)
12         {
13             this.begruessung = begruessung;
14         }
15
16         public void Hallo()
17         {
18             Console.WriteLine(begruessung);
19         }
20     }
21 }
```

9

Das ist unsere bereits bekannte Klasse `Begruesser` mit einem Zugriffsmodifizierer bei allen Elementen. Die Member-Variable `begruessung` hat den Zugriffsmodifizierer `private`, da sie nur innerhalb ihrer Klasse verwendet werden darf. Übergabeparameter und lokale Variablen benötigen keinen Zugriffsmodifizierer, da ihre Gültigkeit generell auf die jeweilige Methode beschränkt ist.

C# kennt noch die Zugriffsmodifizierer `protected` und `internal`, aber die sind für das, was wir bisher besprochen haben, nicht relevant. Auf diese Zugriffsmodifizierer gehen wir ein, sobald wir sie benötigen.

## 9.6 Properties statt Variablen verwenden

In diesem Kapitel beschäftigen wir uns mit den sogenannten Properties. Properties sind eigentlich eine Form von syntactic sugar, die das Verwenden des sogenannten Getter-/Setter-Entwurfsmusters vereinfachen. Daher betrachten wir zuerst dieses Entwurfsmuster. Zunächst sehen wir uns eine einfache Klasse an, die Teil einer Ampelsteuerung sein könnte.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Ampel
6  {
```

## 9 Grundlagen der Objektorientierten Programmierung

```
7     public enum Ampelfarbe
8     {
9         Rot,
10        Gelb,
11        Gruen
12    }
13
14    public class Ampel
15    {
16        public Ampelfarbe Ampelfarbe;
17    }
18 }
```

Die Klasse `Ampel` hat nur eine `public`-Membervariable namens `Ampelfarbe` vom Typ `Ampelfarbe`, was eine Aufzählung mit den Werten `Rot`, `Gelb` und `Gruen` ist. Eine Ampel-Instanz erzeugen wir mit:

```
1 var eineAmpel = new Ampel();
```

und mit

```
1 eineAmpel.Ampelfarbe = Ampelfarbe.Rot;
```

setzen wir die Ampel auf die Farbe `Rot`. Bis hierher ziemlich unspektakulär. Wenn wir jetzt eine Anforderung an unsere Ampelsteuerung bekommen, die von uns verlangt, eine Statistik-Funktion in die Ampelsteuerung zu integrieren, die die Anzahl der `Rot`-phasen von jeder Ampel mitzählt, dann müssen wir an allen Stellen des Programms für die Ampelsteuerung, an denen eine Ampel auf `Rot` gestellt wird, einen Programmcode einfügen, der einen Zähler für unsere Statistik hochzählt. Bei einer komplexen Ampelsteuerung für eine gesamte Großstadt wird das schnell zu einer Herkulesaufgabe, die noch dazu sehr fehleranfällig ist. Wenn man nun als Programmierer einmal in so einer Situation ist, kommt man da nicht so einfach wieder heraus. In so einem Fall bleibt einem nichts anderes übrig, als sein gesamtes Programm nach den relevanten Stellen zu durchsuchen und die nötigen Änderungen zu machen.

Allerdings hätte man bei der Programmierung von Anfang an etwas anders machen können, um gar nicht erst in diese Situation zu kommen. Das führt uns zum Getter-/Setter-Entwurfsmuster. Unter Berücksichtigung des Getter-/Setter-Entwurfsmusters hätten wir die Klasse `Ampel` von Anfang an wie folgt implementiert:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Ampel
6 {
7     public enum Ampelfarbe
8     {
9         Rot,
```

```
10     Gelb,
11     Gruen
12 }
13
14 public class Ampel
15 {
16     private Ampelfarbe ampelfarbe;
17
18     public Ampelfarbe GetAmpelfarbe()
19     {
20         return ampelfarbe;
21     }
22
23     public void SetAmpelFarbe(Ampelfarbe ampelfarbe)
24     {
25         this.ampelfarbe=ampelfarbe;
26     }
27 }
28 }
```

Die Membervariable `ampelfarbe` ist jetzt privat und deswegen auch klein geschrieben. Von außen kann nicht mehr auf `ampelfarbe` zugegriffen werden. Aber mit den `public`-Methoden `GetAmpelfarbe()` und `SetAmpelFarbe()` kann die Membervariable gelesen und geschrieben werden. Diese beiden Methoden nennt man auch den Getter und den Setter der Membervariablen `ampelfarbe`. Mit der Anweisung:

```
1 eineAmpel.SetAmpelFarbe(Ampelfarbe.Rot);
```

kann man zum Beispiel eine Ampel auf Rot stellen. Mit so einer Ampel-Klasse gibt es auch für die neue Anforderung mit der Statistikfunktion eine einfache Lösung.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Ampel
6 {
7     public enum Ampelfarbe
8     {
9         Rot,
10        Gelb,
11        Gruen
12    }
13
14 public class Ampel
15 {
16     private Ampelfarbe ampelfarbe;
17     private int anzahlRotPhasen;
18
19     public Ampelfarbe GetAmpelfarbe()
20     {
21         return ampelfarbe;
```

## 9 Grundlagen der Objektorientierten Programmierung

```

22     }
23
24     public void SetAmpelFarbe(Ampelfarbe ampelfarbe)
25     {
26         this.ampelfarbe=ampelfarbe;
27
28         if(ampelfarbe == Ampelfarbe.Rot)
29         {
30             anzahlRotPhasen++;
31         }
32     }
33
34     public int GetAnzahlRotPhasen()
35     {
36         return anzahlRotPhasen;
37     }
38 }
39 }
```

Um die Rotphasen einer Ampel zu zählen, führen wir die Membervariable `anzahlRotPhasen` ein. Sie ist `privat`, damit sie nicht unkontrolliert von außerhalb der Klasse `Ampel` verändert werden kann. Aber lesenden Zugriff von außen wollen wir erlauben, daher implementieren wir die `public`-Methode `GetAnzahlRotPhasen` als Getter. Im Setter von `ampelfarbe` erledigen wir das Hochzählen der Membervariablen `anzahlRotPhasen`.

```

1 if(ampelfarbe == Ampelfarbe.Rot)
2 {
3     anzahlRotPhasen++;
4 }
```

Wenn wir an den Setter von `ampelfarbe` den Wert `Ampelfarbe.Rot` übergeben, erhöhen wir den Wert von `anzahlRotPhasen` um eins. Damit haben wir die Statistikfunktionalität in der Klasse `Ampel` gekapselt.

Ob unsere Ampel-Klasse korrekt funktioniert, überprüfen wir mit einem kleinen Testprogramm:

```

1 using System;
2
3 namespace Ampel
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var eineAmpel = new Ampel();
10
11             eineAmpel.SetAmpelFarbe(Ampelfarbe.Rot);
12             eineAmpel.SetAmpelFarbe(Ampelfarbe.Gelb);
13             eineAmpel.SetAmpelFarbe(Ampelfarbe.Gruen);
14             eineAmpel.SetAmpelFarbe(Ampelfarbe.Rot);
```

```

15             Console.WriteLine($"Die Anzahl der Rotphasen ist:
16                 {eineAmpel.GetAnzahlRotPhasen()}");
17         }
18     }
19 }
20 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the following text:

```

Microsoft Visual Studio-Debugging-Konsole
Die Anzahl der Rotphasen ist: 2
C:\Users\rober\source\repos\CSharpBuch\Ampel\Ampel\bin\Debug\netcoreapp3.1\Ampel.exe (Prozess "20080") wurde mit Code "0"
beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 9.6.1 Ausgabe des Ampel-Testprogramms mit Getter-/Setter-Entwurfsmuster

Aufgrund dieser Erfahrungen und Überlegungen wurde in der Objektorientierten Programmierung schon sehr früh begonnen, generell auf `public`-Membervariablen zu verzichten und das Getter/Setter-Entwurfsmuster zu verwenden. In C# gibt es für dieses Getter-/Setter-Entwurfsmuster syntactic sugar in Form von sogenannten Properties zur Vereinfachung. Mit Properties hätten wir die Klasse Ampel zunächst so implementiert:

```

1  using System;
2  using System.Collections.Generic;
3  using System;
4  using System.Collections.Generic;
5  using System.Text;
6
7  namespace Ampel
8  {
9      public enum Ampelfarbe
10     {
11         Rot,
12         Gelb,
13         Gruen
14     }
15
16     public class Ampel
17     {
18         private Ampelfarbe ampelfarbe;
19
20         public Ampelfarbe Ampelfarbe
21         {
22             get
23             {
24                 return ampelfarbe;
25             }
26             set
27             {
28                 ampelfarbe = value;
29             }
30         }
31     }
```

## 9 Grundlagen der Objektorientierten Programmierung

32 }

Wir haben eine `private`-Membervariable, auf die nicht von außen zugegriffen werden kann. Und für den Zugriff von außen haben wir eine sogenannte Property mit einem `public`-Zugriffsmodifizierer. Die Property hat, genau wie eine Membervariable, einen Typ – in unserem Fall: `Ampelfarbe`. Der Property folgen ein Getter, gekennzeichnet durch das Schlüsselwort `get`, und ein Setter, gekennzeichnet durch das Schlüsselwort `set`. Dem Getter und dem Setter folgen, jeweils umgeben von geschweiften Klammern, ein Codeblock, der beim Lesen beziehungsweise beim Schreiben der Property ausgeführt werden soll. Der Setter kennt zusätzlich noch das Schlüsselwort `value`, das für den dem Setter zugewiesenen Wert steht. Zusammengefasst ist das eine `private`-Membervariable und ein Getter und ein Setter für den Zugriff – nur mit einer anderen Syntax. Jetzt wollen wir die Vorteile dieser neuen Syntax untersuchen. Dazu schauen wir uns den Zugriff auf die Property an:

```
1 var eineAmpel = new Ampel();
2 eineAmpel.Ampelfarbe = Ampelfarbe.Rot;
3 var ampelfarbe = eineAmpel.Ampelfarbe;
```

Der Lese- und Schreibzugriff erfolgt identisch wie der Zugriff auf `public`-Membervariablen. Das bedeutet: Wenn wir alte Programme haben, die `public`-Membervariablen verwenden, können wir daraus Properties machen, ohne dass wir den Programmcode, der diese `public`-Membervariablen verwendet, ändern müssen. Das ist ein riesiger Vorteil dieser Syntax gegenüber dem klassischen Getter/Setter Entwurfsmuster. Wenn wir eine Property haben, bei der (noch) kein Programmcode beim Lesen und/oder Schreiben ablaufen soll, können wir diese Property auch vereinfacht deklarieren:

```
1 public Ampelfarbe Ampelfarbe { get; set; }
```

Die `private`-Membervariable erzeugt der Compiler dann im Hintergrund automatisch. Falls wir dann eine neue Anforderung erhalten, die es nötig macht, Programmcodes beim Lesen und/oder Schreiben der Property einzuführen, müssen wir dann eine `private`-Membervariable ergänzen. Zusätzlich hat die Property-Syntax noch ein paar interessante Features:

```
1 public Ampelfarbe Ampelfarbe { get; }
```

Wenn wir den Setter weglassen, haben wir eine `readonly`-Property. Nur der Konstruktor der zugehörigen Klasse kann darauf schreibend zugreifen:

```
1 public Ampelfarbe Ampelfarbe { get; private set; }
```

Jetzt haben wir eine Property mit einem public-Getter und mit einem private-Setter. Das heißt, die Property kann außerhalb der Klasse gelesen, aber nur innerhalb der Klasse geschrieben werden.

Betrachten wir nun die Klasse Ampel mit integrierter Statistik-Funktionalität in der Property-Syntax:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Ampel
6  {
7      public enum Ampelfarbe
8      {
9          Rot,
10         Gelb,
11         Gruen
12     }
13
14     public class Ampel
15     {
16         public int AnzahlRotphasen { get; private set; }
17
18         private Ampelfarbe ampelfarbe;
19
20         public Ampelfarbe Ampelfarbe
21         {
22             get
23             {
24                 return ampelfarbe;
25             }
26             set
27             {
28                 ampelfarbe = value;
29
30                 if(ampelfarbe == Ampelfarbe.Rot)
31                 {
32                     AnzahlRotphasen++;
33                 }
34             }
35         }
36     }
37 }
```

9

Die Anzahl der Rotphasen speichern wir in der public-Property `AnzahlRotphasen`. Damit ihr Wert nicht von außerhalb der Klasse überschrieben werden kann, hat sie einen private-Setter. Den Wert für die Ampelfarbe speichern wir in der public-Property `Ampelfarbe`, damit wir sie von außen setzen können. Der Setter der Property `Ampelfarbe` erhöht bei jedem Schreibzugriff die Property `AnzahlRotPhasen` um eins, wenn `Ampelfarbe` auf `Ampelfarbe.Rot` gesetzt wird. Bei der Verwendung der

## 9 Grundlagen der Objektorientierten Programmierung

Klasse Ampel können wir die Property Ampelfarbe genauso benutzen, als wäre sie eine public-Membervariable.

```

1  using System;
2
3  namespace Ampel
4  {
5      class Program
6      {
7          public Ampelfarbe Ampelfarbe { get; private set; }
8
9          static void Main(string[] args)
10         {
11             var eineAmpel = new Ampel();
12             eineAmpel.Ampelfarbe = Ampelfarbe.Rot;
13             eineAmpel.Ampelfarbe = Ampelfarbe.Gelb;
14             eineAmpel.Ampelfarbe = Ampelfarbe.Gruen;
15             eineAmpel.Ampelfarbe = Ampelfarbe.Rot;
16
17             Console.WriteLine($"Die Anzahl der Rotphasen
18             ist:{eineAmpel.AnzahlRotphasen}");
19         }
20     }
21 }
```

```

Microsoft Visual Studio-Debugging-Konsole
Die Anzahl der Rotphasen ist: 2
C:\Users\rober\source\repos\CSharpBuch\Ampel\Ampel\bin\Debug\netcoreapp3.1\Ampel.exe (Prozess "20080") wurde mit Code "0"
beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 9.6.2 Ausgabe des Ampel-Testprogramms mit Property-Syntax

Zum Schluss des Kapitels möchte ich Ihnen noch eine Diskussion vorstellen, die immer wieder in diversen Internetforen für Programmierer auftaucht. Dabei geht es um die folgende Frage:

„Eigentlich könnte ich die Eigenschaften einer neuen Klasse zunächst als `public`-Membervariablen modellieren und später, wenn ich zusätzlich einen Programmcode benötige, der beim Lesen und/oder Schreiben einer Membervariable ausgeführt werden soll, mache ich aus den entsprechenden `public`-Membervariablen einfach Properties. Spricht irgendetwas gegen diese Technik?“

Nachdem, was wir bis jetzt gelernt haben, spricht tatsächlich nichts gegen diese Vorgehensweise. Aber in der professionellen Software-Entwicklung gibt es ein Argument, das dagegen spricht und erklärt, warum wir auf die Verwendung von `public`-Membervariablen grundsätzlich verzichten sollten. Stattdessen sollten wir bei Properties, die zunächst beim Schreiben und/oder Lesen keinen Programmcode benötigen, die verkürzte Schreibweise verwenden:

```
1 public int Zaehler { get; set; }
```

Der Grund dafür ist folgender: Wenn wir eine `public`-Membervariable in eine Property umwandeln, verlieren wir die Binärkompatibilität. Um zu erklären, welche Probleme das verursachen kann, muss etwas weiter ausgeholt werden. Wenn wir zum Beispiel die Methode `Console.WriteLine()` verwenden, verwenden wir eine Methode des .NET-Frameworks. Sie wird von der Klassenbibliothek „System.Console.dll“ zur Verfügung gestellt. Diese Datei befindet sich auf unserem Computer und wurde durch die Installation des .NET-Frameworks dort installiert.

Wenn wir ein Programm entwickeln und in kompilierter Form an Benutzer ausliefern, und unser Programm die Methode `Console.WriteLine()` verwendet, wird `Console.WriteLine()` beim Benutzer des Programms von der Klassenbibliothek „System.Console.dll“ aufgerufen, die auf dem Computer des Benutzers installiert ist. „System.Console.dll“ muss auf dem Computer des Benutzers aber nicht exakt identisch mit „System.Console.dll“ auf unserem Computer sein. Zum Beispiel könnte unser Benutzer eine neuere Version der Klassenbibliothek haben, die von einem etwas neueren Update des .NET-Frameworks auf seinem Computer installiert wurde oder umgekehrt.

Wenn Microsoft keine dramatischen Änderungen bei „System.Console.dll“ gemacht hat, funktioniert unser geliefertes Programm trotzdem. Jetzt gibt es aber nicht nur „System.Console.dll“, sondern jede Menge anderer Klassenbibliotheken im .NET-Framework und es gibt auch die Möglichkeit, dass die von uns entwickelte Software eigene Klassenbibliotheken mit ausliefert oder dass wir Klassenbibliotheken von irgendwelchen Fremdherstellern mitliefern, wofür wir wiederum Updates an unsere Benutzer schicken könnten, um eine Fehlerkorrektur des Fremdherstellers an unsere Benutzer zu liefern.

Wenn jetzt Microsoft oder ein anderer Hersteller einer Klassenbibliothek in seinem Produkt eine `public`-Membervariable verwendet hätte und in der neuen Version diese `public`-Membervariable in eine Property geändert hätte und wir zufällig diese Membervariable in unserem Programm verwendet hätten, würde unser Programm nicht mehr funktionieren, da unser Programm eine Membervariable erwartet, aber eine Property vorfindet.

Ein größeres Softwareprodukt besteht normalerweise nicht nur aus einer ausführbaren Datei, sondern bringt noch etliche vom Programmierer erstellte Klassenbibliotheken mit sich. Und oft genügt es, wenn Sie für eine Fehlerkorrektur nicht das ganze Produkt neu liefern, sondern nur eine korrigierte Klassenbibliothek. Und wenn Sie dann in der korrigierten Klassenbibliothek eine Membervariable in eine Property ändern, funktioniert ihr Programm nicht mehr.

## 9 Grundlagen der Objektorientierten Programmierung

Wenn Sie ein Programm schreiben, das nur von ein paar Kollegen innerhalb Ihrer Abteilung verwendet wird, ist das kein großes Problem. Eine Korrektur durch erneutes Kompilieren und erneutes Ausliefern kann schnell erledigt werden. Wenn Ihnen der Fehler bei einem Softwarepaket unterläuft, das bei Tausenden von Kunden und von Hunderttausenden von Benutzern verwendet wird, ist das eine mittlere Katastrophe. Daher verwenden Programmierer, die einen professionellen Anspruch an sich selbst haben, prinzipiell keine `public`-Membervariablen.

### 9.7 Operatoren überladen

Operatoren in C# kennen wir bereits. Zum Beispiel kennen wir die Operatoren `+, *, /`, für Ganzzahlen und Gleitkommazahlen. Wir haben auch schon gesehen, dass wir den Operator `+` auch für Strings verwenden können. Bei Strings führt der Operator `+` eine sogenannte Konkatenation durch, das heißt, er hängt die Strings hintereinander. Der Operator `+` hat also unterschiedliche Funktionalitäten, je nachdem, welchen Typ die Operanden haben, auf die er angewendet wird. Mit der Operatorüberladung können wir für eigene Typen (Klassen) eigene Funktionalitäten für Operatoren definieren.

Eine Klasse `Bruch`, um mathematische Brüche darzustellen, können wir uns schnell und einfach erstellen:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Brueche
6 {
7     public class Bruch
8     {
9         public int Zaehler { get; set; }
10        public int Nenner { get; set; }
11    }
12 }
13 }
```

Zwei `public`-Properties für Zähler und Nenner des Bruchs genügen und wir können Brüche speichern. Natürlich könnten wir jetzt noch Methoden schreiben, um Brüche zu addieren, zu subtrahieren, zu multiplizieren und zu dividieren. Aber es wäre wesentlich eleganter und praktischer, wenn wir dafür die mathematischen Operatoren `+, *, /` verwenden könnten, so wie bei Ganzzahlen und Gleitkommazahlen auch.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Brueche
6 {
```

```
7  public class Bruch
8  {
9      public int Zaehler { get; set; }
10     public int Nenner { get; set; }
11
12     public Bruch(int zaehler, int nenner)
13     {
14         Zaehler = zaehler;
15         Nenner = nenner;
16     }
17
18     public static Bruch operator +(Bruch a)
19     {
20         return a;
21     }
22
23     public static Bruch operator -(Bruch a)
24     {
25         return new Bruch(-a.Zaehler, a.Nenner);
26     }
27
28     public static Bruch operator +(Bruch a, Bruch b)
29     {
30         return new Bruch(a.Zaehler * b.Nenner + b.Zaehler *
31             a.Nenner, a.Nenner * b.Nenner);
32     }
33
34     public static Bruch operator -(Bruch a, Bruch b)
35     {
36         return a + (-b);
37     }
38
39     public static Bruch operator *(Bruch a, Bruch b)
40     {
41         return new Bruch(a.Zaehler * b.Zaehler, a.Nenner *
42             b.Nenner);
43     }
44
45     public static Bruch operator /(Bruch a, Bruch b)
46     {
47         return new Bruch(a.Zaehler * b.Nenner, a.Nenner *
48             b.Zaehler);
49     }
50 }
51 }
```

9

Zuerst erhält die Klasse Bruch einen Konstruktor, der Werte für Zähler und Nenner entgegennimmt. Das erleichtert uns die Verwendung der Klasse etwas. Dann deklarieren wir für jeden Operator, den wir für den Typ Bruch überladen wollen, eine eigene Methode. Methoden, die Operatoren überladen, müssen statisch sein. Der Name einer Methode, die einen Operator überlädt, ist der Operator selbst. Vor den Namen der Methode schreiben wir das Schlüsselwort `operator`. Der Rückgabetyp einer Operatormethode ist der Typ, für den die Überladung gelten soll, in unserem Beispiel

## 9 Grundlagen der Objektorientierten Programmierung

Bruch. Die Übergabeparameter von Operatormethoden haben ebenfalls den Typ, für den die Überladung gelten soll – also auch Bruch. Ein Operator kann mehrere Überladungen für einen Typ haben. In unserem Beispiel überladen wir die Operatoren + und - jeweils zweimal. Einmal für die Verwendung mit einem Operanden und einmal für die Verwendung mit zwei Operanden.

Die erste Operatormethode überlädt den Operator + für die Verwendung mit einem Operanden, das entspricht dem Ausdruck `+einBruch`, wobei die Variable `einBruch` vom Typ `Bruch` ist. Wenn wir für `einBruch` den Wert  $\frac{1}{2}$  annehmen, so entspricht das dem Ausdruck  $+\frac{1}{2}$ , und da  $+\frac{1}{2}$  gleich  $\frac{1}{2}$  ist, gibt die Operatormethode einfach nur den übergebenen Parameter zurück.

Die zweite Operatormethode überlädt den Operator - für die Verwendung mit einem Operanden, das entspricht dem Ausdruck `-einBruch`, wobei die Variable `einBruch` vom Typ `Bruch` ist. Wenn wir für `einBruch` den Wert  $\frac{1}{2}$  annehmen, so entspricht das dem Ausdruck  $-\frac{1}{2}$ . Das Voranstellen eines Minus-Zeichens entspricht in der Mathematik einer Multiplikation mit -1. Wird ein Bruch mit -1 multipliziert, so wird einfach das Vorzeichen des Zählers umgedreht und genau das tun wir in dieser Operatormethode.

Die dritte Operatormethode ist eine Überladung des Operators + für zwei Operanden, das entspricht dem Ausdruck `ersterBruch + zweiterBruch`, wobei `ersterBruch` und `zweiterBruch` den Typ `Bruch` haben. Wenn wir für `ersterBruch` den Wert  $\frac{1}{2}$  und für `zweiterBruch` den Wert  $\frac{1}{4}$  annehmen, so entspricht das dem Ausdruck  $\frac{1}{2} + \frac{1}{4}$ . Um die beiden Brüche zu addieren, müssen wir jeden Bruch mit dem Nenner des jeweils anderen Bruchs multiplizieren:

$$\frac{1 \cdot 4}{2 \cdot 4} + \frac{1 \cdot 2}{4 \cdot 2} \text{ Das ergibt: } \frac{4}{8} + \frac{2}{8} = \frac{6}{8}$$

Die Mathematiker unter Ihnen erkennen jetzt natürlich, dass wir das Ergebnis noch kürzen sollten und somit  $\frac{3}{4}$  erhalten würden. Allerdings ist der Algorithmus zum Kürzen eines Bruchs nicht trivial und da er zum Verständnis der Operatorüberladung nicht beiträgt, soll er an dieser Stelle ausgespart werden. So ergibt sich für unser Beispiel für den Zähler die Formel: `a.Zahler * b.Nenner + b.Zahler * a.Nenner` und für den Nenner ergibt sich `a.Nenner * b.Nenner`.

Die vierte Operatorüberladung der Klasse `Bruch` überlädt den Operator - für zwei Operanden, was dem Ausdruck `ersterOperand - zweiterOperand` entspricht. Mit einem kleinen mathematischen Kunstriff schreiben wir das so: `ersterOperand + (-zweiterOperand)`. Damit haben wir den Operator - für zwei Operanden durch eine Kombination aus dem Operator + für zwei Operanden und dem Operator - für einen Operanden ersetzt. Diese beiden Operatoren haben wir für den Typ `Bruch` bereits definiert und können sie hier gleich verwenden. Es genügt

daher für den Operator - für zwei Operanden, einfach den Ausdruck  $a + (-b)$  zurückgeben. Hier verzichten wir auch aus den oben genannten Gründen auf das finale Kürzen des Bruchs.

Die fünfte Operatorüberladung ist die Überladung des Operators \* mit zwei Operanden, sie entspricht dem Ausdruck `ersterOperand * zweiterOperand`, wobei `ersterOperand` und `zweiterOperand` jeweils vom Typ `Bruch` sind. Brüche werden multipliziert, indem die Zähler der beiden Operanden miteinander multipliziert werden, das ergibt den Zähler des Ergebnisbruchs. Den Nenner des Ergebnisbruchs erhält man durch Multiplizieren der Nenner der beiden Operanden. Auch hier wird auf das finale Kürzen des Bruchs verzichtet.

Die letzte Operatorüberladung ist die Überladung des Operators / mit zwei Operanden, sie entspricht dem Ausdruck `ersterOperand / zweiterOperand`, wobei `ersterOperand` und `zweiterOperand` jeweils vom Typ `Bruch` sind. Brüche werden dividiert, indem man den Zähler des ersten Operanden mit dem Nenner des zweiten Operanden multipliziert, das ergibt den Zähler des Ergebnisbruchs. Den Nenner des Ergebnisbruchs erhält man durch Multiplizieren des Nenners des ersten Operanden mit dem Zähler des zweiten Operanden. Auch hier verzichten wir wieder auf das finale Kürzen des Bruchs.

9

Mit dem folgenden Programm können wir unsere Klasse `Bruch` testen:

```
1  using System;
2
3  namespace Brueche
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var a = new Bruch(1, 2);
10             var b = new Bruch(1, 4);
11
12             var plusa = +a;
13             var minusa = -a;
14             var summe = a + b;
15             var differenz = a - b;
16             var produkt = a * b;
17             var quotient = a / b;
18
19             Console.WriteLine($"a = {a.Zaehler}/{a.Nenner}");
20             Console.WriteLine($"b = {b.Zaehler}/{b.Nenner}");
21             Console.WriteLine($"+a = {plusa.Zaehler}/{plusa.
Nenner}");
22             Console.WriteLine($"-a = {minusa.Zaehler}/{minusa.
Nenner}");
23             Console.WriteLine($"a + b = {summe.Zaehler}/{summe.
Nenner}");
24             Console.WriteLine($"a - b = {differenz.Zaehler}/
```

## 9 Grundlagen der Objektorientierten Programmierung

```

28         {differenz.Nenner});
29         Console.WriteLine($"a * b = {produkt.Zaehler}/
30             {produkt.Nenner}");
31         Console.WriteLine($"a / b = {quotient.Zaehler}/
32             {quotient.Nenner}");
33     }
34 }
35 }
```

```

Microsoft Visual Studio-Debugging-Konsole
a = 1/2
b = 1/4
+a = 1/2
-a = -1/2
a + b = 6/8
a - b = 2/8
a * b = 1/8
a / b = 4/2

C:\Users\rober\source\repos\Brueche\Brueche\bin\Debug\netcoreapp3.1\Brueche.exe (Prozess "16240") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 9.7.1 Ausgabe des Testprogramms für die überladenen Operatoren

## 9.8 Übungsaufgabe: Mit Objekten programmieren

Nachdem wir die Grundlagen der Objektorientierten Programmierung behandelt haben, vertiefen wir dieses Wissen mit einer umfangreicheren praktischen Übung. Wir werden Schritt für Schritt eine Anwendung zum Verwalten von Telefonnummern und E-Mail-Adressen entwickeln. Unsere Anwendung soll das Neuanlegen, Ändern, Suchen und Anzeigen von Telefonnummern und E-Mail-Adressen beherrschen. Wir werden diese Anwendung objektorientiert schreiben und dabei auf die Verwendung von public-Membervariablen verzichten.

### Teilaufgabe 1:

Schreiben Sie eine Klasse für einen Verzeichniseintrag. Die Klasse soll Vorname, Nachname, Festnetznummer, Mobilfunknummer und E-Mailadresse für einen Eintrag speichern können. Zudem soll die Klasse die Methode `ZeigeEintrag()` die, die Daten eines Eintrags anzeigt, zur Verfügung stellen.

### Teilaufgabe 2:

Schreiben Sie die Klasse `Verzeichnis`. Sie soll ein Dictionary zum Speichern der Verzeichniseinträge enthalten. Als Schlüssel für das Dictionary wird der String `Vorname:Nachname` verwendet. Die Klasse `Verzeichnis` soll zudem im Konstruktor das Dictionary mit ein paar Verzeichniseinträgen initialisieren. Die Klasse soll die public-Methode `NeuerEintrag()` enthalten die, die String-Parameter `vorname`, `nachname`, `festnetz`, `mobilfunk` und `email` entgegennimmt und

damit einen neuen Eintrag im Dictionary erstellt. Außerdem soll die Klasse die Methode `ZeigeAlleEintraege()`, die alle Verzeichniseinträge am Bildschirm ausgibt, bereitstellen.

### Teilaufgabe 3:

Erweitern Sie die Klasse `Verzeichnis` um die folgenden drei Methoden:

Die Methode `ZeigeEintrag()` nimmt den String-Parameter `schluessel` entgegen und überprüft, ob dieser Schlüssel im Dictionary enthalten ist. Wenn ja, gibt die Methode den Verzeichniseintrag am Bildschirm aus, wenn nein, eine entsprechende Meldung, dass es für diesen Schlüssel keinen Eintrag im Verzeichnis gibt.

Die Methode `ZeigeEintraegeFuerNachname()` nimmt den String-Paramater `nachname` entgegen und gibt alle Einträge mit diesem Nachnamen am Bildschirm aus. Werden keine Einträge für den entsprechenden Nachnamen gefunden, so wird eine Meldung ausgegeben, dass es für diesen Nachnamen keine Einträge im Verzeichnis gibt.

Die Methode `ZeigeEintraegeFuerVorname()` nimmt den String-Paramater `vorname` entgegen und gibt alle Einträge mit diesem Vornamen am Bildschirm aus. Werden keine Einträge für den entsprechenden Vornamen gefunden, so wird eine Meldung ausgegeben, dass es für diesen Vornamen keine Einträge im Verzeichnis gibt.

9

### Teilaufgabe 4:

Erweitern Sie die Klasse `Verzeichnis`: Schreiben Sie die Methode `AktualisiereEintrag()`, die die String-Parameter `schluessel`, `festnetz`, `mobilfunk` und `email` entgegennimmt. Die Methode soll den Verzeichniseintrag, der dem übergebenen Schlüssel entspricht, mit den Werten in den entsprechenden übergebenen Parametern aktualisieren. Wenn der übergebene Schlüssel nicht im Verzeichnis gefunden wird, soll die Methode eine entsprechende Meldung ausgeben.

Schreiben Sie die Methode `LoescheEintrag()`. Die Methode soll den String-Parameter `schluessel` entgegennehmen und den Verzeichniseintrag, der dem übergebenen Schlüssel entspricht, aus dem Verzeichnis entfernen. Wenn der übergebene Schlüssel nicht im Verzeichnis gefunden wird, soll die Methode eine entsprechende Meldung ausgeben.

### Teilaufgabe 5:

Schreiben Sie die Klasse `VerzeichnisApp`. Die Klasse besitzt die private-Membervariable `verzeichnis` von Typ `Verzeichnis`. Im parameterlosen public-Konstruktur der Klasse `VerzeichnisApp` weisen Sie der Membervariablen `verzeichnis` eine neue Instanz der Klasse `Verzeichnis` zu.

## 9 Grundlagen der Objektorientierten Programmierung

Des Weiteren besitzt die Klasse `VerzeichnisApp` die Methode `ZeigeMenu()`, welches den Bildschirm löscht und das folgende Menü am Bildschirm ausgibt:

1. Alle Einträge anzeigen.
  2. Einen bestimmten Eintrag anzeigen
  3. Einträge suchen nach Vornamen.
  4. Einträge suchen nach Nachnamen.
  5. Eintrag hinzufügen.
  6. Eintrag ändern.
  7. Eintrag löschen
  8. Programm beenden.
- 

Wählen Sie einen Menüpunkt (1-8)

Das Löschen des Bildschirms können Sie mit der Methode `Console.Clear()` erledigen.

Schreiben Sie die private-Methode `LeseMenuPunkt()`, welche eine Eingabe von der Tastatur einliest und einen Integer-Wert zwischen eins und acht für den entsprechenden Menüpunkt zurückgibt. Die Methode soll den Benutzer wiederholt zur Eingabe einer Zahl zwischen eins und acht auffordern, solange bis der Benutzer eine gültige Eingabe macht.

Schreiben Sie die public-Methode `Start`, welche in einer Schleife erst das Programm-Menü anzeigt und dann mit Hilfe der Methode `LeseMenuPunkt()` einen Menüpunkt von der Tastatur einliest und dann den Menüpunkt ausführt. Implementieren Sie in diesem Schritt nur die Ausführung des Menüpunkts acht: Programm beenden. Bei Eingabe anderer Menüpunkte soll die Schleife wieder mit dem Aufruf des Programm-Menüs beginnen.

Rufen Sie die Methode `Start` in der Hauptmethode der Klasse `Programm` auf.

### Teilaufgabe 6:

Erweitern Sie die Klasse `VerzeichnisApp` um die folgenden private-Methoden:

Die parameterlose Methode `ZeigeAlleEintraege()` hat keinen Rückgabewert. Sie soll den Bildschirm löschen und dann alle Einträge des Verzeichnisses ausgeben. Danach soll die Methode eine Zeile mit 52 Minuszeichen ausgeben und dann den folgenden Text: „Drücken Sie die Enter-Taste, um zum Menü zu gelangen.“ Zum Schluss soll die Methode noch eine Benutzereingabe abfragen.

Die parameterlose Methode `ZeigeEintrag()` hat keinen Rückgabewert. Sie soll den Bildschirm löschen und dann einen Schlüssel (Vorname:Nachname) des Verzeichnisses vom Benutzer abfragen. Die Methode soll anhand des Schlüssels einen Eintrag im Verzeichnis suchen und ihn am Bildschirm ausgeben. Wird der Schlüssel im Verzeichnis nicht gefunden, so soll eine entsprechende Meldung am Bildschirm ausgegeben werden. Danach soll die Methode eine Zeile mit 52 Minuszeichen ausgeben und dann der folgende Text: „Drücken Sie die Enter-Taste, um zum Menü zu gelangen.“ Zum Schluss soll die Methode noch eine Benutzereingabe abfragen.

Die parameterlose Methode `SucheEintragNachVornamen()` hat keinen Rückgabewert. Sie soll den Bildschirm löschen und dann einen Vornamen vom Benutzer abfragen. Die Methode soll alle Einträge im Verzeichnis suchen, die dem eingegebenen Vornamen entsprechen und sie am Bildschirm ausgeben. Wird kein Eintrag im Verzeichnis gefunden, so soll eine entsprechende Meldung am Bildschirm ausgegeben werden. Danach soll die Methode eine Zeile mit 52 Minuszeichen ausgeben und dann den folgenden Text: „Drücken Sie die Enter-Taste, um zum Menü zu gelangen.“ Zum Schluss soll die Methode noch eine Benutzereingabe abfragen.

Die parameterlose Methode `SucheEintragNachNachnamen()` hat keinen Rückgabewert. Sie soll den Bildschirm löschen und dann einen Nachnamen vom Benutzer abfragen. Die Methode soll alle Einträge im Verzeichnis suchen, die dem eingegebenen Nachnamen entsprechen und sie am Bildschirm ausgeben. Wird kein Eintrag im Verzeichnis gefunden, so soll eine entsprechende Meldung am Bildschirm ausgegeben werden. Danach soll die Methode eine Zeile mit 52 Minuszeichen ausgeben und dann den folgenden Text: „Drücken Sie die Enter-Taste, um zum Menü zu gelangen.“ Zum Schluss soll die Methode noch eine Benutzereingabe abfragen.

Verwenden Sie, soweit möglich, innerhalb der Methoden andere Methoden, die Sie bereits in den vorangegangenen Teilaufgaben geschrieben haben. Die obigen Methoden haben alle am Ende einen identischen Vorgang. Kapseln Sie diesen Vorgang in der private-Methode `Zurueck`.

Erweitern Sie die Methode `Start` der Klasse `VerzeichnisApp`. Implementieren Sie den Aufruf der Menüpunkte 1-4.

### Teilaufgabe 7:

In der letzten Teilaufgabe erweitern Sie die Klasse `VerzeichnisApp` um die folgenden Methoden:

## 9 Grundlagen der Objektorientierten Programmierung

Die parameterlose Methode `FuegeEintragHinzu()` hat keinen Rückgabewert. Sie löscht zuerst den Bildschirm und fragt dann nacheinander einen Vornamen, einen Nachnamen, eine Festnetznummer, eine Mobilfunknummer und eine E-Mailadresse vom Benutzer ab. Die Methode legt mit diesen Daten einen neuen Verzeichniseintrag an. Zum Schluss wird die Methode `Zurueck` aufgerufen, die in Teilaufgabe 6 erstellt wurde.

Die parameterlose Methode `AendereEintrag()` hat keinen Rückgabewert. Sie löscht zuerst den Bildschirm und fragt dann nacheinander einen Schlüssel, eine Festnetznummer, eine Mobilfunknummer und eine E-Mailadresse des Benutzers ab. Die Methode ändert mit diesen abgefragten Daten den Verzeichniseintrag, der dem abgefragten Schlüssel entspricht. Zum Schluss wird die Methode `Zurueck` aufgerufen, die in Teilaufgabe 6 erstellt wurde.

Die parameterlose Methode `LoescheEintrag()` hat keinen Rückgabewert. Sie löscht zuerst den Bildschirm und fragt anschließend einen Schlüssel vom Benutzer ab. Die Methode löscht den Verzeichniseintrag, der dem abgefragten Schlüssel entspricht. Zum Schluss wird die Methode `Zurueck` aufgerufen, die in Teilaufgabe 6 erstellt wurde.

Verwenden Sie, soweit möglich, innerhalb der Methoden andere Methoden, die Sie bereits in den vorangegangenen Teilaufgaben geschrieben haben.

Erweitern Sie die Methode `Start` der Klasse `VerzeichnisApp`. Implementieren Sie den Aufruf der Menüpunkte 5-7.

Damit haben Sie alle Menüpunkte implementiert und somit die Anwendung fertig gestellt.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class VerzeichnisEintrag
8      {
9          public string Vorname {get; set;}
10         public string Nachname { get; set; }
11         public string Festnetz { get; set; }
12         public string Mobilfunk { get; set; }
13         public string Email { get; set; }
14
15         public VerzeichnisEintrag(string vorname, string
16             nachname, string festnetz, string mobilfunk, string
17             email)
18         {
19             Vorname = vorname;
20             Nachname = nachname;
21             Festnetz = festnetz;
22             Mobilfunk = mobilfunk;
23             Email = email;
24         }
25
26         public void ZeigeEintrag()
27         {
28             Console.WriteLine($"{Vorname} {Nachname}; Festnetz:
29                 {Festnetz}; Mobilfunk: {Mobilfunk}; Email: {Email}");
30         }
31     }
32 }
```

9

## 9 Grundlagen der Objektorientierten Programmierung

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class Verzeichnis
8      {
9          private Dictionary<string, VerzeichnisEintrag>
10         verzeichnis = new Dictionary<string,
11         VerzeichnisEintrag>();
12
13         public Verzeichnis()
14     {
15         NeuerEintrag("Robert", "Schiefele", "08912356489",
16         "01715589665", "robert.schiefele@email.de");
17         NeuerEintrag("Katia", "Japa", "0897589445",
18         "017259894", "katia.japa@email.de");
19         NeuerEintrag("Theo", "Tester", "089885595445",
20         "017365894", "theo.tester@email.de");
21     }
22
23     public void NeuerEintrag(string vorname, string
24     nachname, string festnetz, string mobilfunk, string
25     email)
26     {
27         verzeichnis.Add(${vorname}:{nachname}", new
28         VerzeichnisEintrag(vorname, nachname, festnetz,
29         mobilfunk, email));
30     }
31
32     public void ZeigeAlleEintraege()
33     {
34         foreach(var eintrag in verzeichnis.Values)
35         {
36             eintrag.ZeigeEintrag();
37         }
38     }
39 }
```

**Musterlösung für Teilaufgabe 3:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class Verzeichnis
8      {
9          private Dictionary<string, VerzeichnisEintrag>
10         verzeichnis = new Dictionary<string,
11 VerzeichnisEintrag>();
12
13         public Verzeichnis()
14         {
15             NeuerEintrag("Robert", "Schiefele", "08912356489",
16             "01715589665", "robert.schiefele@email.de");
17             NeuerEintrag("Katia", "Japa", "0897589445",
18             "017259894", "katia.japa@email.de");
19             NeuerEintrag("Theo", "Tester", "089885595445",
20             "017365894", "theo.tester@email.de");
21         }
22
23         public void NeuerEintrag(string vorname, string
24             nachname, string festnetz, string mobilfunk, string
25             email)
26         {
27             verzeichnis.Add(${vorname}:{nachname}", new
28             VerzeichnisEintrag(vorname, nachname, festnetz,
29             mobilfunk, email));
30         }
31
32         public void ZeigeAlleEintraege()
33         {
34             foreach(var eintrag in verzeichnis.Values)
35             {
36                 eintrag.ZeigeEintrag();
37             }
38         }
39
40         public void ZeigeEintrag(string schluessel)
41         {
42             if(verzeichnis.ContainsKey(schluessel))
43             {
44                 verzeichnis[schluessel].ZeigeEintrag();
45             }
46             else
47             {
48                 Console.WriteLine($"Eintrag: {schluessel} nicht
49                 gefunden.");
50             }
51         }
52
53         public void ZeigeEintraegeFuerNachname(string nachname)
54         {
```

## 9 Grundlagen der Objektorientierten Programmierung

```
55         var gefunden = false;
56
57         foreach(var element in verzeichnis)
58         {
59             if(element.Value.Nachname == nachname)
60             {
61                 element.Value.ZeigeEintrag();
62                 gefunden = true;
63             }
64         }
65
66         if(!gefunden)
67         {
68             Console.WriteLine($"Keinen Eintrag für {nachname}
69             gefunden");
70         }
71     }
72
73     public void ZeigeEintraegeFuerVorname(string vorname)
74     {
75         var gefunden = false;
76
77         foreach (var element in verzeichnis)
78         {
79             if (element.Value.Vorname == vorname)
80             {
81                 element.Value.ZeigeEintrag();
82                 gefunden = true;
83             }
84         }
85
86         if (!gefunden)
87         {
88             Console.WriteLine($"Keinen Eintrag für {vorname}
89             gefunden");
90         }
91     }
92 }
93 }
```

## Musterlösung für Teilaufgabe 4

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class Verzeichnis
8      {
9          private Dictionary<string, VerzeichnisEintrag> verzeichnis
10         = new Dictionary<string, VerzeichnisEintrag>();
11
12         public Verzeichnis()
13         {
14             NeuerEintrag("Robert", "Schiefele", "08912356489",
15             "01715589665", "robert.schiefele@email.de");
16             NeuerEintrag("Katia", "Japa", "0897589445",
17             "017259894", "katia.japa@email.de");
18             NeuerEintrag("Theo", "Tester", "089885595445",
19             "017365894", "theo.tester@email.de");
20         }
21
22         public void NeuerEintrag(string vorname, string
23             nachname, string festnetz, string mobilfunk, string email)
24         {
25             verzeichnis.Add(${vorname}:{nachname}", new
26             VerzeichnisEintrag(vorname, nachname, festnetz,
27             mobilfunk, email));
28         }
29
30         public void ZeigeAlleEintraege()
31         {
32             foreach(var eintrag in verzeichnis.Values)
33             {
34                 eintrag.ZeigeEintrag();
35             }
36         }
37
38         public void ZeigeEintrag(string schluessel)
39         {
40             if(verzeichnis.ContainsKey(schluessel))
41             {
42                 verzeichnis[schluessel].ZeigeEintrag();
43             }
44             else
45             {
46                 Console.WriteLine($"Eintrag: {schluessel} nicht
47                 gefunden.");
48             }
49         }
50
51         public void ZeigeEintraegeFuerNachname(string nachname)
52         {
53             var gefunden = false;
```

## 9 Grundlagen der Objektorientierten Programmierung

```
55         foreach(var element in verzeichnis)
56         {
57             if(element.Value.Nachname == nachname)
58             {
59                 element.Value.ZeigeEintrag();
60                 gefunden = true;
61             }
62         }
63
64         if(!gefunden)
65         {
66             Console.WriteLine($"Keinen Eintrag für {nachname}
67             gefunden");
68         }
69     }
70
71     public void ZeigeEintraegeFuerVorname(string vorname)
72     {
73         var gefunden = false;
74
75         foreach (var element in verzeichnis)
76         {
77             if (element.Value.Vorname == vorname)
78             {
79                 element.Value.ZeigeEintrag();
80                 gefunden = true;
81             }
82         }
83
84         if (!gefunden)
85         {
86             Console.WriteLine($"Keinen Eintrag für {vorname}
87             gefunden");
88         }
89     }
90
91     public void AktualisiereEintrag(string schluessel,
92     string festnetz, string mobilfunk, string email)
93     {
94         if(verzeichnis.ContainsKey(schluessel))
95         {
96             verzeichnis[schluessel].Festnetz = festnetz;
97             verzeichnis[schluessel].Mobilfunk = mobilfunk;
98             verzeichnis[schluessel].Email = email;
99         }
100        else
101        {
102            Console.WriteLine($"Eintrag: {schluessel} nicht
103            gefunden.");
104        }
105    }
106
107    public void LoescheEintrag(string schluessel)
108    {
109        if(verzeichnis.ContainsKey(schluessel))
110        {
```

## 9.8 Übungsaufgabe: Mit Objekten programmieren

```
111         verzeichnis.Remove(schlüssel);
112     }
113     else
114     {
115         Console.WriteLine($"Eintrag: {schlüssel} nicht
116                     gefunden.");
117     }
118 }
119 }
120 }
```

## 9 Grundlagen der Objektorientierten Programmierung

## Musterlösung für Teilaufgabe 5

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class VerzeichnisApp
8      {
9          private Verzeichnis verzeichnis;
10
11         public VerzeichnisApp()
12         {
13             verzeichnis = new Verzeichnis();
14         }
15
16         private void ZeigeMenu()
17         {
18             Console.Clear();
19             Console.WriteLine("1. Alle Einträge anzeigen.");
20             Console.WriteLine("2. Einen bestimmten Eintrag
21                 anzeigen");
22             Console.WriteLine("3. Einträge suchen nach
23                 Vornamen.");
24             Console.WriteLine("4. Einträge suchen nach
25                 Nachnamen.");
26             Console.WriteLine("5. Eintrag hinzufügen.");
27             Console.WriteLine("6. Eintrag ändern.");
28             Console.WriteLine("7. Eintrag löschen");
29             Console.WriteLine("8. Programm beenden.");
30             Console.
31             WriteLine("-----");
32             Console.WriteLine("Wählen Sie einen Menüpunkt (1-
33             8)");
34         }
35
36         private int LeseMenuPunkt()
37         {
38             int menupunkt;
39             do
40             {
41                 if (int.TryParse(Console.ReadLine(), out
42                     menupunkt))
43                 {
44                     if (menupunkt >= 1 && menupunkt <= 8)
45                     {
46                         break;
47                     }
48                 }
49                 Console.WriteLine("Wählen Sie einen Menüpunkt (1-
50                     8)");
51             }
52             while (1==1);
53
54             return menupunkt;
```

## 9.8 Übungsaufgabe: Mit Objekten programmieren

```

55     }
56
57     public void Start()
58     {
59         var ende = false;
60         while(!ende)
61         {
62             ZeigeMenu();
63
64             switch(LeseMenuPunkt())
65             {
66                 case 8:
67                     ende = true;
68                     break;
69             }
70         }
71     }
72 }
73 }
```

```

1 using System;
2
3 namespace TelefonVerzeichnisObjektorientiert
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             new VerzeichnisApp().Start();
10        }
11    }
12 }
```

9



The screenshot shows a terminal window with a black background and white text. At the top, the path 'C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...' is visible. Below it, a numbered list of menu items is displayed:

1. Alle Einträge anzeigen.
2. Einen bestimmten Eintrag anzeigen
3. Einträge suchen nach Vornamen.
4. Einträge suchen nach Nachnamen.
5. Eintrag hinzufügen.
6. Eintrag ändern.
7. Eintrag löschen.
8. Programm beenden.

Below this list is a separator line '-----'. At the bottom, the text 'Wählen Sie einen Menüpunkt (1-8)' is followed by a cursor character at position 8.

Abb. 9.8.1 Musterlösung Teilaufgabe 5: Auswahl Menüpunkt 8

## 9 Grundlagen der Objektorientierten Programmierung



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar reads "Microsoft Visual Studio-Debugging-Konsole". The content area displays a list of menu items numbered 1 through 8, followed by a separator line and the instruction "Wählen Sie einen Menüpunkt (1-8)". Below this, the number "8" is entered. The console then outputs the command "C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerzeichnisObjektorientiert.exe (Prozess "26616") wurde mit Code "0" beendet." It also provides instructions for closing the console automatically when debugging ends: "Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen". Finally, it says "Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen."

Abb. 9.8.2 Musterlösung Teilaufgabe 5: Ausführung Menüpunkt 8

**Musterlösung für Teilaufgabe 6:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class VerzeichnisApp
8      {
9          private Verzeichnis verzeichnis;
10
11         public VerzeichnisApp()
12         {
13             verzeichnis = new Verzeichnis();
14         }
15
16         private void ZeigeMenu()
17         {
18             Console.Clear();
19             Console.WriteLine("1. Alle Einträge anzeigen.");
20             Console.WriteLine("2. Einen bestimmten Eintrag
21                 anzeigen");
22             Console.WriteLine("3. Einträge suchen nach
23                 Vornamen.");
24             Console.WriteLine("4. Einträge suchen nach
25                 Nachnamen.");
26             Console.WriteLine("5. Eintrag hinzufügen.");
27             Console.WriteLine("6. Eintrag ändern.");
28             Console.WriteLine("7. Eintrag löschen");
29             Console.WriteLine("8. Programm beenden.");
30             Console.
31             WriteLine("-----");
32             Console.WriteLine("Wählen Sie einen Menüpunkt (1-
33                 8)");
34         }
35
36         private int LeseMenuPunkt()
37         {
38             int menupunkt;
39             do
40             {
41                 if (int.TryParse(Console.ReadLine(), out
42                     menupunkt))
43                 {
44                     if (menupunkt>=1 && menupunkt<=8)
45                     {
46                         break;
47                     }
48                 }
49                 Console.WriteLine("Wählen Sie einen Menüpunkt (1-
50                     8)");
51             }
52             while (1==1);
53
54             return menupunkt;
```

## 9 Grundlagen der Objektorientierten Programmierung

```
55
56
57     }
58
59     private void ZeigeAlleEintraege()
60     {
61         Console.Clear();
62         verzeichnis.ZeigeAlleEintraege();
63         Zurueck();
64     }
65
66     private void ZeigeEintrag()
67     {
68         Console.Clear();
69         Console.WriteLine("Geben Sie einen Schlüssel
70             (Vorname:Nachname) ein.");
71         verzeichnis.ZeigeEintrag(Console.ReadLine());
72         Zurueck();
73     }
74
75     private void SucheEintragNachVornamen()
76     {
77         Console.Clear();
78         Console.WriteLine("Geben Sie einen Vornamen ein.");
79         verzeichnis.ZeigeEintraegeFuerVorname(Console.
80         ReadLine());
81         Zurueck();
82     }
83
84     private void SucheEintragNachNachnamen()
85     {
86         Console.Clear();
87         Console.WriteLine("Geben Sie einen Nachnamen ein.");
88         verzeichnis.ZeigeEintraegeFuerNachname(Console.
89         ReadLine());
90         Zurueck();
91     }
92
93     private void Zurueck()
94     {
95         Console.
96         WriteLine("-----");
97         Console.WriteLine("Drücken Sie die Enter-Taste, um
98             zum Menü zu gelangen.");
99         Console.ReadLine();
100    }
101
102    public void Start()
103    {
104        var ende = false;
105        while(!ende)
106        {
107            ZeigeMenu();
108
109            switch(LeseMenuPunkt())
110            {
111                case 1:
112                    ZeigeAlleEintraege();
```

## 9.8 Übungsaufgabe: Mit Objekten programmieren

```

111         break;
112     case 2:
113         ZeigeEintrag();
114         break;
115     case 3:
116         SucheEintragNachVornamen();
117         break;
118     case 4:
119         SucheEintragNachNachnamen();
120         break;
121     case 8:
122         ende = true;
123         break;
124     }
125   }
126 }
127 }
128 }
```



Abb. 9.8.3 Musterlösung Teilaufgabe 6 Auswahl Menüpunkt 1



Abb. 9.8.4 Musterlösung Teilaufgabe 6 Ausführung Menüpunkt 1



Abb. 9.8.5 Musterlösung Teilaufgabe 5 Auswahl Menüpunkt 2



Abb. 9.8.6 Musterlösung Teilaufgabe 6 Ausführung Menüpunkt 2

## 9 Grundlagen der Objektorientierten Programmierung



Abb. 9.8.7 Musterlösung Teilaufgabe 6 Auswahl Menüpunkt 3



Abb. 9.8.8 Musterlösung Teilaufgabe 6 Ausführung Menüpunkt 3



Abb. 9.8.9 Musterlösung Teilaufgabe 6 Auswahl Menüpunkt 4



Abb. 9.8.10 Musterlösung Teilaufgabe 6 Ausführung Menüpunkt 4

**Musterlösung für Teilaufgabe 7:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace TelefonVerzeichnisObjektorientiert
6  {
7      public class VerzeichnisApp
8      {
9          private Verzeichnis verzeichnis;
10
11         public VerzeichnisApp()
12         {
13             verzeichnis = new Verzeichnis();
14         }
15
16         private void ZeigeMenu()
17         {
18             Console.Clear();
19             Console.WriteLine("1. Alle Einträge anzeigen.");
20             Console.WriteLine("2. Einen bestimmten Eintrag
21                 anzeigen");
22             Console.WriteLine("3. Einträge suchen nach
23                 Vornamen.");
24             Console.WriteLine("4. Einträge suchen nach
25                 Nachnamen.");
26             Console.WriteLine("5. Eintrag hinzufügen.");
27             Console.WriteLine("6. Eintrag ändern.");
28             Console.WriteLine("7. Eintrag löschen");
29             Console.WriteLine("8. Programm beenden.");
30             Console.
31             WriteLine("-----");
32             Console.WriteLine("Wählen Sie einen Menüpunkt (1-
33                 8)");
34         }
35
36         private int LeseMenuPunkt()
37         {
38             int menupunkt;
39             do
40             {
41                 if (int.TryParse(Console.ReadLine(), out
42                     menupunkt))
43                 {
44                     if (menupunkt>=1 && menupunkt<=8)
45                     {
46                         break;
47                     }
48                 }
49                 Console.WriteLine("Wählen Sie einen Menüpunkt (1-
50                     8)");
51             }
52             while (1==1);
53
54             return menupunkt;
```

## 9 Grundlagen der Objektorientierten Programmierung

```
55
56
57     }
58
59     private void ZeigeAlleEintraege()
60     {
61         Console.Clear();
62         verzeichnis.ZeigeAlleEintraege();
63         Zurueck();
64     }
65
66     private void ZeigeEintrag()
67     {
68         Console.Clear();
69         Console.WriteLine("Geben Sie einen Schlüssel
70         (Vorname:Nachname) ein.");
71         verzeichnis.ZeigeEintrag(Console.ReadLine());
72         Zurueck();
73     }
74
75     private void SucheEintragNachVornamen()
76     {
77         Console.Clear();
78         Console.WriteLine("Geben Sie einen Vornamen ein.");
79         verzeichnis.ZeigeEintraegeFuerVorname(Console.
80         ReadLine());
81         Zurueck();
82     }
83
84     private void SucheEintragNachNachnamen()
85     {
86         Console.Clear();
87         Console.WriteLine("Geben Sie einen Nachnamen ein.");
88         verzeichnis.ZeigeEintraegeFuerNachname(Console.
89         ReadLine());
90         Zurueck();
91     }
92
93     private void FuegeEintragHinzu()
94     {
95         Console.Clear();
96         Console.WriteLine("Geben Sie einen Vornamen ein.");
97         var vorname = Console.ReadLine();
98         Console.WriteLine("Geben Sie einen Nachnamen ein.");
99         var nachname = Console.ReadLine();
100        Console.WriteLine("Geben Sie eine Festnetznummer
101        ein.");
102        var festnetz = Console.ReadLine();
103        Console.WriteLine("Geben Sie eine Mobilfunknummer
104        ein.");
105        var mobilfunk = Console.ReadLine();
106        Console.WriteLine("Geben Sie eine E-Mailadresse
107        ein.");
108        var email = Console.ReadLine();
109
110        verzeichnis.NeuerEintrag(vorname, nachname, festnetz,
mobilfunk, email);
```

## 9.8 Übungsaufgabe: Mit Objekten programmieren

```
111         Zurueck();
112     }
113
114     private void AendereEintrag()
115     {
116         Console.Clear();
117         Console.WriteLine("Geben Sie einen Schlüssel
118         (Vorname:Nachname) ein.");
119         var schluessel = Console.ReadLine();
120         Console.WriteLine("Geben Sie eine neue Festnetznummer
121         ein.");
122         var festnetz = Console.ReadLine();
123         Console.WriteLine("Geben Sie eine neue
124         Mobilfunknummer ein.");
125         var mobilfunk = Console.ReadLine();
126         Console.WriteLine("Geben Sie eine neue E-Mailadresse
127         ein.");
128         var email = Console.ReadLine();
129
130         verzeichnis.AktualissiereEintrag(schluessel,
131         festnetz, mobilfunk, email);
132
133         Zurueck();
134     }
135
136     private void LoescheEintrag()
137     {
138         Console.Clear();
139         Console.WriteLine("Geben Sie einen Schlüssel
140         (Vorname:Nachname) ein.");
141         var schluessel = Console.ReadLine();
142         verzeichnis.LoescheEintrag(schluessel);
143         Zurueck();
144     }
145
146     private void Zurueck()
147     {
148         Console.
149         WriteLine("-----");
150         Console.WriteLine("Drücken Sie die Enter-Taste, um
151         zum Menü zu gelangen.");
152         Console.ReadLine();
153     }
154
155     public void Start()
156     {
157         var ende = false;
158         while(!ende)
159         {
160             ZeigeMenu();
161
162             switch(LeseMenuPunkt())
163             {
164                 case 1:
165                     ZeigeAlleEintraege();
166                     break;
167                 case 2:
```

## 9 Grundlagen der Objektorientierten Programmierung

```

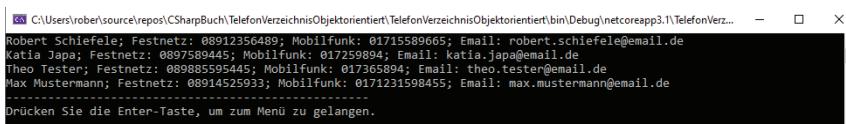
168             ZeigeEintrag();
169             break;
170         case 3:
171             SucheEintragNachVornamen();
172             break;
173         case 4:
174             SucheEintragNachNachnamen();
175             break;
176         case 5:
177             FuegeEintragHinzu();
178             break;
179         case 6:
180             AendereEintrag();
181             break;
182         case 7:
183             LoescheEintrag();
184             break;
185         case 8:
186             ende = true;
187             break;
188     }
189 }
190 }
191 }
192 }
```



**Abb. 9.8.11** Musterlösung Teilaufgabe 7 Auswahl Menüpunkt 5



**Abb. 9.8.12** Musterlösung Teilaufgabe 7 Ausführung Menüpunkt 5



**Abb. 9.8.13** Musterlösung Teilaufgabe 7 Anzeige aller Einträge nach Ausführung Menüpunkt 5

## 9.8 Übungsaufgabe: Mit Objekten programmieren

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
1. Alle Einträge anzeigen.
2. Ein einen bestimmten Eintrag anzeigen.
3. Einträge suchen nach Vornamen.
4. Einträge suchen nach Nachnamen.
5. Eintrag hinzufügen.
6. Eintrag ändern.
7. Eintrag löschen.
8. Programm beenden.

Wählen Sie einen Menüpunkt (1-8)
6
```

Abb. 9.8.14 Musterlösung Teilaufgabe 7 Auswahl Menüpunkt 6

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
Geben Sie einen Schlüssel (Vorname:Nachname) ein.
Max:Mustermann
Geben Sie eine neue Festnetznummer ein.
08911111111111
Geben Sie eine neue Mobilfunknummer ein.
01721111111111
Geben Sie eine neue E-Mailadresse ein.
m.mustermann@email.de
Drücken Sie die Enter-Taste, um zum Menü zu gelangen.
```

Abb. 9.8.15 Musterlösung Teilaufgabe 7 Ausführung Menüpunkt 6

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
Robert Schiefele; Festnetz: 08912356489; Mobilfunk: 01715589665; Email: robert.schiefele@email.de
Katia Japa; Festnetz: 0897589445; Mobilfunk: 017259894; Email: katia.japa@email.de
Theo Tester; Festnetz: 089885595445; Mobilfunk: 017365894; Email: theo.tester@email.de
Max Mustermann; Festnetz: 08911111111111; Mobilfunk: 01721111111111; Email: m.mustermann@email.de
Drücken Sie die Enter-Taste, um zum Menü zu gelangen.
```

Abb. 9.8.16 Musterlösung Teilaufgabe 7 Anzeige aller Einträge nach Ausführung Menüpunkt 6

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
1. Alle Einträge anzeigen.
2. Ein einen bestimmten Eintrag anzeigen.
3. Einträge suchen nach Vornamen.
4. Einträge suchen nach Nachnamen.
5. Eintrag hinzufügen.
6. Eintrag ändern.
7. Eintrag löschen.
8. Programm beenden.

Wählen Sie einen Menüpunkt (1-8)
7
```

Abb. 9.8.17 Musterlösung Teilaufgabe 7 Auswahl Menüpunkt 7

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
Geben Sie einen Schlüssel (Vorname:Nachname) ein.
Max:Mustermann
Drücken Sie die Enter-Taste, um zum Menü zu gelangen.
```

Abb. 9.8.18 Musterlösung Teilaufgabe 7 Ausführung Menüpunkt 7

```
C:\Users\rober\source\repos\CSharpBuch\TelefonVerzeichnisObjektorientiert\TelefonVerzeichnisObjektorientiert\bin\Debug\netcoreapp3.1\TelefonVerz...
Robert Schiefele; Festnetz: 08912356489; Mobilfunk: 01715589665; Email: robert.schiefele@email.de
Katia Japa; Festnetz: 0897589445; Mobilfunk: 017259894; Email: katia.japa@email.de
Theo Tester; Festnetz: 089885595445; Mobilfunk: 017365894; Email: theo.tester@email.de
Drücken Sie die Enter-Taste, um zum Menü zu gelangen.
```

Abb. 9.8.19 Musterlösung Teilaufgabe 7 Anzeige aller Einträge nach Ausführung Menüpunkt 7

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 10

## Objektorientierung für Fortgeschrittene

In den folgenden Kapiteln werden wir tiefer in die Objektorientierte Programmierung einsteigen und dabei abstrakte Konzepte wie Vererbung, Schnittstellen und Polymorphismus kennenlernen. Zudem betrachten wir Erweiterungsmethoden und die sogenannten „Generics“. Und danach behandeln wir das Konzept der Delegaten und Lambda-Ausdrücke, welches uns zur sogenannten „Funktionalen Programmierung“ führt.

Vielleicht haben einige von Ihnen schon etwas darüber gelesen und dabei auch viel Unsinn aufgeschnappt, der gerne über die Funktionale Programmierung geschrieben wird wie zum Beispiel, dass die Ära der Objektorientierten Programmierung am Ende wäre und jetzt die Zeit der Funktionalen Programmierung beginnen würde.

Die Funktionale Programmierung ist kein neuer Ansatz, der die Objektorientierung ablöst. Die Funktionale Programmierung basiert auf dem Lambda-Kalkül, das in den 1930ern von den Mathematikern Alonzo Church und Stephen Cole Kleene eingeführt wurde. Die funktionale Programmierung erweitert die Objektorientierung um weitere interessante Möglichkeit der Gestaltung von Computerprogrammen. Aber beginnen wir mit dem ersten Schritt in Richtung fortgeschrittene Objektorientierung: der Vererbung.

### 10.1 Vererbung: abgeleitete Klassen

Vererbung in der Objektorientierung bedeutet, dass eine Klasse die Eigenschaften und Methoden einer anderen Klasse erben kann. Betrachten wir dazu einmal die folgenden Klasse:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Vererbung
6  {
7      public class Adresse
8      {
9          public string Strasse { get; set; }
10         public string Ort { get; set; }
11         public string Plz { get; set; }
12
13         public void ZeigeAdresse()
14     }
```

## 10 Objektorientierung für Fortgeschrittene

```
15         Console.WriteLine(Strasse);
16         Console.WriteLine(${Plz} {Ort}");
17     }
18 }
19 }
```

Die Klasse `Adresse` hat public-Properties zum Speichern von Straße, Ort und PLZ und eine public-Methode, um die Adresse am Bildschirm auszugeben. In einer Anwendung könnten wir uns jetzt verschiedene Objekte vorstellen, die eine Adresse haben. Zum Beispiel eine Person, eine Firma oder ein Gebäude. Natürlich könnten diese Objekte alle eine Property vom Typ `Adresse` haben, um ihre Adresse zu speichern. Aber mit dem Hilfsmittel der Vererbung kann man das viel eleganter lösen. Mit Vererbung würde eine Klasse `Person` dann so aussehen:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Vererbung
6  {
7      public class Person : Adresse
8      {
9          public string Vorname { get; set; }
10         public string Nachname { get; set; }
11
12         public void ZeigePerson()
13         {
14             Console.WriteLine(${Vorname} {Nachname}");
15             ZeigeAdresse();
16         }
17     }
18 }
```

Nach dem Klassennamen `Person` schreiben wir einen Doppelpunkt, gefolgt vom Namen der Klasse, von der unsere Klasse `Person` erben soll. In diesem Fall `Adresse`. Die Klasse `Person` definiert die Properties `Vorname` und `Nachname`, um Vor- und Nachnamen einer Person zu speichern. Die public-Methode `ZeigePerson()` gibt die Werte der Properties `Vorname` und `Nachname` am Bildschirm aus. Zur Ausgabe der Adresse einer Person verwendet die Methode `ZeigePerson()` die von der Klasse `Adresse` geerbte Methode `ZeigeAdresse()`. Obwohl die Klasse `Person` keine Properties definiert, um eine Adresse zu speichern, kann sie dafür die von der Klasse `Adresse` geerbten Properties `Strasse`, `Plz` und `Ort` verwenden.

Sehen wir uns einmal ein kleines Testprogramm für die Klasse `Person` an:

```
1  using System;
2
3  namespace Vererbung
4  {
5      class Program
```

```

6   {
7     static void Main(string[] args)
8     {
9       var person = new Person();
10      person.Vorname = "Max";
11      person.Nachname = "Mustermann";
12      person.Strasse = "Musterstrasse 6";
13      person.Plz = "12345";
14      person.Ort = "Musterhausen";
15      person.ZeigePerson();
16    }
17  }
18 }
```

Wir erzeugen eine Instanz der Klasse Person und weisen den Properties Vorname, Nachname, Strasse, Plz und Ort Werte zu und rufen die Methode ZeigePerson().

Abb. 10.1.1 Ausgabe des Testprogramms für die Klasse Person

Wie wir sofort erkennen können, macht es bei der Verwendung der Instanz einer Klasse keinen Unterschied, ob eine Property oder Methode von der Klasse selbst definiert oder von einer Basisklasse geerbt wird. Man sagt auch, die Klasse Person ist von der Klasse Adresse abgeleitet. Wir können jetzt beliebig viele Klassen von der Klasse Adresse erben lassen. Und wenn wir zum Beispiel eine neue Anforderung an unsere Applikation bekommen, dass zum Beispiel bei der Adresse nicht nur der Ort, sondern auch das Land gespeichert werden soll, dann erweitern wir die Klasse Adresse um eine Property Land und alle Klassen, die von Adresse erben, können die neue Property verwenden.

10

Die Väter der objektorientierten Programmierung haben eine Mehrfachvererbung in beide Richtungen vorgesehen und das wurde auch zum Beispiel in die Sprache C++ eingebaut. In der Praxis hat man aber gesehen, dass das mehr Verwirrung als echten Nutzen bringt. Daher wurde bei C# die Mehrfachvererbung nur in eine Richtung erlaubt. Das heißt, es können beliebig viele Klassen von einer Basisklasse erben. Aber eine Klasse kann immer nur von einer Klasse erben. Als Merkhilfe können Sie sich an der Biologie orientieren. Ein Vater kann viele Kinder haben, aber ein Kind hat nur einen Vater.

Die Vererbung von Klassen kann auch über mehrere Stufen erfolgen, wie in der Biologie kann es auch „Großväter-Klassen“ geben. Betrachten wir dazu die Klasse Student:

## 10 Objektorientierung für Fortgeschrittene

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Vererbung
6  {
7      public class Student : Person
8      {
9          public string Universitaet { get; set; }
10
11         public void ZeigeStudent()
12         {
13             Console.WriteLine(Universitaet);
14             ZeigePerson();
15         }
16     }
17 }
```

Die Klasse `Student` hat nur eine Property: `Universitaet`. Hier speichern wir den Namen der Universität, die der Student besucht. Des Weiteren definiert die Klasse `Student` die Methode `ZeigeStudent()`, die den Wert der Property `Universitaet` ausgibt und für die weitere Ausgabe auf die, von der Klasse `Person` geerbte Methode `ZeigePerson()`, zurückgreift. Die Klasse `Student` erbt von der Klasse `Person` und da die Klasse `Person` von der Klasse `Adresse` erbt, erbt die Klasse `Student` auch indirekt von der Klasse `Adresse`. Die Klasse `Student` kann daher wie folgt verwendet werden:

```
1  using System;
2
3  namespace Vererbung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var student = new Student();
10             student.Universitaet = "Freie Universität Berlin";
11             student.Vorname = "Max";
12             student.Nachname = "Mustermann";
13             student.Strasse = "Musterstrasse 6";
14             student.Plz = "12345";
15             student.Ort = "Musterhausen";
16             student.ZeigeStudent();
17         }
18     }
19 }
```

Die Klasse `Student` hat Zugriff auf alle Properties der Klassen `Person` und `Adresse`.

## 10.1 Vererbung: abgeleitete Klassen

```
C:\Users\rober\source\repos\CSharpBuch\Vererbung\bin\Debug\netcoreapp3.1\Vererbung.exe (Prozess "6560") wurde
mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.1.2 Ausgabe des Testprogramms für die Klasse Student

Unsere drei Klassen `Adresse`, `Person` und `Student` haben wir alle ohne Konstruktor definiert. Das heißt, alle drei verwenden den parameterlosen Standard-Konstruktor. Was passiert, wenn die Klasse `Adresse` nachträglich einen Konstruktor mit Übergabeparametern erhält?

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Vererbung
6  {
7      public class Adresse
8      {
9          public Adresse(string strasse, string ort, string plz)
10         {
11             Strasse = strasse;
12             Ort = ort;
13             Plz = plz;
14         }
15
16         public string Strasse { get; set; }
17         public string Ort { get; set; }
18         public string Plz { get; set; }
19
20         public void ZeigeAdresse()
21         {
22             Console.WriteLine(Strasse);
23             Console.WriteLine($"{Plz} {Ort}");
24         }
25     }
26 }
```

10

Wenn wir jetzt kompilieren, erhalten wir für die Klasse `Person` den folgenden Fehler:

```
1  Es wurde kein Argument angegeben, das dem formalen
2  Parameter "strasse" von "Adresse.Adresse(string, string,
3  string)" entspricht.
```

Der Fehler wird in unserem Programmcode für die Klassendefinition von `Person` angegeben. Das ist etwas verwirrend. Auch die Fehlermeldung an sich ist für ungeübte

## 10 Objektorientierung für Fortgeschrittene

Programmierer wenig hilfreich. Damit ist Folgendes gemeint: Wenn eine Klasse von einer anderen Klasse erbt, dann muss sie auch deren Konstruktoren nachbilden, da C# im Gegensatz zu C++ keine Konstruktor-Vererbung unterstützt. Aber auch für dieses Problem gibt es eine Lösung:

```
1 public Person(string strasse, string ort, string plz):
2     base(strasse, ort, plz)
3 {
4 }
```

Die Klasse Person erhält einen Konstruktor, der wie der Konstruktor der Klasse Adresse drei String-Parameter enthält: strasse, ort und plz. Die Namen der Parameter sind beliebig, aber wenn wir die gleichen Namen vergeben wie im Konstruktor von Adresse, dann ist unser Programm lesbarer. Der Deklaration des Konstruktors folgt ein Doppelpunkt und das Schlüsselwort base und die drei Parameter des Konstruktors von Person in runden Klammern und durch Kommata getrennt. Wie der Name base schon vermuten lässt, rufen wir damit den Konstruktor der Basisklasse Adresse auf und übertragen ihm die drei an die Klasse Person übergebenen Parameter.

Allerdings ist das Problem noch nicht ganz behoben. Jetzt taucht der Fehler bei der Klasse Student auf. Aber dieses Problem können wir mit der gleichen Lösung beheben.

```
1 public Student(string strasse, string ort, string plz) :
2     base(strasse, ort, plz)
3 {
4 }
```

Leider sind wir noch nicht ganz fertig, da die Klasse Student jetzt keinen parameterlosen Konstruktor mehr hat, müssen wir im Hauptprogramm dem Konstruktor von Student noch die entsprechenden Parameter übergeben:

```
1 using System;
2
3 namespace Vererbung
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var student = new Student("Musterstrasse 6",
10                 "12345", "Musterhausen");
11             student.Universitaet = "Freie Universität Berlin";
12             student.Vorname = "Max";
13             student.Nachname = "Mustermann";
14
15             student.ZeigeStudent();
16         }
17 }
```

```
17     }
18 }
```

Natürlich können wir dann auf die Zuweisung der Properties `Strasse`, `Plz` und `Ort` verzichten. Und unser Programm funktioniert wieder.

```
Microsoft Visual Studio-Debugging-Konsole
Freie Universität Berlin
Max Mustermann
Musterstrasse 6
Musterhausen 12345
C:\Users\rober\source\repos\CSharpBuch\Vererbung\Vererbung\bin\Debug\netcoreapp3.1\Vererbung.exe (Prozess "11948") wurde
mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.1.3 Ausgabe des Testprogramms für die Klasse Student mit Konstruktor

Wenn eine Klasse, die von einer anderen Klasse erbt, deren Konstruktor nachbildet, kann sie dabei seine Funktionalität auch zusätzlich noch erweitern:

```
1 public Person(string vorname, string nachname, string strasse,
2   string ort, string plz):base(strasse, ort, plz)
3 {
4     Vorname = vorname;
5     Nachname = nachname;
6 }
```

Jetzt übergeben wir dem Konstruktor der Klasse `Person` zusätzlich noch die Parameter `vorname` und `nachname`, mit denen wir die Properties `Vorname` und `Nachname` setzen. Die Parameter `strasse`, `ort` und `plz` reichen wir an den Konstruktor der Basisklasse `Adresse` weiter. Diese Erweiterung müssen wir jetzt natürlich auch in der Klasse `Student` nachziehen, da der Konstruktor von `Person` der Basisklasse von `Student` jetzt fünf statt drei Parameter verlangt.

```
1 public Student(string universitaet, string vorname,
2   string nachname, string strasse, string ort, string plz) :
3 base(vorname, nachname, strasse, ort, plz)
4 {
5     Universitaet = universitaet;
6 }
```

Zusätzlich zu den beiden Parametern `vorname` und `nachname`, die wir an den Konstruktor der Basisklasse `Person` weiterreichen, führen wir im Konstruktor von `Student` noch den Parameter `universitaet` ein, mit dem wir die Property `Universitaet` setzen.

Damit unser Programm wieder funktioniert, müssen wir noch das Hauptprogramm an den geänderten Konstruktor von `Student` anpassen:

```
1 using System;
```

## 10 Objektorientierung für Fortgeschrittene

```

2
3 namespace Vererbung
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var student = new Student("Freie Universität
10                Berlin", "Max", "Mustermann", "Musterstrasse 6",
11                "12345", "Musterhausen");
12
13             student.ZeigeStudent();
14         }
15     }
16 }
```

Das Setzen der Property Universitaet kann wieder entfallen, da dies jetzt vom Konstruktor der Klasse Student erledigt wird.

```

Microsoft Visual Studio-Debugging-Konsole
Freie Universität Berlin
Max Mustermann
Musterstrasse 6
Musterhausen 12345

C:\Users\rober\source\repos\CSharpBuch\Vererbung\Vererbung\bin\Debug\netcoreapp3.1\Vererbung.exe (Prozess "9996") wurde
mit Code "0" beendet.

Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 10.1.4** Ausgabe des Testprogramms für die Klasse Student mit erweitertem Konstruktor

### 10.2 Das Interface: Eine definierte Schnittstelle

In diesem Kapitel werden wir die sogenannten Interfaces einführen. Ein Interface wird ähnlich wie eine Klasse definiert. Statt dem Schlüsselwort `class` schreiben wir das Schlüsselwort `interface`. Zudem enthält ein Interface keine lokalen Variablen und auch keine Membervariablen. Das Interface selbst kann einen Zugriffsmodifizierer haben, der gilt dann aber für alle Methoden und Properties des Interfaces. Für Methoden und Properties darf kein Zugriffsmodifizierer vergeben werden, es gilt ja der Zugriffsmodifizierer des Interfaces.

Zudem enthält ein Interface keinen ausführbaren Programmcode, sondern nur die sogenannten Signaturen von Methoden und Properties. Die Signatur einer Methode ist nur der Kopf der Methode ohne Zugriffsmodifizierer und ohne Programmcode und auch ohne geschweifte Klammern. Eine Methodensignatur wird mit einem Semikolon abgeschlossen. Die Signatur einer Property ist die Deklaration der Property ohne Zugriffsmodifizier und ohne Programmcode im Getter oder Setter. Eine Property-Signatur kann auch nur einen Getter beinhalten, dann spricht man von einer `readonly`-Property.

Klassen können von Interfaces erben. Hierbei kann eine Klasse auch von mehreren Interfaces erben. Da ein Interface aber nur aus Signaturen ohne ausführbaren Programmcode besteht, kann eine Klasse keine vorprogrammierten Funktionalitäten von Interfaces erben. Vielmehr ist ein Interface als eine Art Vorschrift zu verstehen, die einer Klasse vorschreibt, welche Properties und Methoden die Klasse implementieren muss, um dem Interface zu genügen. Den ausführbaren Programmcode für im Interface definierte Methoden und Properties muss die Klasse selbst liefern. Ein Interface kann auch als Typ für eine Variable verwendet werden. Einer so deklarierten Variablen können dann Objekte mit verschiedenen Typen zugewiesen werden, sofern diese Typen das Interface implementieren, das als Typ für die Variable verwendet wird.

Doch jetzt zur Praxis:

```
1  using System.Text;
2
3  namespace Interfaces
4  {
5      public interface IFahrbar
6      {
7
8          string Typ { get; }
9
10         int AnzahlRaeder { get; }
11
12         void Fahren();
13     }
14 }
```

10

Unser Interface heißt `IFahrbar`. Der Name beginnt mit einem großen `I`, gefolgt von einem weiteren Großbuchstaben. Der Name ist grammatisch ein Adjektiv, das eine Möglichkeit ausdrückt. Hätten wir die Projektvorgabe, alle technischen Namen englisch zu benennen, würde unser Interface `IDrivable` heißen. Diese Benennungsvorschrift ist aber nicht verpflichtend, sondern nur eine Konvention unter Programmierern, um die Lesbarkeit von Programmen zu verbessern. Allerdings empfehle ich Ihnen, sich strikt an diese Konvention zu halten, sonstouten Sie sich sehr schnell als Anfänger.

Dieses Interface soll für alle Objekte gelten, die fahrbare sind, wie zum Beispiel Autos, Motorräder, Fahrräder etc. Alle fahrbaren Objekte haben einen `Typ`, wie zum Beispiel „E-Bike“. Sie haben alle Räder, nur unterschiedlich viele. Und Sie müssen natürlich auch fahren können, das wollen wir mit der Methode `Fahren` ausdrücken.

Eine Klasse kann ein Interface verwenden, man sagt die Klasse implementiert das Interface. Damit eine Klasse ein Interface implementiert, schreibt man genauso wie bei der Vererbung einen Doppelpunkt nach dem Klassennamen und dann den Interface-Namen. Daher ist die Namenskonvention für Interfaces so wichtig. Durch sie erkennt man

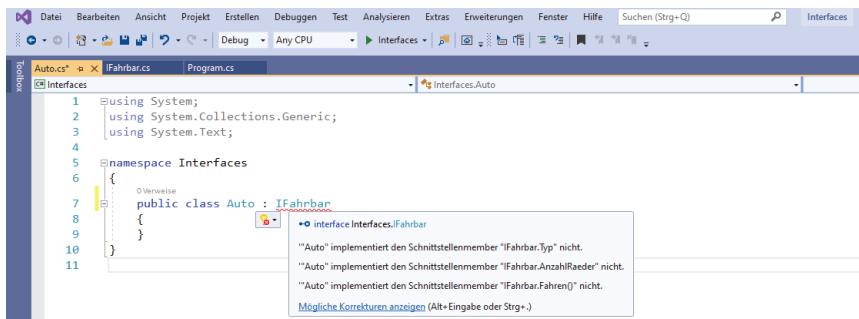
## 10 Objektorientierung für Fortgeschrittene

sofort, ob es sich um Vererbung von einer Klasse oder um ein Interface handelt. Jetzt wollen wir das Interface `IFahrbar` mit der Klasse `Auto` testen.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Interfaces
6  {
7      public class Auto : IFahrbar
8      {
9      }
10 }
```

Wenn wir einfach eine Klasse definieren und dann mit einem Doppelpunkt ein Interface an den Klassennamen hängen, zeigt uns Visual Studio sofort einen Fehler an. Das liegt daran, dass wir das Interface nicht implementiert haben. Das heißt, in unserer Klasse müssen wir genau die Methoden und Properties schreiben, für die im Interface eine Signatur definiert ist. Wir können das händisch erledigen oder uns von Visual Studio dabei helfen lassen. Bewegen Sie die Maus auf den Interface-Namen nach dem Namen der Klasse, dann sehen Sie folgenden Tooltip:



**Abb. 10.2.1** Editorunterstützung zum Implementieren eines Interfaces

Klicken Sie auf „Mögliche Korrekturen anzeigen“:

## 10.2 Das Interface: Eine definierte Schnittstelle

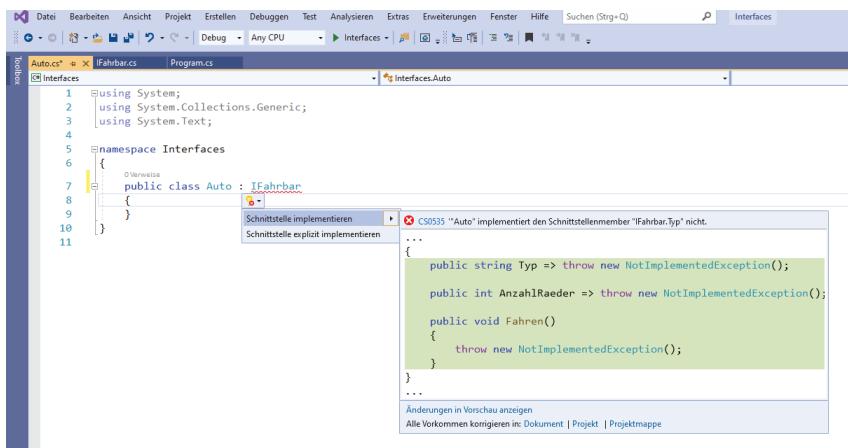


Abb. 10.2.2 Implementieren eines Interfaces mit Editorunterstützung

Und dann klicken Sie auf „Schnittstelle implementieren“:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Interfaces
6  {
7      public class Auto : IFahrbar
8      {
9          public string Typ => throw new NotImplementedException();
10
11         public int AnzahlRaeder => throw new
12             NotImplementedException();
13
14         public void Fahren()
15         {
16             throw new NotImplementedException();
17         }
18     }
19 }

```

10

Visual Studio erstellt Ihnen dann alle Methoden und Properties, die vom Interface verlangt werden. Alle Methoden und Properties werden von Visual Studio so geschrieben, dass sie eine Exception werfen. Zudem verwendet Visual Studio bei den Properties den sogenannten Lambda-Operator. Exceptions und den Lambda-Operator werden wir in diesem Buch an einer späteren Stelle genauer betrachten. Jetzt ändern wir die Methoden und Properties des Interfaces erst mal so ab, dass sie unsere Zwecke erfüllen.

## 10 Objektorientierung für Fortgeschrittene

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Interfaces
6  {
7      public class Auto : IFahrbar
8      {
9          public Auto()
10         {
11             Typ = "Auto";
12             AnzahlRaeder = 4;
13         }
14
15         public string Typ { get; }
16
17         public int AnzahlRaeder { get; }
18
19         public void Fahren()
20         {
21             Console.WriteLine($"Ich bin ein {Typ} und fahre auf
22             {AnzahlRaeder} Rädern.");
23         }
24     }
25 }
```

Die Properties `Typ` und `AnzahlRaeder` bekommen nur einen Getter, da wir schreibenden Zugriff von außen nicht erlauben wollen. Im Konstruktor von `Auto` setzen wir die beiden Properties auf die Werte „Auto“ und 4. In der Methode `Fahren` geben wir einen kurzen Text aus, der uns den Typ und die Anzahl der Räder mitteilt. Im Hauptprogramm schreiben wir einen einfachen Test für die Klasse `Auto`.

```

1  using System;
2
3  namespace Interfaces
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              new Auto().Fahren();
10         }
11     }
12 }
```



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output text is:

```

Microsoft Visual Studio-Debugging-Konsole
Ich bin ein Auto und fahre auf 4 Rädern.
C:\Users\rober\source\repos\CSharpBuch\Interfaces\Interfaces\bin\Debug\netcoreapp3.1\Interfaces.exe (Prozess "25688") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.2.3** Ausgabe des Testprogramms für die Klasse `Auto`

Die Klasse Auto arbeitet nicht anders als zu erwarten war. Wozu haben wir dann das Interface geschrieben? Als organisatorischer Merkzettel, damit wir nicht vergessen die nötigen Methoden und Properties zu implementieren? Dafür wäre der Aufwand dann doch etwas zu hoch. Um den Sinn und Zweck von Interfaces zu verstehen, benötigen wir eine weitere Klasse, die das Interface `IFahrbar` implementiert.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Interfaces
6  {
7      public class Motorrad : IFahrbar
8      {
9          public Motorrad()
10         {
11             Typ = "Motorrad";
12             AnzahlRaeder = 2;
13         }
14
15         public string Typ { get; }
16
17         public int AnzahlRaeder { get; }
18
19         public void Fahren()
20         {
21             Console.WriteLine($"Ich bin ein {Typ} und fahre auf
22             {AnzahlRaeder} Rädern.");
23         }
24     }
25 }
```

10

Die Klasse `Motorrad` funktioniert so ähnlich wie die Klasse `Auto`, mit dem einzigen Unterschied, dass sie im Konstruktor die Property `Typ` auf „Motorrad“ und die Property `AnzahlRaeder` auf 2 setzt.

Wir können jetzt das Interface `IFahrbar` auch wie einen Variablenotyp verwenden. Und Methoden, die Objekte vom Typ `IFahrbar` verarbeiten, können sowohl Objekte vom Typ `Auto` als auch Objekte vom Typ `Motorrad` verarbeiten. Als Anwendungsbeispiel betrachten wir die Fabrikklasse `Fahrzeugfabrik`, die immer nur den Typ `IFahrbar` erzeugt, dabei aber sowohl `Auto`-Objekte als auch `Motorrad`-Objekte erzeugen kann.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Interfaces
6  {
7      public enum FahrzeugTyp
8      {

```

## 10 Objektorientierung für Fortgeschrittene

```

9         Auto,
10        Motorrad
11    }
12
13    public class Fahrzeugfabrik
14    {
15        private FahrzeugTyp fahrzeugTyp;
16
17        public Fahrzeugfabrik(FahrzeugTyp fahrzeugTyp)
18        {
19            this.fahrzeugTyp = fahrzeugTyp;
20        }
21
22        public IFahrbar ErzeugeFahrzeug()
23        {
24            if(fahrzeugTyp == FahrzeugTyp.Motorrad)
25            {
26                return new Motorrad();
27            }
28
29            return new Auto();
30        }
31    }
32 }
```

Wir definieren für unsere Fahrzeugfabrik die Hilfsaufzählung `Fahrzeugtyp`, die die unterstützten Fahrzeugtypen definiert. Dem Konstruktor der Fabrik übergeben wir den Typ des Fahrzeugs, das wir erzeugen wollen, und die Methode `ErzeugeFahrzeug()` erzeugt entweder ein Motorrad-Objekt oder ein Auto-Objekt, gibt aber immer ein Objekt vom Typ `IFahrbar` zurück.

Im Hauptprogramm sehen wir, wie wir `IFahrbar`-Objekte, die entweder Motorrad-Objekte oder Auto-Objekte sind, verarbeiten können.

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Interfaces
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             IFahrbar motorrad = new Fahrzeugfabrik(FahrzeugTyp.
11             Motorrad).ErzeugeFahrzeug();
12             IFahrbar auto = new Fahrzeugfabrik(FahrzeugTyp.
13             Auto).
14             ErzeugeFahrzeug();
15
16             var fahrzeuge = new List<IFahrbar>();
17             fahrzeuge.Add(motorrad);
18             fahrzeuge.Add(auto);
19 }
```

### 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung

```

20         foreach(var fahrzeug in fahrzeuge)
21         {
22             fahrzeug.Fahren();
23         }
24     }
25 }
26 }
```

Über die Fahrzeugfabrik erzeugen wir ein neues Motorrad-Objekt. Obwohl die Fabrikmethode ein Objekt vom Typ Motorrad erzeugt, erhalten wir das Motorrad-Objekt mit dem Typ `IFahrbar`. Genauso verfahren wir mit einem Auto-Objekt. Da unsere beiden Objekte `motorrad` und `auto` jetzt formal den gleichen Typ haben, können wir sie der Liste `fahrzeuge` zuweisen. Die Liste `fahrzeuge` benötigt hierfür den Typ `List<IFahrbar>`.

Am Ende des Hauptprogramms laufen wir mit einer `foreach`-Schleife über die Liste und rufen für jedes Element die Methode `Fahren` auf. Dabei wird für das Motorrad-Objekt die Methode `Fahren` der Klasse `Motorrad` und für das Auto-Objekt die Methode `Fahren` der Klasse `Auto` aufgerufen.

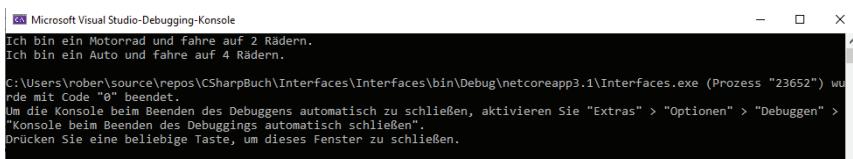


Abb. 10.2.4 Ausgabe des Testprogramms für die Klasse `IFahrbar`

### 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung

Nachdem wir die Interfaces von C# kennen gelernt haben, ist es an der Zeit, die Speicherverwaltung von C# näher zu beleuchten. Wenn ein Programm auf einem Computer gestartet wird, belegt es sofort Platz im Hauptspeicher des Computers. Platz, um den Programmcode unterzubringen, und Platz, um die statischen Membervariablen unterzubringen. Zusätzlich verwendet ein Programm auch den dynamischen Speicher, der zur Laufzeit des Programms reserviert wird.

Jedes Mal, wenn Sie einem String einen Wert zuweisen, jedes Mal, wenn Sie mit dem Schlüsselwort `new` eine neue Instanz einer Klasse erzeugen oder ein Array initialisieren und jedes Mal, wenn Sie einem Listentyp, wie `List`, `Dictionary` etc., ein neues Element hinzufügen, wird der Hauptspeicher reserviert. Und wenn ein Programm zur Laufzeit mehr Speicher reservieren möchte als das Betriebssystem zur Verfügung stellen kann, dann stürzt das Programm ab und alle Daten, die der Benutzer eingegeben hat, die noch nicht gespeichert wurden, gehen verloren. Wenn das bei einer Desk-

## 10 Objektorientierung für Fortgeschrittene

top-Applikation, wie zum Beispiel MS-Word passiert, ist der Text, der seit dem letzten Speichern getippt wurde, verloren.

Wenn das aber bei einer Server-Applikation, wie zum Beispiel MS-Exchange passiert, sind alle Daten aller Nutzer, die die Server-Applikation in einem Cache hält und noch nicht gespeichert hat, verloren. Zudem können die Benutzer eine Server-Applikation nicht einfach neu starten, das muss ein Administrator erledigen.

Außerdem kann es bei Server-Applikationen passieren, dass bei einem unerwarteten Absturz die sogenannte Datenintegrität verloren geht. Das heißt, die Daten, die die Server-Applikation verwenden, liegen nicht mehr in dem von der Applikation verwendeten Format vor, weil die Datensätze durch den Absturz nur unvollständig geschrieben wurden. Dann kann die Anwendung erst mal nicht verwendet werden, bis sie von einem Experten repariert wurde. Im Fall von MS-Exchange würde das heißen, sämtliche Kollegen können E-Mails weder versenden noch empfangen.

Sie sehen, in der professionellen Software-Entwicklung ist die Speicherverwaltung ein sehr wichtiges Thema.

In der Sprache C++ wird einmal reservierter Speicher nicht automatisch freigegeben. Der Programmierer muss sich darum kümmern, dass nicht mehr benötigter Speicher wieder freigegeben wird. In C# gibt es dagegen den sogenannten Garbage Collector (Englisch für Müllsammler). Das ist ein niedrig priorisierte Prozess, der reservierten Speicher findet und, wenn er nicht mehr benötigt wird, auch wieder frei gibt. Das klingt erst mal so, als müsste uns als C#-Programmierer die Speicherverwaltung nicht interessieren. Leider ist das nicht so. Der Garbage Collector ist zwar eine große Hilfe, aber ein Rundum-Sorglos-Paket ist er nicht. Daher muss ein C#-Programmierer dem Garbage Collector manchmal etwas helfen.

Um zu verstehen, was der Garbage Collector tut, rufen wir uns die am Anfang dieses Buches besprochene Architektur des .NET-Frameworks ins Gedächtnis. C# wird in den CIL-Code kompiliert und der kompilierte CIL-Code lebt in seiner eigenen Umgebung, der CLR. Der CIL-Code wird auch als verwalteter Code bezeichnet. Wenn wir es nur mit verwaltetem Code innerhalb einer CLR zu tun hätten, dann wären unsere Programme „blind“, „taub“ und „stumm“. Das heißt, wir könnten keine Tastaturabfragen machen, keine Maussteuerung in unsere Programme einbauen, keine Bildschirmausgaben machen und nichts auf der Festplatte speichern. Die Tastatur, die Maus, der Bildschirm, die Festplatte etc. sind alles vom Betriebssystem verwaltete Ressourcen, auf die wir nur mit einem unverwalteten Code zugreifen können. Unverwaltet heißt in diesem Fall: nicht von der CLR verwaltet.

Als C# Programmierer merken wir normalerweise nichts davon, denn der .NET-Framework liefert uns Klassen und Methoden, wie zum Beispiel `Console.WriteLine()`

### 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung

und `Console.ReadLine()`, mit denen wir aus der CLR heraus auf unverwaltete Ressourcen zugreifen können. Der Garbage Collector lebt in der CLR und kann daher nur nicht mehr benötigten Speicher von verwaltetem Code freigeben. Daraus ergeben sich zwei Punkte, bei denen wir als Programmierer aufpassen müssen.

1. Der Garbage Collector gibt nur Speicher frei, der nicht mehr benötigt wird, das heißt Objekte, für die keine Referenz mehr existiert. Wenn wir die Referenzen auf unsere Objekte lange halten, zum Beispiel in statischen Variablen, dann kann der Garbage Collector diesen Speicher nicht freigeben.
2. Wenn wir unverwalteten Code verwenden oder Objekte des .NET-Frameworks verwenden, die ihrerseits einen unverwalteten Code verwenden, gibt es Situationen, in denen wir uns als Programmierer um die Speicherverwaltung kümmern müssen. Das Thema, unverwalteten Code in eigenen Klassen zu verwenden, würde den Rahmen dieses Buches sprengen. Allerdings werden wir Klassen des .NET-Frameworks zum Beispiel beim Dateizugriff verwenden, die mit unverwalteten Ressourcen sehr viel Speicher belegen können, so dass wir explizit dafür sorgen müssen, dass dieser Speicher wieder freigegeben wird.

Damit wollen wir die Theorie beenden und in der Praxis ein paar Experimente mit dem Garbage Collector durchführen.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GarbageCollector
6  {
7      public class GrosseKlasse
8      {
9      }
10 }
```

10

Die Klasse `GrosseKlasse` ist eine Klasse ohne Methoden, Membervariablen und Properties. Sie tut einfach gar nichts.

```
1  using System;
2
3  namespace GarbageCollector
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var grosseKlasse = new GrosseKlasse();
10             Console.ReadLine();
11         }
12     }
13 }
```

## 10 Objektorientierung für Fortgeschrittene

Das Hauptprogramm erzeugt eine Instanz dieser Klasse und wartet auf eine Tastatureingabe, nach der das Programm endet.

Wir starten das Programm und sehen in den Visual Studio Diagnosetools den Hauptspeicherverbrauch unseres Programms.

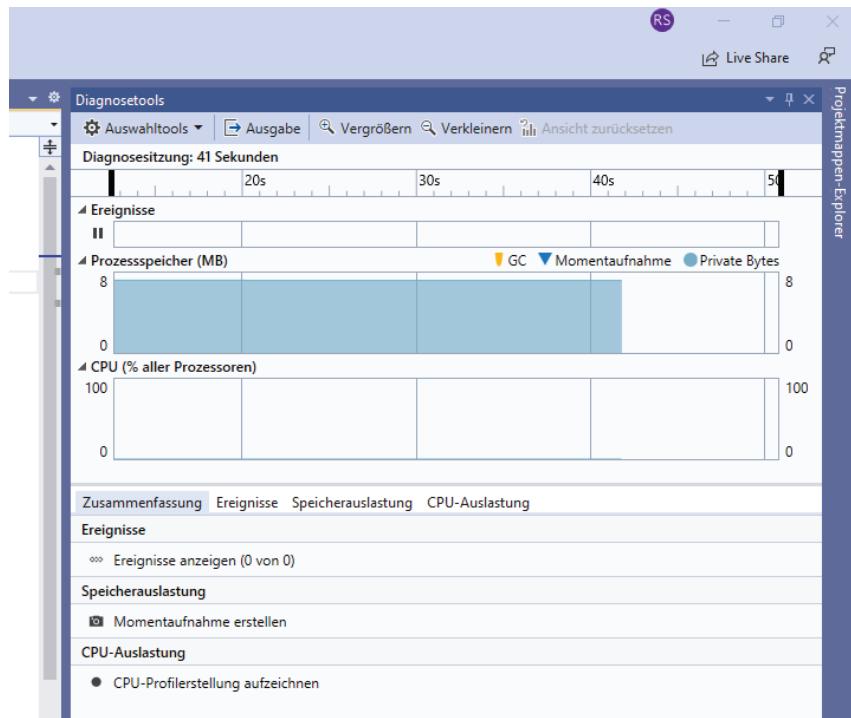
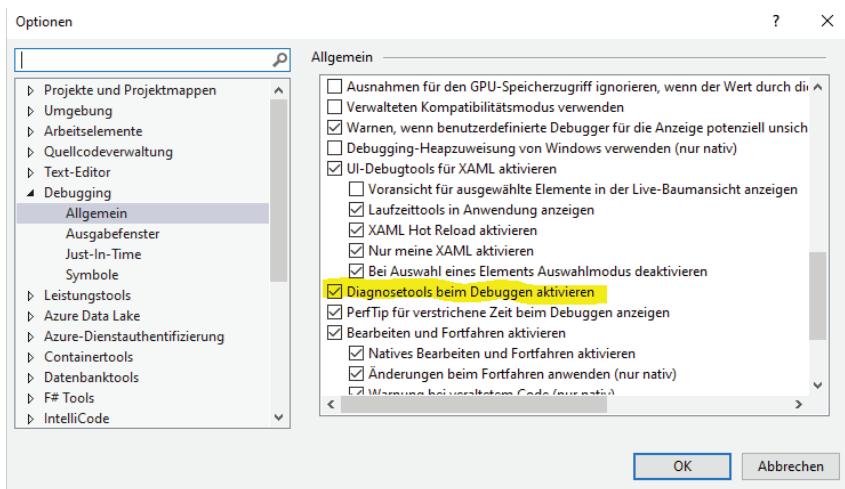


Abb. 10.3.1 Basis speicherbedarf des Experimentierprogramms

Unser Experimentierprogramm verbraucht 8 MB Hauptspeicher. Falls Ihr Visual Studio keine Diagnosetools anzeigt: Öffnen Sie das Optionenfenster, indem Sie im Menü auf „Extras/Optionen...“ klicken.

## 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung



**Abb. 10.3.2** Aktivierung der Diagnosetools

Wählen Sie die Optionen für „Debugging/Allgemein“ aus und aktivieren Sie die Checkbox „Diagnosetools beim Debuggen aktivieren“.

Jetzt wollen wir den Speicherverbrauch unseres Programms in verschiedenen Situationen untersuchen.

10

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GarbageCollector
6  {
7      public class GrosseKlasse
8      {
9          public GrosseKlasse()
10         {
11             var zeile = "Das ist eine Text Zeile.";
12             var vieleZeilen = new List<string>();
13
14             for (var i = 0; i < 1000000; i++)
15             {
16                 vieleZeilen.Add(zeile);
17             }
18
19         }
20     }
21 }
```

## 10 Objektorientierung für Fortgeschrittene

Unsere Experimentierklasse hat jetzt einen Konstruktor, der eine Million Strings im Hauptspeicher anlegt. Das sollte den Speicherverbrauch etwas in die Höhe treiben.

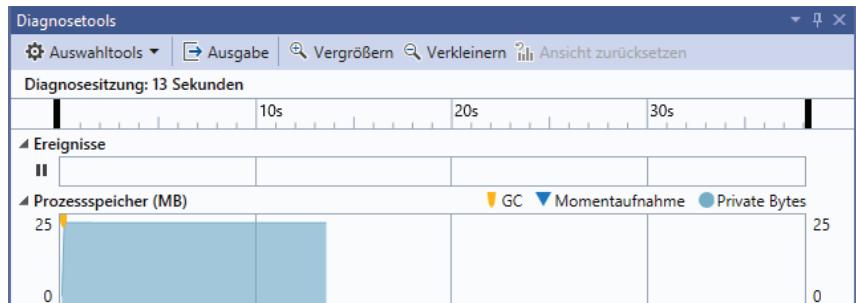


Abb. 10.3.3 Speicherverlauf des Experimentierprogramms

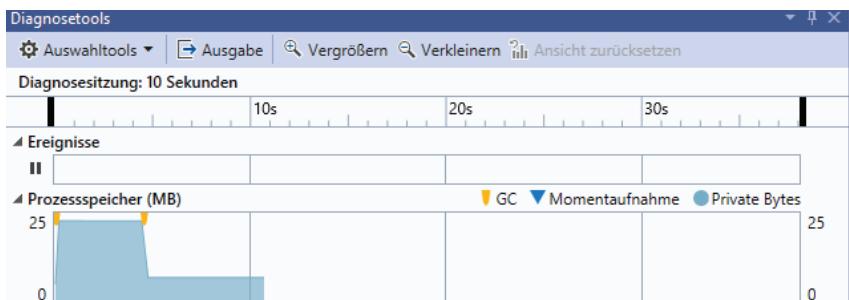
Jetzt verbraucht unser Programm 25 MB Hauptspeicher. Aber sollte dieser Speicher nicht sofort wieder freigegeben werden? Wir belegen diesen Speicher nur mit einer lokalen Variablen und nachdem der Konstruktor der Klasse `GrosseKlasse` fertig ist, gibt es keine Referenz mehr auf diesen Speicher. Auf der linken Seite der Speichergrafik sehen wir einen kleinen gelben Pfeil, der nach unten zeigt. Dieser Pfeil zeigt uns an, dass beim Start unseres Programms der Garbage Collector einmal gelaufen ist und danach nicht mehr, daher wurde der Speicher auch nicht aufgeräumt. Der Garbage Collector wurde nicht aktiv, weil wir einfach noch genügend Hauptspeicher übrig hatten. Als nächstes versuchen wir das Aufräumen durch den Garbage Collector zu erzwingen, dazu ändern wir das Hauptprogramm etwas ab.

```

1  using System;
2
3  namespace GarbageCollector
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var grosseKlasse = new GrosseKlasse();
10             Console.ReadLine();
11             GC.Collect();
12             Console.ReadLine();
13         }
14     }
15 }
```

Nach dem Instanzieren von `GrosseKlasse` wartet das Hauptprogramm auf eine Tastatureingabe, dann ruft es die statische Methode `Collect()` der Klasse `GC` auf, dadurch beginnt der Garbage Collector mit der Bereinigung des Speichers. Danach wartet unser Programm wieder auf eine Tastatureingabe bevor es sich beendet.

## 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung



**Abb. 10.3.4** Speicherverlauf des Experimentierprogramms mit explizitem Aufruf des Garbage Collectors

Der Speicherverlauf beginnt wieder mit einem Bereinigungslauf des Garbage Collectors, dann steigt der Speicherbedarf auf 25 MB an. Nach einer Weile (wenn wir die Enter-Taste gedrückt haben), sehen wir wieder einen gelben Pfeil, der nach unten zeigt. Das heißt, der Garbage Collector führt den vom Programm angestoßenen Bereinigungslauf durch und gibt den Speicher, den wir im Konstruktor der Klasse `GrosseKlasse` belegt haben, wieder frei. Der Speicherverbrauch beträgt wieder 8 MB, das ist genauso viel, wie die „leere“ Klasse benötigen würde.

Im Folgenden werden wir den Verlauf der Speicherauslastung in verschiedenen Situationen betrachten, um daraus verschiedene Aspekte abzuleiten, die wir bei der Entwicklung speicherhungriger Applikationen beachten müssen.

10

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GarbageCollector
6  {
7      public class GrosseKlasse
8      {
9          string zeile = "Das ist eine Text Zeile.";
10
11         public List<string> VieleZeilen { get; set; } = new
12             List<string>();
13
14         public GrosseKlasse()
15         {
16             for (var i = 0; i < 1000000; i++)
17             {
18                 VieleZeilen.Add(zeile);
19             }
20         }
21     }
22 }
```

## 10 Objektorientierung für Fortgeschrittene

Wir erzeugen wieder wie vorher eine Million Strings im Konstruktor, aber diesmal speichern wir sie nicht in einer lokalen Variablen, sondern in einer public-Property. Am Hauptprogramm ändern wir nichts und sehen uns wieder den Speicherverlauf an.

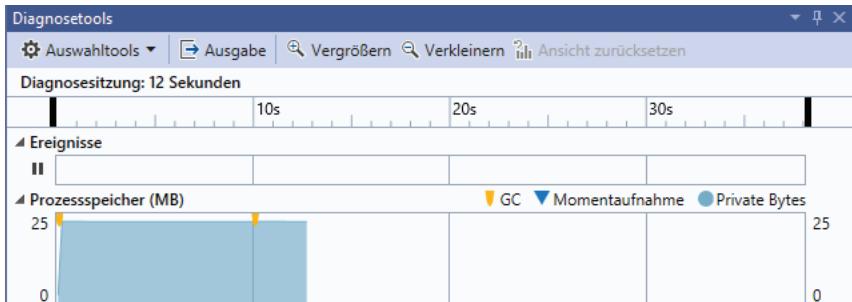


Abb. 10.3.5 Speicherverlauf des Experimentierprogramms mit gehaltener Referenz

Wir sehen zwar am gelben Pfeil, dass der Garbage Collector gelaufen ist, der Speicher wurde aber nicht freigegeben. Mit der Variablen `grosseKlasse` besteht eine Referenz auf die Klasse `GrosseKlasse` und mit der Property `VieleZeilen` besteht eine Referenz auf die Liste mit einer Million Strings, daher „glaubt“ der Garbage Collector, dass der Speicher noch gebraucht wird und gibt ihn nicht frei. Damit der Garbage Collector den Speicher freigeben kann, müssen wir die Referenz auf den Speicher aufheben. Somit ist es an der Zeit, das Schlüsselwort `null` einzuführen. Null bedeutet nichts und wenn wir einer Variablen `null` zuweisen, ist damit die Referenz auf den belegten Speicher wieder aufgehoben. Ändern wir unser Hauptprogramm, so dass wir der Variablen `grosseKlasse` erst eine Instanz der Klasse `GrosseKlasse` und dann nach `null` zuweisen.

```

1  using System;
2
3  namespace GarbageCollector
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var grosseKlasse = new GrosseKlasse();
10             grosseKlasse = null;
11             Console.ReadLine();
12             GC.Collect();
13             Console.ReadLine();
14         }
15     }
16 }
```

Und betrachten wir dann den Speicherverlauf.

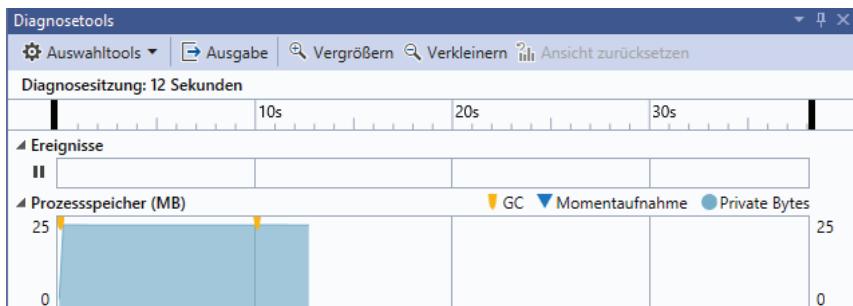


Abb. 10.3.6 Speicherverlauf des Experimentierprogramms mit freigegebener Referenz

Das Aufheben der Referenz auf die Klasse `GrosseKlasse` hatte scheinbar nicht die gewünschte Wirkung. Die Methode `GC.Collect()` führt nicht zwangsläufig eine Freigabe von allen Speicherbereichen durch, die nicht mehr gebraucht werden. Der Aufruf dieser Methode versteht sich lediglich als Bitte an den Garbage Collector, Speicher frei zugeben. In unserem Beispiel haben wir 25 MB Speicher belegt, als der Garbage Collector einen Bereinigungsversuch startet, da der Computer, mit dem dieser Speicherverlauf aufgezeichnet wurde, aber mit 16 GB Speicher ausgestattet ist, erkennt der Garbage Collector, dass noch freier Speicher im Überfluss vorhanden ist und gibt nur direkte Referenzen frei. Unsere Liste mit einer Million Strings hat noch eine Referenz, nämlich die Property `VieleZeilen` der Klasse `GrosseKlasse`. Da es aber genügend freien Speicher gibt, analysiert der Garbage Collector nicht weiter und gibt auch keinen Speicher frei. Sehen wir mal, was passiert, wenn wir die direkte Referenz auf unsere Liste mit einer Million Strings aufheben.

10

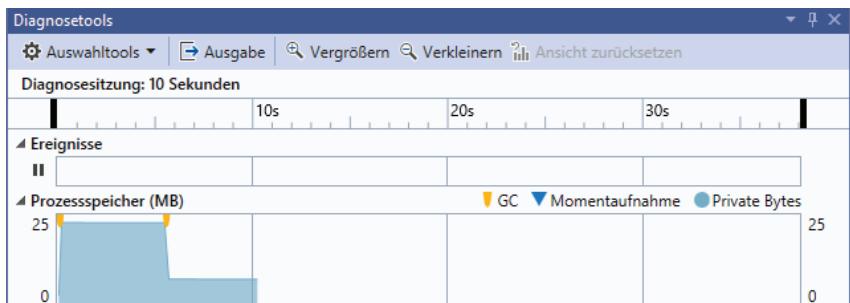
```

1  using System;
2
3  namespace GarbageCollector
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var grosseKlasse = new GrosseKlasse();
10             grosseKlasse.VieleZeilen = null;
11             Console.ReadLine();
12             GC.Collect();
13             Console.ReadLine();
14         }
15     }
16 }
```

Die Zeile `grosseKlasse.VieleZeilen = null;` hebt die Referenz auf die Liste auf.

Der Speicherverlauf sieht dann wie folgt aus:

## 10 Objektorientierung für Fortgeschrittene



**Abb. 10.3.7** Speicherverlauf des Experimentierprogramms mit freigegebener direkten Referenz

Wenn die direkte Referenz auf den Speicher aufgehoben ist, gibt der Garbage Collector den Speicher wieder frei. Das gleiche Verhalten würden wir auch beobachten, wenn die Property `private` wäre oder wenn wir statt einer Property eine Membervariable verwenden würden.

In unserem nächsten Versuch wollen wir untersuchen, was passiert, wenn nur die indirekte Referenz auf unseren Speicher aufgehoben wird und das Angebot an noch nicht belegtem Speicher geringer ist.

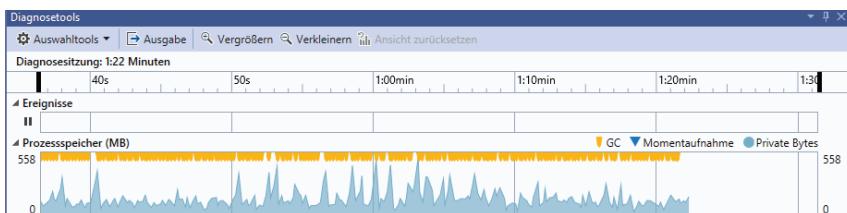
```

1  using System;
2  using System.Collections.Generic;
3
4  namespace GarbageCollector
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              while (true)
11              {
12                  var grosseKlasse = new GrosseKlasse();
13              }
14          }
15      }
16  }
```

Wir erzeugen in einer unendlichen Schleife immer neue Instanzen der Klasse `GrosseKlasse`. Allerdings ist die Variable, der wir die neu erzeugte Instanz von `GrosseKlasse` zuweisen, in der Schleife deklariert, das heißt, die Variable `grosseKlasse` „lebt“ jeweils nur für einen Schleifendurchgang, dann wird die Referenz aufgehoben und beim nächsten Schleifendurchgang eine neue Referenz auf eine neue Instanz von `GrosseKlasse` erstellt und so weiter.

Sehen wir uns den zugehörigen Speicherverlauf an.

## 10.3 Der Garbage Collector: Eine automatische Speicherverwaltung



**Abb. 10.3.8** Speicherverlauf des Experimentierprogramms wenn der Speicher immer wieder neu belegt wird

Der Speicherverlauf steigt und fällt. Der Garbage Collector wird selbstständig aktiv und gibt den Speicher wieder frei.

Betrachten wir zum Abschluss noch ein letztes Szenario:

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace GarbageCollector
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var liste = new List<GrosseKlasse>();
11
12             while (true)
13             {
14                 var grosseKlasse = new GrosseKlasse();
15                 liste.Add(grosseKlasse);
16             }
17         }
18     }
19 }
```

10

Wir erzeugen in einer unendlichen Schleife immer neue Objekte vom Typ `GrosseKlasse` und fügen sie einer Liste hinzu. Das sorgt dafür, dass wir die Referenzen auf unsere neu erzeugten Objekte nie aufheben und damit unser Speicherverbrauch immer weiter steigt, weil der Garbage Collector keinen Speicher freigeben kann.

## 10 Objektorientierung für Fortgeschrittene



**Abb. 10.3.9** Speicherverlauf des Experimentierprogramms wenn der Speicherverbrauch immer weiter steigt

Das Speicherdiagramm zeigt, dass der Garbage Collector mehrfach gelaufen ist, aber keinen Speicher freigeben konnte. Der aktuelle Speicherverbrauch in diesem Beispiel beträgt 62 GB und das, obwohl der für dieses Beispiel verwendete Computer nur 16 GB Hauptspeicher besitzt. Das ist deshalb möglich, weil das Betriebssystem den Anwendungen mehr Speicher zuteilen kann, als eigentlich vorhanden ist. Dabei wird aktuell nicht benötigter Hauptspeicher auf die Festplatte ausgelagert und wenn dieser Speicher dann wieder benötigt wird, wird er wieder von der Festplatte geladen. Dieses Verfahren wird Memory Swapping genannt.

Als Programmierer sollten wir vermeiden, das Betriebssystem in die Situation zu bringen, dass es ständig Speicher aus- und dann wieder einlagern muss, weil dadurch das gesamte System sehr stark verlangsamt wird.

Bei unserem Beispielprogramm ist sofort ersichtlich, dass das Programm immer weiter Speicher „frisst“ und den Speicher nie wieder zurückgibt. In komplexeren Szenarien kann diese Situation aber ungewollt eintreten und das Programm kann dann nach intensiver Benutzung mit einem „Out of Memory“-Fehler abstürzen. Bei Desktop-Programmen ist das zwar ärgerlich und der Benutzer kann nicht gespeicherte Daten verlieren, aber er kann das Programm wenigstens neu starten. Bei Serverprogrammen kann der Schaden wesentlich größer sein, da hier alle Benutzer des jeweiligen Serverprogramms betroffen sind und zum Neustart des Serverprogramms ein Administrator benötigt wird.

Durch unsere Versuche mit dem Garbage Collector haben wir gesehen, dass der Garbage Collector verwaltete Ressourcen, also Variablen und Listenstrukturen in C#, sehr gut allein verwalten kann. Allerdings sollten wir als Programmierer darauf achten, dass unsere Programme nicht ständig neuen Speicher belegen und die Referenz darauf nie wieder aufgehoben wird. Das Aufheben einer Referenz erreichen wir, indem wir der entsprechenden Variablen null zuweisen.

Sollte es Ihnen einmal passieren, dass eines Ihrer Programme immer weiter Speicher frisst und dann abstürzt, wenn kein Speicher mehr zu kriegen ist, dann ist der erste Verdächtige ein Cache, den Sie eventuell in Ihrer Applikation implementiert haben.

Ein Cache ist in C# üblicherweise eine Listenstruktur, an die Daten hinzugefügt werden, die von der Festplatte oder einem anderen Ort mit geringer Zugriffsgeschwindigkeit geladen werden, um dann, wenn solche Daten zum wiederholten Mal benötigt werden, schnell auf diesen Cache zugreifen zu können. Als Programmierer müssen wir so einen Cache irgendwie begrenzen, so dass - im einfachsten Fall - beim Erreichen einer Obergrenze, dem Cache keine neuen Objekte mehr hinzugefügt werden oder besser: Objekte, die nur mit einer geringen Wahrscheinlichkeit wieder benötigt werden, aus dem Cache entfernt werden.

Wenn wir es mit sogenannten nicht verwalteten Ressourcen zu tun haben, müssen wir dem Garbage Collector etwas unter die Arme greifen. Wie das geht, betrachten wir im Kapitel über die den Zugriff auf Dateien, wenn wir die erste nicht verwaltete Ressource kennen, lernen.

## 10.4 Polymorphie: Virtuelle Methoden und Properties

In die diesem Kapitel betrachten wir einen weiteren fortgeschrittenen Aspekt der objektorientierten Programmierung: die Polymorphie. Der Begriff leitet sich aus dem Griechischen ab und lässt sich am besten mit Vielgestaltigkeit übersetzen. Zu dem Thema kann man sehr viel abstrakte Theorie schreiben, aber am besten versteht man die Polymorphie an einem praktischen Beispiel. Dazu sehen wir uns die Klasse `Adresse` und die Klasse `Person`, die wir bereits bei der Vererbung kennengelernt haben, nochmals an.

10

```
1  using System;
2
3  namespace Polymorphie
4  {
5      public class Adresse
6      {
7          public string Ort { get; set; }
8          public string Strasse { get; set; }
9
10         public Adresse(string ort, string strasse)
11         {
12             Ort = ort;
13             Strasse = strasse;
14         }
15
16         public void AusgabeAdresse()
17         {
18             Console.WriteLine(Strasse);
19             Console.WriteLine(Ort);
20         }
21     }
22 }
```

## 10 Objektorientierung für Fortgeschrittene

```
1  using System;
2
3  namespace Polymorphie
4  {
5      public class Person : Adresse
6      {
7          public string Vorname { get; set; }
8          public string Nachname { get; set; }
9
10         public Person(string vorname, string nachname,
11                     string ort, string strasse):base(ort, strasse)
12         {
13             Vorname = vorname;
14             Nachname = nachname;
15         }
16
17         public void AusgabePerson()
18         {
19             Console.WriteLine($"{Vorname} {Nachname}");
20             AusgabeAdresse();
21         }
22     }
23 }
```

Das ist zunächst einmal nichts Neues. Die Klasse `Person` erbt von der Klasse `Adresse`. Die Methode `AusgabePerson()` gibt die Properties `Vorname` und `Nachname` am Bildschirm aus und verwendet zur Ausgabe von `Strasse` und `Ort` die von der Klasse `Adresse` geerbte Methode `AusgabeAdresse()`.

Als nächstes betrachten wir ein kleines Experiment mit den beiden Klassen.

```
1  using System.Collections.Generic;
2
3  namespace Polymorphie
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var adresse = new Adresse("Neustadt",
10                         "Hauptstrasse 25");
11              var person = new Person("Max", "Mustermann",
12                         "Musterhausen", "Musterstraße 1");
13
14              var adressListe = new List<Adresse>();
15
16              adressListe.Add(adresse);
17              adressListe.Add(person);
18
19              foreach(var element in adressListe)
20              {
21                  element.AusgabeAdresse();
22              }
23 }
```

```

23     }
24 }
25 }
```

Zuerst erstellen wir eine Instanz der Klasse `Adresse` und eine Instanz der Klasse `Person`. Dann erzeugen wir eine neue Liste von Typ `List<Adresse>` und weisen sowohl die Instanz der Klasse `Adresse` als auch die Instanz der Klasse `Person` dieser Liste zu.

Obwohl wir bei der Deklaration der Liste festgelegt haben, dass sie Objekte vom Typ `Adresse` enthalten soll, erhalten wir keine Fehlermeldung, wenn wir der Liste ein Objekt vom Typ `Person` zuweisen.

Das ist eines der Merkmale von Polymorphie: Abgeleitete Klassen können genauso verwendet werden wie ihre Elternklasse.

Am Ende unseres Programms laufen wir mit einer Schleife über die Liste und rufen für jedes Element der Liste die Methode `AusgabeAdresse()` auf. Das funktioniert auch für das Objekt vom Typ `Person` in der Liste, da die Klasse `Person` die Methode `AusgabeAdresse()` von der Klasse `Adresse` erbt.

Allerdings können wir in unserer Schleife nur `Ort` und `Strasse` ausgeben, da innerhalb der Schleife das Objekt vom Typ `Person` so behandelt wird, als wäre es vom Typ `Adresse` und die Klasse `Adresse` stellt nur eine Methode zur Ausgabe von `Ort` und `Strasse` zur Verfügung.

```

Microsoft Visual Studio-Debugging-Konsole
Hauptstrasse 25
Neustadt
Musterstraße 1
Musterhausen

C:\Users\rober\source\repos\CSharpBuch\Polymorphie\Polymorphie\bin\Debug\netcoreapp3.1\Polymorphie.exe (Prozess "6808")
wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.4.1** Ausgabe der Adressliste mit einem Adress-Objekt und einem Person-Objekt

Jetzt ändern wir die Klasse `Adresse` wie folgt ab:

```

1 using System;
2
3 namespace Polymorphie
4 {
5     public class Adresse
6     {
7         public string Ort { get; set; }
8         public string Strasse { get; set; }
9
10        public Adresse(string ort, string strasse)
```

## 10 Objektorientierung für Fortgeschrittene

```

11     {
12         Ort = ort;
13         Strasse = strasse;
14     }
15
16     public virtual void Ausgabe()
17     {
18         Console.WriteLine(Strasse);
19         Console.WriteLine(Ort);
20     }
21 }
22 }
```

Wir haben die Methode `AusgabeAdresse()` in `Ausgabe` umbenannt und das Schlüsselwort `virtual` zwischen Zugriffsmodifizierer und Rückgabetyp der Methode eingefügt. Die Umbenennung der Methode ist nicht zwingend. Aber wir werden später sehen, dass der Name `Ausgabe` jetzt besser zum Zweck der Methode passt. Durch das Schlüsselwort `virtual` haben wir die Methode als virtuell gekennzeichnet. Wozu das gut ist, sehen wir, wenn wir eine modifizierte Variante der Klasse `Person` betrachten.

```

1 using System;
2
3 namespace Polymorphie
4 {
5     public class Person : Adresse
6     {
7         public string Vorname { get; set; }
8         public string Nachname { get; set; }
9
10        public Person(string vorname, string nachname,
11                      string ort, string strasse):base(ort, strasse)
12        {
13            Vorname = vorname;
14            Nachname = nachname;
15        }
16
17        public override void Ausgabe()
18        {
19            Console.WriteLine($" {Vorname} {Nachname}");
20            base.Ausgabe();
21        }
22    }
23 }
```

Die Methode `AusgabePerson()` heißt jetzt auch `Ausgabe` wie in der Klasse `Adresse`. Zusätzlich haben wir das Schlüsselwort `override` zwischen den Zugriffsmodifizierer und den Rückgabetyp der Methode eingefügt. Damit überschreiben wir für die Klasse `Person` die von der Klasse `Adresse` geerbte virtuelle Methode `Ausgabe`. Mit der Anweisung `base.Ausgabe();` rufen wir die Originalmethode der Basisklasse `Adresse` auf, um das Ausgeben von `Ort` und `Strasse` zu erledigen.

## 10.4 Polymorphie: Virtuelle Methoden und Properties

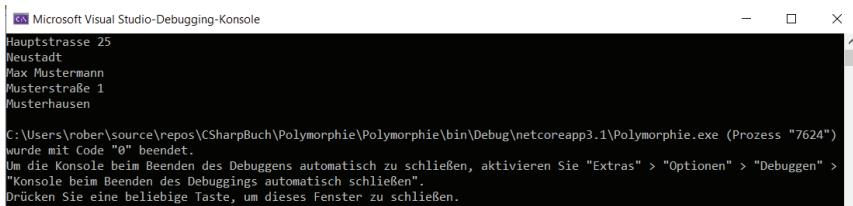
Da wir die Methoden `AusgabeAdresse()` und `AusgabePerson()` umbenannt haben, benötigen wir auch im Hauptprogramm eine kleine Anpassung:

```

1  using System.Collections.Generic;
2
3  namespace Polymorphie
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var adresse = new Adresse("Neustadt",
10                 "Hauptstrasse 25");
11              var person = new Person("Max", "Mustermann",
12                 "Musterhausen", "Musterstraße 1");
13
14              var adressListe = new List<Adresse>();
15
16              adressListe.Add(adresse);
17              adressListe.Add(person);
18
19              foreach(var element in adressListe)
20              {
21                  element.Ausgabe();
22              }
23          }
24      }
25 }
```

10

In der Schleife wird jetzt nicht mehr die Methode `AusgabeAdresse()`, sondern die Methode `Ausgabe` gerufen. Wenn wir das Programm starten, sehen wir, dass jetzt für das Objekt `Person` die Properties `Vorname` und `Nachname` mit ausgegeben werden.



```

Microsoft Visual Studio-Debugging-Konsole
Hauptstrasse 25
Neustadt
Max Mustermann
Musterstraße 1
Musterhausen

C:\Users\rrober\source\repos\CSharpBuch\Polymorphie\Polymorphie\bin\Debug\netcoreapp3.1\Polymorphie.exe (Prozess "7624")
wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 10.4.2** Ausgabe der Adressliste mit einem `Adresse`-Objekt und einem `Person`-Objekt mit einer virtuellen Methode

Der echte Typ eines Objektes wird bei seiner Erzeugung festgelegt. Mit der Anweisung:

```

1  var person = new Person("Max", "Mustermann", "Musterhausen",
2     "Musterstraße 1");
```

## 10 Objektorientierung für Fortgeschrittene

erzeugen wir ein Objekt vom Typ Person und weisen es einer Variablen vom Typ Person zu.

Mit der Anweisung:

```
1 Adresse person = new Person("Max", "Mustermann", "Musterhausen",
2 "Musterstraße 1");
```

erzeugen wir ein Objekt vom Typ Person und weisen es einer Variablen vom Typ Adresse zu. Über die Variable person, die jetzt vom Typ Adresse ist, können wir nur auf die von der Klasse Adresse definierten Properties Ort und Strasse zugreifen, da wir den Typ Adresse explizit für die Variable person festgelegt haben. Aber intern hält die Variable person ein Objekt vom Typ Person mit allen Möglichkeiten der Klasse Person, die sind allerdings für die Variable person verborgen, da hier der Typ Adresse gilt.

Aber die Methode Ausgabe ist virtuell und existiert in beiden Klassen in unterschiedlichen Implementierungen. Daher wird die Anweisung person.Ausgabe() immer die Variante der Methode aufrufen, die von der Klasse Person implementiert wurde. Wir erinnern uns: Das Objekt person wurde von einem Konstruktor der Klasse Person erzeugt.

Mit einem Type Cast können wir die verborgenen Fähigkeiten der Klasse Person wieder zugänglich machen.

```
1 using System;
2 using System.Collections.Generic;
3
4 namespace Polymorphie
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Adresse adresse = new Person("Max", "Mustermann",
11             "Musterhausen", "Musterstraße 1");
12             Person person = (Person)adresse;
13
14             Console.WriteLine(person.Nachname);
15
16         }
17     }
18 }
```

Ein Objekt wird vom Konstruktor der Klasse Person erzeugt und in der Variablen adresse vom Typ Adresse gespeichert. Das ist möglich, weil die Klasse Person von der Klasse Adresse abgeleitet ist.

Anschließend wenden wir einen Type Cast auf die Variable `adresse` an und legen damit fest, dass das Ergebnis des Casts als Objekt vom Typ `Person` anzusehen ist. Das Ergebnis des Cast weisen wir der Variablen `person` vom Typ `Person` zu. Um zu zeigen, dass es sich wirklich um ein Objekt vom Typ `Person` handelt, geben wir `person.Nachname` am Bildschirm aus.

Der Cast funktioniert nur deswegen, weil das zu castende Objekt wirklich vom Typ `Person` ist. Hätten wir das Objekt mit einem Konstruktor der Klasse `Adresse` erzeugt, würde ein Cast zu `Person` zu dem Fehler `InvalidCastException` führen.

Wir können eine Variable auch prüfen, ob sie unserem gewünschten Typ entspricht:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Polymorphie
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              Adresse objekt1 = new Adresse("Musterhausen",
11                  "Musterstraße 1");
12              Person objekt2 = new Person("Max", "Mustermann",
13                  "Musterhausen", "Musterstraße 1");
14
15              UntersucheObjekt("objekt1", objekt1);
16              UntersucheObjekt("objekt2", objekt2);
17          }
18
19          static void UntersucheObjekt(string name, Adresse obj)
20          {
21              Person person = obj as Person;
22              if(person == null)
23              {
24                  Console.WriteLine($"Das Objekt {name} hat den
25                      Typ Adresse");
26              }
27              else
28              {
29                  Console.WriteLine($"Das Objekt {name} hat den
30                      Typ Person");
31              }
32          }
33      }
34 }
```

Im Hauptprogramm erzeugen wir die Objekte `objekt1` und `objekt2` vom Typ `Adresse` und vom Typ `Person`. Mit der Methode `UntersucheObjekt()` prüfen wir, welchen Typ das übergebene Objekt hat. Dazu weisen wir das übergebene Objekt

## 10 Objektorientierung für Fortgeschrittene

der lokalen Variablen `person` zu, welche vom Typ `Person` ist. Am Ende der Zuweisung schreiben wir das Schlüsselwort `as` und den Typ `Person`. Diese Syntax bewirkt etwas ähnliches wie ein Type Cast, aber im Gegensatz zu einem Type Cast gibt es keinen Fehler, falls der Cast nicht möglich ist, sondern die Variable `person` erhält in diesem Fall den Wert `null`.

```

Microsoft Visual Studio-Debugging-Konsole
Das Objekt objekt1 hat den Typ Adresse
Das Objekt objekt2 hat den Typ Person

C:\Users\rober\source\repos\CSharpBuch\Polymorphie\Polymorphie\bin\Debug\netcoreapp3.1\Polymorphie.exe (Prozess "19192")
wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 10.4.3** Ausgabe des Testprogramms für `objekt1` und `objekt2`

Die Klasse `Person` kann die Methode `Ausgabe()` der Klasse `Adresse` überschreiben, weil die Methode `Ausgabe()` virtuell ist. Wenn Sie in Ihren eigenen Programmen feststellen, dass Sie eine Methode einer Basisklasse in einer abgeleiteten Klasse überschreiben wollen, können Sie die Methode der Basisklasse durch Hinzufügen des Schlüsselworts `virtual` einfach in eine virtuelle Methode verwandeln.

Wenn Sie eine Klasse einer externen Klassenbibliothek als Basisklasse verwenden, können Sie eine Methode, die nicht virtuell ist, auch nicht zu einer virtuellen Methode machen, da Sie bei einer externen Klassenbibliothek den Programmcode nicht verändern können.

C# erlaubt auch die Überschreibung von nicht-virtuellen Methoden. Dieser Mechanismus wird das Verbergen von Methoden genannt. Im folgenden Beispiel wollen wir diese Technik näher untersuchen.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace PolymorphieMitNew
6  {
7      public class NeueStringListe : List<string>
8      {
9          public new void Add(string item)
10         {
11             base.Add(item);
12             Console.WriteLine($"\"{item}\" wurde der Liste
13             hinzugefügt.");
14         }
15     }
16 }

```

Die Klasse `NeueStringListe` leiten wir von `List<string>` ab. `List<string>` ist Bestandteil des .NET-Frameworks und wir können den Programmcode von

## 10.4 Polymorphie: Virtuelle Methoden und Properties

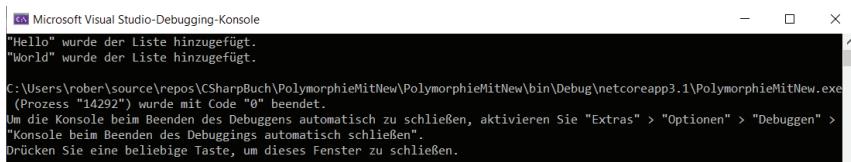
`List<string>` nicht verändern. Die Methode `Add` der Klasse `List<string>` ist nicht virtuell, daher können wir Sie mit dem Schlüsselwort `override` nicht überschreiben. Deshalb verwenden wir anstelle von `override` das Schlüsselwort `new`. In unserer neuen Variante der Methode `Add()` rufen wir zuerst die Originalmethode der Basisklasse auf und dann geben wir einen Text am Bildschirm aus, der anzeigt, welches Element der Liste hinzugefügt wurde.

Als nächste wollen wir die Klasse `NeueStringListe` testen:

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace PolymorphieMitNew
5  {
6      public class Program
7      {
8          static void Main(string[] args)
9          {
10             var liste = new NeueStringListe();
11             liste.Add("Hello");
12             liste.Add("World");
13         }
14     }
15 }
```

Im Hauptprogramm erzeugen wir eine Instanz der Klasse `NeueStringListe`, dann rufen wir zweimal die Methode `Add()` auf, um der Liste zwei Einträge hinzuzufügen. Wie erwartet, macht unser Testprogramm beim Hinzufügen eines Elements zur Liste eine Bildschirmausgabe.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output text is:  
"Hello" wurde der Liste hinzugefügt.  
"World" wurde der Liste hinzugefügt.

Below the console window, the status bar displays the path: C:\Users\rober\source\repos\SharpBuch\PolymorphieMitNew\PolymorphieMitNew\bin\Debug\netcoreapp3.1\PolymorphieMitNew.exe (Prozess "14292") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

Abb. 10.4.4 Ausgabe des Testprogramms

Nach diesem Beispiel könnten wir annehmen, dass sich Methoden, die mit `new` überschrieben sind, genauso verhalten wie virtuelle Methoden, die mit `override` überschrieben sind. Dem ist aber nicht so. Es gibt einen kleinen, aber wichtigen Unterschied zwischen den beiden Techniken. Eine mit `new` überschriebene Methode ist nicht virtuell, das heißt beim Aufruf über die Basisklasse eines Objekts wird auch nur die Methode der Basisklasse ausgeführt, nicht die überschriebene Methode des eigentlichen Objekts. Das folgende Beispiel verdeutlicht dieses Verhalten:

```

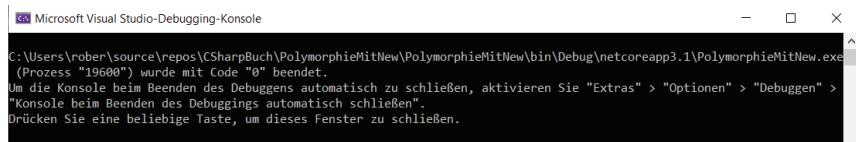
1  using System;
2  using System.Collections.Generic;
```

## 10 Objektorientierung für Fortgeschrittene

```

3
4 namespace PolymorphieMitNew
5 {
6     public class Program
7     {
8         static void Main(string[] args)
9         {
10             List<string> liste = new NeueStringListe();
11             liste.Add("Hello World");
12         }
13     }
14 }
```

Wir erzeugen eine Instanz der Klasse NeueStringListe und weisen sie der Variablen liste zu. Die Variable liste ist vom Typ List<string>, da List<string> die Basisklasse von NeueStringListe ist, funktioniert diese Zuweisung. Der Aufruf liste.Add() führt aber die Methode Add() der Basisklasse List<string> aus und es erfolgt keine Bildschirmausgabe beim Hinzufügen eines Elements zur Liste.



**Abb. 10.4.5** Das Testprogramm macht keine Bildschirmausgabe.

Abschließend möchte ich noch erwähnen, dass auch Properties als virtuell deklariert und mit new oder override überschrieben werden können. Überschreiben beziehungsweise Verbergen von Properties erfolgt analog zum Vorgehen bei Methoden.

Im folgenden Beispiel betrachten wir wieder eine Variante der Klasse Person:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace VirtuelleProperties
6 {
7     public class Person
8     {
9         public string Vorname { get; set; }
10        public string Nachname { get; set; }
11
12        public virtual string Briefanrede
13        {
14            get
15            {
16                return $"Sehr geehrter Herr {Nachname},";
17            }
18        }
19    }
```

20 }

Die Property `Briefanrede` ist virtuell deklariert.

Jetzt leiten wir noch die Klasse `PersonMitTitel` von der Klasse `Person` ab.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace VirtuelleProperties
6  {
7      public class PersonMitTitel : Person
8      {
9          public string Titel { get; set; }
10         public override string Briefanrede
11         {
12             get
13             {
14                 return $"Sehr geehrter Herr {Titel} {Nachname},";
15             }
16         }
17     }
18 }
```

Die Klasse `PersonMitTitel` überschreibt die Property `Briefanrede` der Klasse `Person` mit einer eigenen Implementierung.

10

Mit dem folgenden Testprogramm können wir den Unterschied der beiden Implementierungen der Property `BriefAnrede` demonstrieren.

```
1  using System;
2
3  namespace VirtuelleProperties
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var person = new Person
10             {
11                 Vorname = "Robert",
12                 Nachname = "Schiefele"
13             };
14
15             var personMitTitel = new PersonMitTitel
16             {
17                 Vorname = "Gustav",
18                 Nachname = "Müller-Lüdenscheid",
19                 Titel = "Dr."
20             };
21
22             Console.WriteLine(person.Briefanrede);
```

## 10 Objektorientierung für Fortgeschrittene

```

23             Console.WriteLine(personMitTitel.Briefanrede);
24         }
25     }
26 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the following text:

```

Microsoft Visual Studio-Debugging-Konsole
Ihr geehrter Herr Schreiter,
Sehr geehrtes Herr Dr. Müller-Lüdenscheid,
```

Das Projekt "VirtuelleProperties" wurde mit Code "W" beendet.

Um die Konsole beim Beenden des Debuggers automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".

Klicken Sie eine beliebige Taste, um dieses Fenster zu schließen.

Abb. 10.4.6 Ausgabe einer virtuellen Property und ihrer Überschreibung

## 10.5 Abstrakte und versiegelte Klassen

Mit den beiden Schlüsselwörtern `abstract` und `sealed` kann das Verhalten von Klassen bezüglich der Vererbung feinjustiert werden. Wenn Sie die Klassenhierarchie für ein Programm oder eine neue Funktionalität planen, werden Sie zunächst versuchen, Klassen zu identifizieren, die ähnlich sind, das heißt Klassen, die zum Teil die gleichen Properties und/oder Methoden haben. Eine Klasse, die nur die Properties und/oder Methoden enthält, die mehrere andere Klassen auch haben, könnte als Basisklasse für eine Vererbung dienen. Bei derartigen Basisklassen macht es meistens keinen Sinn, Instanzen dieser Klassen zu erzeugen, sie dienen lediglich als Basisklasse, um andere Klassen davon abzuleiten.

Solche Klassen werden als abstrakte Klassen bezeichnet. In C# schreibt man nach dem Zugriffsmodifizierer einer solchen Klasse das Schlüsselwort `abstract`. Damit verhindern Sie, dass eine Instanz dieser Klasse erstellt werden kann.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace AbstrakteKlassen
6  {
7      public abstract class Fahrzeug
8      {
9          public int Bezeichnung { get; set; }
10         public int AnzahlRaeder { get; set; }
11     }
12 }
```

Die Klasse `Fahrzeug` ist mit dem Schlüsselwort `abstract` versehen. Sie kann als Basisklasse für verschiedene Fahrzeug-Klassen dienen, wie zum Beispiel `Auto`, `Fahrrad`, `Motorrad` etc. Aber das Erzeugen einer Instanz der Klasse `Fahrzeug` wird bereits vom Compiler verhindert.

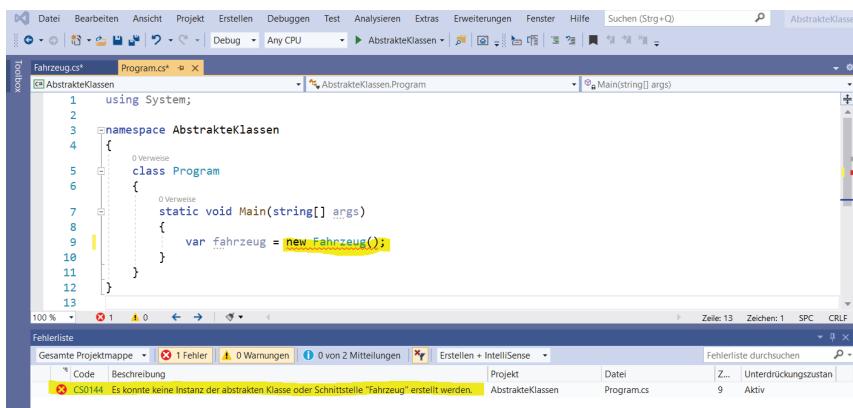


Abb. 10.5.1 Eine Instanz einer abstrakten Klasse kann nicht erstellt werden.

In C# werden auch abstrakte Methoden unterstützt. Bei abstrakten Methoden steht das Schlüsselwort `abstract` zwischen dem Zugriffsmodifizierer und dem Rückgabewert der Methode. Die Klasse `Fahrzeug` könnte zum Beispiel die abstrakte Methode `Fahren()` bereitstellen.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace AbstrakteKlassen
6  {
7      public abstract class Fahrzeug
8      {
9          public int Bezeichnung { get; set; }
10         public int AnzahlRaeder { get; set; }
11
12         public abstract void Fahren();
13     }
14 }
15 }
```

10

Eine abstrakte Methode hat keinen ausführbaren Programmcode, daher muss nach der Signatur der Methode auch ein Semikolon angegeben werden. Eine abstrakte Methode ist automatisch auch eine virtuelle Methode. Eine Methode, die mit dem Schlüsselwort `virtual` als virtuell deklariert wurde, kann überschrieben werden. Eine Methode, die als abstrakt deklariert wurde, muss überschrieben werden. Jede Klasse, die von der Klasse `Fahrzeug` erbt, muss die Methode `Fahren()` mit `override` überschreiben. Nur abstrakte Klassen können auch abstrakte Methoden enthalten.

Das Gegenstück zum Schlüsselwort `abstract` ist das Schlüsselwort `sealed` (Englisch für *versiegelt*). Wenn wir eine Klassenhierarchie aufbauen und mit einer Basis-

## 10 Objektorientierung für Fortgeschrittene

klasse Fahrzeug beginnen, liegt es nahe, die beiden Klassen Auto und Fahrrad von Fahrzeug abzuleiten. Des Weiteren können wir von Fahrrad eine Klasse Motorrad ableiten (durch Hinzufügen eines Motors). Aber bei der Klasse Auto könnten wir zum Beispiel zu dem Schluss kommen, dass es keinen Sinn macht, die Klasse Auto weiter abzuleiten. Um das Ableiten der Klasse Auto zu verhindern, können wir das Schlüsselwort sealed verwenden.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace AbstrakteKlassen
6  {
7      public sealed class Auto : Fahrzeug
8      {
9          public Auto()
10         {
11             AnzahlRaeder = 4;
12         }
13     }
14 }
```

Nach dem Zugriffsmodifizierer schreiben wir das Schlüsselwort sealed, damit kann die Klasse Auto nicht mehr abgeleitet werden.

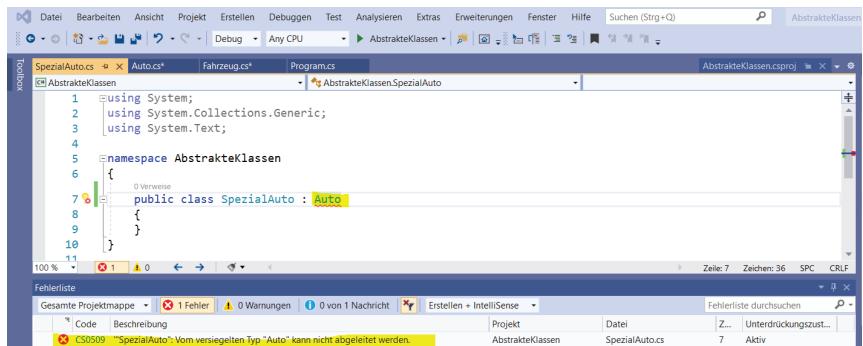


Abb. 10.5.2 Eine versiegelte Klasse kann nicht abgeleitet werden.

Das Schlüsselwort sealed kann wie abstract auch für Methoden verwendet werden. Versiegelte Methoden können mit override nicht mehr überschrieben werden. Es können nur Methoden versiegelt werden, die Überschreibungen sind.

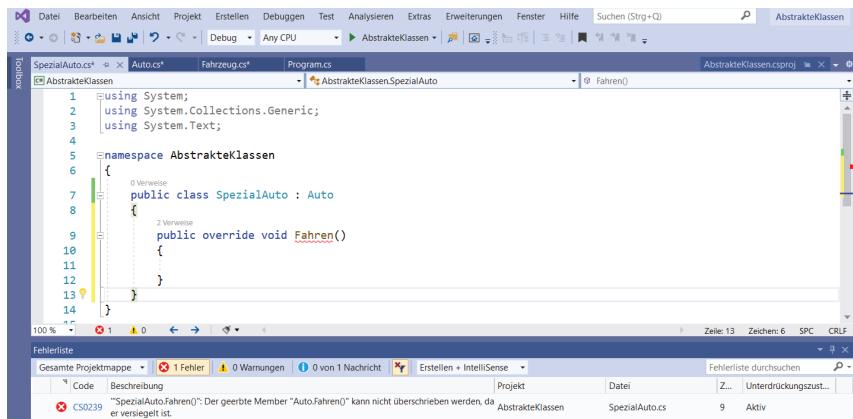
```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace AbstrakteKlassen
```

```

6  {
7      public class Auto : Fahrzeug
8      {
9          public Auto()
10         {
11             AnzahlRaeder = 4;
12         }
13
14         public sealed override void Fahren()
15         {
16             Console.WriteLine("Ein Auto fährt");
17         }
18     }
19 }
```

Diese Variante der Klasse `Auto` ist nicht mehr versiegelt, aber die Überschreibung der von der Klasse `Fahrzeug` geerbten Methode `Fahren()` ist jetzt versiegelt. Dadurch kann die Klasse `SpezialAuto` jetzt von `Auto` erben. Aber wenn `SpezialAuto` versucht, die Methode `Fahren()` zu überschreiben, führt das zu einem Fehler.



10

Abb. 10.5.3 Versiegelte Methoden können nicht überschrieben werden.

## 10.6 Erweiterungsmethoden

Normalerweise fügen Sie einer Klasse eine Methode hinzu, indem sie im zugehörigen Programmcode diese Methode schreiben. Wenn Ihnen eine Klasse aber nur in kompilierter Form, also als Bestandteil einer externen Klassenbibliothek, vorliegt, dann können Sie deren Programmcode auch nicht verändern oder erweitern.

Mit Erweiterungsmethoden können Sie einer Klasse Methoden hinzufügen, ohne den Programmcode dieser Klasse zu verändern und auch ohne den Programmcode dieser Klasse zu kennen.

## 10 Objektorientierung für Fortgeschrittene

Das heißtt, Sie können mit Erweiterungsmethoden auch Klassen aus einer externen Klassenbibliothek zum Beispiel aus dem .NET-Framework erweitern. Was zunächst spektakulär klingt, ist in Wirklichkeit nur ein Stück Syntactic Sugar. Aber trotz alle- dem ein praktisches Hilfsmittel für C#-Programmierer.

Obwohl die Klasse `String` aus dem .NET-Framework schon sehr viele Methoden zur String-Verarbeitung mitbringt, ergeben sich bei jedem neuen Projekt Methoden zur String-Verarbeitung, die hilfreich wären, aber im .NET-Framework nicht existieren. Bevor es Erweiterungsmethoden gab, hat man sich normalerweise eine statische Klasse mit statischen Methoden geschrieben die, die zusätzliche String-Verarbeitung erleidigt haben. Das folgende Beispiel zeigt dieses Verfahren:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Erweiterungsmethoden
6  {
7      public static class StringErweiterungen
8      {
9          public static string AlsBereinigt(string einString)
10         {
11             if(einString == null)
12             {
13                 return string.Empty;
14             }
15             else
16             {
17                 return einString.Trim();
18             }
19         }
20     }
21 }
```

Ein typisches Problem der String-Verarbeitung ist das Bereinigen von Strings. Das heißtt, man möchte sogenannte „White Spaces“, also Leerzeichen, Zeilenumbrüche, Tabulatoren etc. am Anfang und am Ende eines Strings entfernen. Das erleidigt die Methode `Trim()` der Klasse `String` aus dem .NET-Framework. Wenn wir `einString.Trim()` aufrufen und die Variable `einString` hat den Wert `null`, dann erhalten wir eine unschöne `NullReferenceException`. Die statische Methode `AlsBereinigt()` nimmt einen Parameter vom Typ `string` entgegen und falls dieser den Wert `null` hat, gibt die Methode einen leeren String zurück und wenn nicht, wird ein mit der Methode `Trim()` bereinigter String zurückgegeben. Damit führt das Codefragment:

```
1  string text = null;
2  Console.WriteLine(StringErweiterungen.AlsBereinigt(text));
```

nicht mehr zu einem Fehler.

Mit einer kleinen Modifikation können wir die statische Methode `AlsBereinigt()` zu einer Erweiterungsmethode umfunktionieren.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Erweiterungsmethoden
6  {
7      public static class StringErweiterungen
8      {
9          public static string AlsBereinigt(this string einString)
10         {
11             if(einString == null)
12             {
13                 return string.Empty;
14             }
15             else
16             {
17                 return einString.Trim();
18             }
19         }
20     }
21 }
```

Wir müssen lediglich das Schlüsselwort `this` vor dem Typbezeichner des Übergabeparameters `einString` einfügen. Damit ist die Methode `AlsBereinigt()` eine Erweiterungsmethode. Welchen Unterschied das macht, sehen wir in der Verwendung der Methode.

```
1  string text = null;
2  Console.WriteLine(text.AlsBereinigt());
```

Wir können die Methode `AlsBereinigt()` so verwenden, als wäre sie eine Methode der Klasse `String`. Sie kann als Ersatz für die Methode `Trim()` verwendet werden – mit dem Unterschied, dass es keinen Fehler mehr gibt, wenn die zugehörige String-Variable den Wert `null` hat.

Natürlich können wir einer Erweiterungsmethode auch zusätzliche Parameter übergeben.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Erweiterungsmethoden
6  {
7      public static class StringErweiterungen
8      {
9          public static string AlsKuerzel(this string einString,
10                                         int maximaleAnzahlZeichen)
```

## 10 Objektorientierung für Fortgeschrittene

```

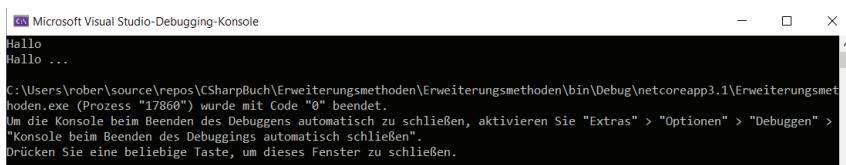
11     {
12         if (einString.Length>maximaleAnzahlZeichen)
13         {
14             return $"(einString.Substring(0,
15             maximaleAnzahlZeichen - 3)}...";
16         }
17     else
18     {
19         return einString;
20     }
21 }
22 }
23 }
```

Die Erweiterungsmethode `AlsKuerzel()` nimmt zusätzlich zum `this`-Parameter `einString` noch zusätzlich den Parameter `maximaleAnzahlZeichen` vom Typ `int` entgegen. Wenn `einString` länger ist als durch den Parameter `maximaleAnzahlZeichen` vorgegeben, wird `einString` abgekürzt und mit drei Punkten ergänzt, so dass die Gesamtlänge des Strings nicht größer ist als durch `maximaleAnzahlZeichen` vorgegeben. Zum Abkürzen verwenden wir die Methode `Substring()` der Klasse `String`, die einen Teil des Strings zurückgibt. Wir übergeben ihr als ersten Parameter den Startindex, der in unserem Fall 0 ist, da wir auf der linken Seite nichts abschneiden wollen. Als zweiten Parameter übergeben wir ihr die Anzahl Zeichen, die wir von unserem String behalten wollen. In unserem Fall ist das drei weniger als `maximaleAnzahlZeichen`. Wenn wir dann drei Punkte ergänzen, haben wir einen String, der genauso lang ist, wie durch `maximaleAnzahlZeichen` vorgegeben.

```

1 using System;
2
3 namespace Erweiterungsmethoden
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             string text1 = "Hallo";
10            string text2 = "Hallo Welt";
11            Console.WriteLine(text1.AlsKuerzel(9));
12            Console.WriteLine(text2.AlsKuerzel(9));
13        }
14    }
15 }
```

Im Hauptprogramm deklarieren wir zwei String-Variablen und belegen sie mit den Werten „Hallo“ und „Hallo Welt“. Dann geben wir die beiden String-Variablen als Kürzel mit maximal 9 Zeichen am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the following text:

```
Microsoft Visual Studio-Debugging-Konsole
Hallo
Hallo ...
C:\Users\rober\source\repos\CSharpBuch\Erweiterungsmethoden\Erweiterungsmethoden\bin\Debug\netcoreapp3.1\Erweiterungsmethoden.exe (Prozess "17860") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.6.1 Ausgabe des Testprogramms für die Methode AlsKuerzel()

## 10.7 Generische Klassen

In diesem Kapitel wollen wir uns dem Thema der generischen Klassen widmen. Aber bevor wir in dieses Thema einsteigen, möchte ich an dieser Stelle einen Einschub machen und kurz auf die beiden Themen: „Überschreiben der Methode `ToString()`“ sowie die Baumstrukturen eingehen. Erstens sind das zwei Themen, die Ihnen in der täglichen Programmierpraxis häufig begegnen und zweitens benötigen wir diese beiden Themen für ein sinnvolles Praxisbeispiel für generische Klassen.

Wenn Sie eine neue Klasse erstellen und für diese Klasse keine Basisklasse zur Vererbung angeben, dann erbt diese Klasse automatisch von der Klasse `Object`. Jede Klassenhierarchie in C# endet irgendwann bei einer Basisklasse, die von keiner anderen Klasse erbt.

Diese Basisklasse erbt dann von der Klasse `Object`. Damit erben alle Klassen entweder direkt oder indirekt von der Klasse `Object`. Das heißt die Methoden der Klasse `Object` stehen allen Klassen zur Verfügung. Eine dieser Methoden ist die Methode `ToString()`. Diese Methode kann überschrieben werden. Zum Beispiel liefert die Klasse `DateTime` eine Überschreibung von `ToString()`, die einen Datumswert in einen String umwandelt.

Übrigens: Die Methode `Console.WriteLine()` ist nicht für alle verschiedene Typen als Überladung implementiert, sondern ruft einfach die `ToString()`-Methode des übergebenen Objekts auf. Damit können mit `Console.WriteLine()` alle theoretisch denkbaren Objekte am Bildschirm ausgegeben werden. Betrachten wir das einmal am Beispiel einer selbstgeschriebenen Klasse:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace ToStringUeberschreiben
6  {
7      public class Person
8      {
9          public Person(string vorname, string nachname)
10         {
11     }
```

## 10 Objektorientierung für Fortgeschrittene

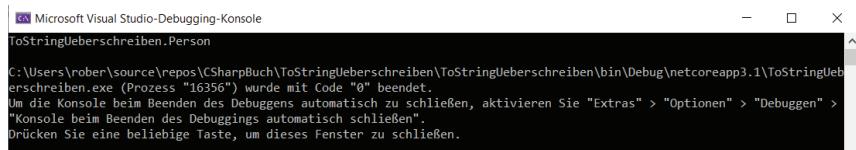
```

11         Vorname = vorname;
12         Nachname = nachname;
13     }
14
15     string Vorname { get; set; }
16     string Nachname { get; set; }
17 }
18 }
```

```

1 using System;
2
3 namespace ToStringUeberschreiben
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var person = new Person("Robert", "Schiefele");
10            Console.WriteLine(person);
11        }
12    }
13 }
```

Wir verwenden hier wieder die bereits bekannte Klasse Person, erstellen eine Instanz dieser Klasse und geben sie mit `Console.WriteLine()` am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The content of the console is:

```

ToStringUeberschreiben.Person
C:\Users\rober\source\repos\CSharpBuch\ToStringUeberschreiben\ToStringUeberschreiben\bin\Debug\netcoreapp3.1\ToStringUeberschreiben.exe (Prozess "16356") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.7.1** Die Klasse Person ohne überschriebene `ToString()`-Methode

Da wir in der Klasse Person die Methode `ToString()` nicht überschrieben haben, wird die Originalmethode der Klasse `Object` aufgerufen. Diese Methode gibt den vollständigen Klassennamen der Klasse aus, also den Namen des zugehörigen Namensraums und den Namen der Klasse durch einen Punkt getrennt.

Als nächstes wollen wir die Methode `ToString()` überschreiben:

```

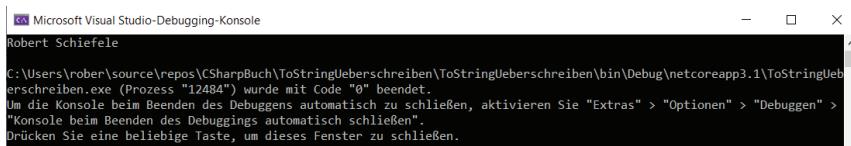
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ToStringUeberschreiben
6 {
7     public class Person
8     {
9         public Person(string vorname, string nachname)
10        {
11            Vorname = vorname;
```

```

12     Nachname = nachname;
13 }
14
15     string Vorname { get; set; }
16     string Nachname { get; set; }
17
18     public override string ToString()
19     {
20         return $" {Vorname} {Nachname}";
21     }
22 }
23 }
```

In der Überschreibung der Methode `ToString()` geben wir die Werte der Properties `Vorname` und `Nachname` mit einem Leerzeichen getrennt zurück.

Dadurch bekommen wir jetzt eine sinnvollere Ausgabe, wenn wir eine Instanz der Klasse `Person` am Bildschirm ausgeben.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The content of the console is:

```

Robert Schiefele
C:\Users\rober\source\repos\CSharpBuch\ToStringÜberschreiben\ToStringÜberschreiben\bin\Debug\netcoreapp3.1\ToStringÜberschreiben.exe (Prozess "12484") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.7.2** Die Klasse Person mit überschriebener `ToString()`-Methode

10

Bevor wir zum eigentlichen Thema generische Klassen kommen, möchte ich noch kurz das Thema Baumstrukturen vorstellen. Eine vollständige Behandlung von Baumstrukturen würde den Rahmen dieses Buches sprengen, daher begnügen wir uns mit einer kurzen Einführung, die ausreicht, um generische Klassen mit einem sinnvollen Beispiel zu behandeln.

Baumstrukturen gehören mit zu den häufigsten Strukturen in Computerprogrammen. Eines der prominentesten Beispiele ist die Datei und Verzeichnisstruktur des Dateiexplorers von Windows. Allgemein besteht eine Baumstruktur aus vielen sogenannten Knotenobjekten. Ein Knotenobjekt enthält die Nutzdaten des Knotens, wie zum Beispiel Name des Verzeichnisses und eine Liste von Dateiobjekten. Zusätzlich enthält ein Knotenobjekt noch eine Liste weiterer Knotenobjekte, die als Kind-Knoten beziehungsweise Unterknoten bezeichnet werden. Eine vollständige Baumstruktur kann durch ein einziges Knotenobjekt repräsentiert werden. Dieses Knotenobjekt nennt man den Wurzelknoten. Der Wurzelknoten enthält Unterknoten, die wiederum Unterknoten enthalten können und so weiter.

Im Folgenden sehen wir ein einfaches Knotenobjekt.

```

1  using System;
2  using System.Collections.Generic;
```

## 10 Objektorientierung für Fortgeschrittene

```

3  using System.Text;
4
5  namespace Baumstrukturen
6  {
7      public class Knoten
8      {
9          public Knoten(string name)
10         {
11             Name = name;
12             Unterknoten = new List<Knoten>();
13         }
14
15         public string Name { get; set; }
16
17         public List<Knoten> Unterknoten { get; set; }
18     }
19 }
```

Als Nutzdaten des Knoten speichern wir lediglich den Namen des Knoten in der String-Property `Name`. Des Weiteren hat unser Knotenobjekt die Property `Unterknoten` mit dem Typ `List<Knoten>`. Im Konstruktor übergeben wir einen Namen für den Knoten und weisen ihn der Property `Name` zu. Zum Start des Programms initialisieren wir im Konstruktor die Property `Unterknoten` mit einer leeren Liste.

Als nächstes wollen wir die Baumstruktur mit Daten füllen.

```

1  using System;
2
3  namespace Baumstrukturen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              //Instanziieren der Knoten
10             var wurzel = new Knoten("Wurzelknoten");
11             var knoten1 = new Knoten("Knoten1");
12             var knoten11 = new Knoten("Knoten11");
13             var knoten12 = new Knoten("Knoten12");
14             var knoten2 = new Knoten("Knoten2");
15             var knoten21 = new Knoten("Knoten21");
16             var knoten22 = new Knoten("Knoten22");
17
18             //Untereinanderhängen der Knoten
19             knoten1.Unterknoten.Add(knoten11);
20             knoten1.Unterknoten.Add(knoten12);
21             knoten2.Unterknoten.Add(knoten21);
22             knoten2.Unterknoten.Add(knoten22);
23             wurzel.Unterknoten.Add(knoten1);
24             wurzel.Unterknoten.Add(knoten2);
25         }
26     }
27 }
```

Zuerst instanziieren wir für den Wurzelknoten und alle Unter- und Unter-Unter-Knoten die nötigen Objekte und dann hängen wir die Knoten untereinander. Dazu fügen wir den Unterknoten-Properties der entsprechenden Knotenobjekte die Unterknotenobjekte hinzu.

Natürlich brauchen wir jetzt noch eine Methode, die den vollständigen Baum am Bildschirm ausgibt. Diese Ausgabe soll den Namen für jeden Knoten in einer Zeile am Bildschirm ausgeben und dabei die Kind-Knoten gegenüber den Eltern-Knoten nach rechts einrücken. Das erscheint auf den ersten Blick gar nicht so einfach, da unsere Baumstruktur theoretisch unendlich tief reichen kann und unsere Ausgabemethode für alle möglichen Bäume unserer Struktur funktionieren soll. Die Lösung für dieses Problem ist ein sogenannter „rekursiver Methodenaufruf“. Das heißt, dies ist eine Methode, die sich selbst aufruft. Wie das funktionieren kann, ohne dabei aus Versehen eine unendliche Schleife zu erzeugen, sehen wir im folgenden Beispielcode. Wie das Ergebnis aussehen soll, sehen Sie in Abbildung 10.7.3.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Xml.Serialization;
5
6  namespace Baumstrukturen
7  {
8      public class Knoten
9      {
10          private int tiefe;
11          private string einrueckung;
12          private string baumausgabe;
13
14          public Knoten(string name)
15          {
16              Name = name;
17              Unterknoten = new List<Knoten>();
18          }
19          public string Name { get; set; }
20
21          public List<Knoten> Unterknoten { get; set; }
22
23          public override string ToString()
24          {
25              tiefe = 0;
26              baumausgabe = "";
27              SetzeEinrueckung();
28
29              BaumAusbabeRekursiv(this);
30              return baumausgabe;
31
32          }
33
34          private void BaumAusbabeRekursiv(Knoten knoten)
35          {
```

## 10 Objektorientierung für Fortgeschrittene

```

36         baumausgabe += ${einrueckung}{knoten.Name}\r\n";
37
38     if (Unterknoten.Count > 0)
39     {
40         tiefe++;
41         SetzeEinrueckung();
42
43         foreach (var unterknoten in knoten.Unterknoten)
44         {
45             BaumAusgabeRekursiv(unterknoten);
46         }
47
48         tiefe--;
49         SetzeEinrueckung();
50     }
51 }
52
53
54     private void SetzeEinrueckung()
55     {
56         einrueckung = "";
57         for (var i = 0; i<tiefe; i++)
58         {
59             einrueckung += "    ";
60         }
61     }
62 }
63 }
```

Unsere Klasse Knoten hat drei private Membervariablen erhalten. In der int-Variablen `tiefe` speichern wir die aktuelle Baumtiefe während der Erzeugung der Ausgabe. Wenn wir die Zeile für den Wurzelknoten erzeugen, ist die Baumtiefe 0, wenn wir die Unterknoten des Wurzelknotens erzeugen, ist die Baumtiefe 1 und bei den Unterknoten der Unterknoten wird die Baumtiefe jeweils um eins erhöht.

Die String-Variable `einrueckung` speichert die aktuelle Einrückung beim Erzeugen der Ausgabe. Wenn wir die Zeile für den Wurzelknoten erzeugen, ist die Einrückung ein leerer String, da wir den Wurzelknoten nicht einrücken wollen. Wenn wir die Unterknoten des Wurzelknotens erzeugen, ist die Einrückung ein String mit vier Leerzeichen. Jedes Mal, wenn sich die Baumtiefe um eins erhöht, erhöht sich die Anzahl der Leerzeichen der Einrückung um vier. Die String-Variable `baumausgabe` speichert den gesamten Baum als String.

Die private Hilfsmethode `SetzeEinrueckung()` setzt die Variable `einrueckung` auf `tiefe * 4` Leerzeichen. Sie wird jedes Mal aufgerufen, nachdem die Variable `tiefe` einen neuen Wert bekommt.

Die überschriebene Methode `ToString()` der Klasse `Knoten` initialisiert die privaten Membervariablen `tiefe` und `baumausgabe` mit den Werten 0 und „Leerer String“ und initialisiert die private Membervariable `einrueckung` mit der Metho-

de `SetzeEinrueckung()`. Dadurch erhält `einrueckung` den Wert „Leerer String“, da `tiefe` mit 0 initialisiert wurde. Danach ruft `ToString()` die rekursive Methode `BaumausgabeRekursiv()` auf und übergibt der Methode mit dem Schlüsselwort `this` die aktuelle Instanz der Klasse `Knoten`.

Die Logik des Durchlaufens des ganzen Baums ist in der rekursiven Methode `BaumausgabeRekursiv()` implementiert. Die Methode `BaumausgabeRekursiv()` hängt an die private Membervariable `baumausgabe` die aktuelle Einrückung, den Wert der Property `Name` des übergebenen Objekts vom Typ `Knoten` und einen Zeilenumbruch an. Falls das `Knoten`-Objekt Unterknoten enthält, wird die private Membervariable `tiefe` um eins erhöht, die private Membervariable `einrueckung` mit der Methode `SetzeEinrueckung()` an die neue Tiefe angepasst und danach werden alle Unterknoten des übergebenen `Knoten`-Objekts durchlaufen. Innerhalb der Schleife ruft sich die Methode `BausgabeRekursiv()` selbst auf und übergibt dabei den aktuellen Unterknoten. Nach der Schleife setzen wir `tiefe` wieder um eins zurück und passen mit `SetzeEinrueckung()` die aktuelle Einrückung an. Mit dieser Technik werden systematisch alle Knoten der Baumstruktur durchlaufen und an die private Membervariable `baumausgabe` angehängt.

Am Ende der überschriebenen Methode `ToString()` wird der Wert von `baumausgabe` zurückgegeben.

In unserem Hauptprogramm können wir jetzt die ganze Baumstruktur am Bildschirm ausgeben.

10

```
1  using System;
2
3  namespace Baumstrukturen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              //Instanziieren der Knoten
10             var wurzel = new Knoten("Wurzelknoten");
11             var knoten1 = new Knoten("Knoten1");
12             var knoten11 = new Knoten("Knoten11");
13             var knoten12 = new Knoten("Knoten12");
14             var knoten2 = new Knoten("Knoten2");
15             var knoten21 = new Knoten("Knoten21");
16             var knoten22 = new Knoten("Knoten22");
17
18             //Untereinanderhängen der Knoten
19             knoten1.Unterknoten.Add(knoten11);
20             knoten1.Unterknoten.Add(knoten12);
21             knoten2.Unterknoten.Add(knoten21);
22             knoten2.Unterknoten.Add(knoten22);
23             wurzel.Unterknoten.Add(knoten1);
24             wurzel.Unterknoten.Add(knoten2);
```

## 10 Objektorientierung für Fortgeschrittene

```

25
26         //Baumstruktur am Bildschirmausgeben
27         Console.WriteLine(wurzel);
28     }
29 }
30 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```

Microsoft Visual Studio-Debugging-Konsole
Wurzelknoten
Knoten1
  Knoten11
  Knoten12
Knoten2
  Knoten21
  Knoten22

C:\Users\rober\source\repos\CSharpBuch\Baumstrukturen\Baumstrukturen\bin\Debug\netcoreapp3.1\Baumstrukturen.exe (Prozess "17344") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.7.3 Bildschirmausgabe der Baumstruktur

Die überschriebene Methode `ToString()` kann für jedes Objekt vom Typ `Knoten` aufgerufen werden. Damit kann können wir mit ihr auch nur einen Teil der Baumstruktur ausgeben.

Mit der Anweisung:

```
1 Console.WriteLine(knoten1);
```

Geben wir nur den ersten Unterknoten der Baumstruktur und dessen Unterknoten am Bildschirm aus.

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```

Microsoft Visual Studio-Debugging-Konsole
Knoten1
  Knoten11
  Knoten12

C:\Users\rober\source\repos\CSharpBuch\Baumstrukturen\Baumstrukturen\bin\Debug\netcoreapp3.1\Baumstrukturen.exe (Prozess "228") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.7.4 Bildschirmausgabe des ersten Unterknotens

Damit beenden wir den Einschub über das Überschreiben der `ToString()`-Methode und die Baumstrukturen und beginnen mit dem eigentlichen Thema: den generischen Klassen.

Die erste generische Klasse haben wir bereits kennengelernt. Das ist der Typ `List<einTyp>`, wobei `einTyp` ein primitiver Typ wie `string` oder ein komplexes Objekt sein kann. Beim Thema generische Klassen geht es darum, wie wir selbst eine derartige Klasse schreiben können. Statt dem Platzhalter `einTyp` werden wir in Zu-

kunft den Buchstaben T verwenden (T wie Typ). Das ist zwar nicht zwingend vorgeschrieben, aber es hat sich als Konvention eingebürgert. Damit heißt der generische Listentyp korrekt `List<T>`, wobei T als der sogenannte generische Typ-Parameter bezeichnet wird.

Im Einschub über Baumstrukturen haben wir gesehen, dass es durchaus anspruchsvoll ist, den Programmcode zu schreiben, der rekursiv durch eine Baumstruktur läuft. Ohne die Möglichkeit der generischen Klassen müssten wir für jede Baumstruktur mit unterschiedlichen Nutzdaten die Methoden, wie die `ToString()`-Methode, immer wieder neu implementieren. Mit generischen Klassen gibt es eine elegante Alternative, mit der solche Methoden nur einmal geschrieben werden müssen und für verschiedene Baumstrukturen mit verschiedenen Nutzdaten wiederverwendet werden können. Im Folgenden werden wir die generische Klasse `Knoten<T>` entwickeln.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T>
8      {
9      }
10 }
```

10

Eine generische Klasse können wir fast wie eine normale Klasse definieren. Nachdem Klassennamen schreiben wir in spitzen Klammern den Typ-Parameter T. Jetzt könnten wir die Klasse `Knoten<T>` genauso verwenden wie `List<T>`. Mit `var knoten = new Knoten<string>();` könnten wir eine Instanz von `Knoten<T>` erzeugen, wobei unsere Nutzdaten vom Typ `string` sind. Bislang haben wir aber keinerlei Funktionalität für unsere Klasse `Knoten<T>` implementiert. Als nächstes fügen wir der Klasse `Knoten<T>` eine private Membervariable und einen Konstruktor hinzu.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T>
8      {
9          private T daten;
10
11         public Knoten(T daten)
12         {
13             this.daten = daten;
14         }
15     }
```

## 10 Objektorientierung für Fortgeschrittene

16 }

Die private Membervariable heißt `daten` und verwendet als Typ den generischen Typ-Parameter `T`. Damit bekommt `daten` den Typ `string`, wenn wir ein Objekt als `Knoten<string>` erzeugen, oder den Typ `Person`, wenn wir ein Objekt als `Knoten<Person>` erzeugen. Dem Konstruktor übergeben wir einen Übergabeparameter vom Typ `T` und weisen ihn der privaten Membervariable `daten` zu. Damit wir die Klasse `Knoten<T>` als Knoten in einer Baumstruktur verwenden können, fehlt noch die Property zum Speichern der Unterknoten.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T>
8      {
9          private T daten;
10         public Knoten(T daten)
11         {
12             this.daten = daten;
13             UnterKnoten = new List<Knoten<T>>();
14         }
15
16         public List<Knoten<T>> UnterKnoten;
17     }
18 }
```

Als Property für die Unterknoten benötigen wir eine Liste von Knoten-Objekten, daher hat die Property `Unterknoten` den Typ `List<Knoten<T>>`. Des Weiteren initialisieren wir im Konstruktor die Property `Unterknoten` mit einer leeren Liste vom Typ `List<Knoten<T>>`.

Jetzt haben wir eine generische Klasse, die eine komplette Baumstruktur speichern kann. Im Hauptprogramm können wir damit schon einen Baum anlegen. Für die Nutzdaten wollen wir die bereits bekannte Klasse `Person` verwenden.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Person
8      {
9          public Person(string vorname, string nachname)
10         {
11             Vorname = vorname;
12             Nachname = nachname;
13         }
14     }
```

```
14     public string Vorname { get; set; }
15     public string Nachname { get; set; }
16
17     public override string ToString()
18     {
19         return $"{Vorname} {Nachname}";
20     }
21 }
22 }
```

```
1 using System;
2
3 namespace GenerischeKlassen
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var personWurzel = new Knoten<Person>(new
10                Person("Robert", "Schiefele"));
11             var personKnoten1 = new Knoten<Person>(new
12                Person("Max", "Mustermann"));
13             var personKnoten11 = new Knoten<Person>(new
14                Person("Theo", "Tester"));
15             var personKnoten12 = new Knoten<Person>(new
16                Person("Thea", "Testerin"));
17             var personKnoten2 = new Knoten<Person>(new
18                Person("Anna", "Müller"));
19             var personKnoten21 = new Knoten<Person>(new
20                Person("Fritz", "Fischer"));
21             var personKnoten22 = new Knoten<Person>(new
22                Person("Erna", "Müller"));
23
24             personKnoten1.UnterKnoten.Add(personKnoten11);
25             personKnoten1.UnterKnoten.Add(personKnoten12);
26
27             personKnoten2.UnterKnoten.Add(personKnoten21);
28             personKnoten2.UnterKnoten.Add(personKnoten22);
29
30             personWurzel.UnterKnoten.Add(personKnoten1);
31             personWurzel.UnterKnoten.Add(personKnoten2);
32         }
33     }
34 }
```

Ähnlich wie im Einschub über Baumstrukturen erzeugen wir zuerst Knoten-Objekte und hängen sie dann untereinander. Dem Konstruktor von `Knoten<Person>` übergeben wir ein neu erzeugtes `Person`-Objekt als anonyme Instanz. Bitte beachten Sie, die Klasse `Person` hat eine überschriebene `ToString()`-Methode, die es uns erlaubt, ein `Person`-Objekt mit `Console.WriteLine()` am Bildschirm auszugeben.

## 10 Objektorientierung für Fortgeschrittene

Als nächstes wollen wir eine überschriebene `ToString()`-Methode für die generische Klasse `Knoten<T>` implementieren, damit wir mit `Console.WriteLine()` die ganze Baumstruktur von `Person`-Objekten am Bildschirm ausgeben können.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T>
8      {
9          private T daten;
10
11         private int tiefe;
12         private string einrueckung;
13         private string baumausgabe;
14
15         public Knoten(T daten)
16         {
17             this.daten = daten;
18             UnterKnoten = new List<Knoten<T>>();
19         }
20
21         public List<Knoten<T>> UnterKnoten;
22
23         public override string ToString()
24         {
25             tiefe = 0;
26             baumausgabe = "";
27             SetzeEinrueckung();
28
29             BaumAusgabeRekursiv(this);
30             return baumausgabe;
31         }
32
33         private void BaumAusgabeRekursiv(Knoten<T> knoten)
34         {
35
36             baumausgabe += $"{einrueckung}{knoten.daten}\r\n";
37
38             if (UnterKnoten.Count > 0)
39             {
40                 tiefe++;
41                 SetzeEinrueckung();
42
43                 foreach (var unterknoten in knoten.UnterKnoten)
44                 {
45                     BaumAusgabeRekursiv(unterknoten);
46                 }
47
48                 tiefe--;
49                 SetzeEinrueckung();
50             }
51         }
52     }
```

```

52     private void SetzeEinrueckung()
53     {
54         einrueckung = "";
55         for (int i = 0; i < tiefte; i++)
56         {
57             einrueckung += "    ";
58         }
59     }
60 }
61 }
62 }
```

Die überschriebene `ToString()`-Methode der generischen Klasse `Knoten<T>` sieht genauso aus wie die überschriebene `ToString()`-Methode der nicht generischen Klasse `Knoten`. Die rekursive Methode `BaumAusgabeRekursiv()` ist bei beiden Klassen fast identisch. Bei `Knoten` ist der Übergabeparameter vom Typ `Knoten` und bei `Knoten<T>` ist er vom Typ `Knoten<T>`. Bei der Klasse `Knoten` fügt die Methode `BaumAusgabeRekursiv()` das Ergebnis des Ausdrucks `knoten.Name` und damit den Wert der String-Property `Name` in die Ausgabe ein. Und bei der Klasse `Knoten<T>` wird das Ergebnis des Ausdrucks `knoten.daten` eingefügt, welches den Typ des generischen Typ Parameters `T` hat. In unserem Beispiel wird dadurch das Ergebnis der überschriebenen `ToString()`-Methode des Objekts `Person` eingefügt. Damit kann das Hauptprogramm einen vollständigen Baum vom Typ `Knoten<Person>` am Bildschirm ausgeben.

```

1  using System;
2
3  namespace GenerischeKlassen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var personWurzel = new Knoten<Person>(new
10                 Person("Robert", "Schiefele"));
11                 var personKnoten1 = new Knoten<Person>(new
12                 Person("Max", "Mustermann"));
13                 var personKnoten11 = new Knoten<Person>(new
14                 Person("Theo", "Tester"));
15                 var personKnoten12 = new Knoten<Person>(new
16                 Person("Thea", "Testerin"));
17                 var personKnoten2 = new Knoten<Person>(new
18                 Person("Anna", "Müller"));
19                 var personKnoten21 = new Knoten<Person>(new
20                 Person("Fritz", "Fischer"));
21                 var personKnoten22 = new Knoten<Person>(new
22                 Person("Erna", "Müller"));

23                 personKnoten1.UnterKnoten.Add(personKnoten11);
24                 personKnoten1.UnterKnoten.Add(personKnoten12);

25                 personKnoten2.UnterKnoten.Add(personKnoten21);
```

## 10 Objektorientierung für Fortgeschrittene

```

28         personKnoten2.UnterKnoten.Add(personKnoten22);
29
30         personWurzel.UnterKnoten.Add(personKnoten1);
31         personWurzel.UnterKnoten.Add(personKnoten2);
32
33         Console.WriteLine(personWurzel);
34     }
35 }
36 }
```

```

Microsoft Visual Studio-Debugging-Konsole
Robert Schiefele
Max Musterman
Theo Tester
Thea Testerin
Anna Müller
Fritz Fischer
Erna Müller

C:\Users\rober\source\repos\CSharpBuch\GenerischeKlassen\GenerischeKlassen\bin\Debug\netcoreapp3.1\GenerischeKlassen.exe
(Prozess "24816") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.7.5** Bildschirmausgabe der Klasse Knoten<Person>

Damit haben wir eine generische Klasse, mit der wir Baumstrukturen erzeugen können, wobei die Nutzdaten eines Knotens eine beliebige Klasse sein können.

Beim Thema generische Klassen gibt es allerdings noch ein paar Feinheiten, die wir ergründen werden, indem wir unserer Klasse eine Finde-Methode hinzufügen.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T>
8      {
9          private T daten;
10
11         private int tiefe;
12         private string einrueckung;
13         private string baumausgabe;
14
15         public Knoten(T daten)
16         {
17             this.daten = daten;
18             UnterKnoten = new List<Knoten<T>>();
19         }
20
21         public List<Knoten<T>> UnterKnoten;
22
23         public override string ToString()
24         {
```

```
25         tiefe = 0;
26         baumausgabe = "";
27         SetzeEinrueckung();
28
29         BaumAuszgabeRekursiv(this);
30         return baumausgabe;
31     }
32
33     private void BaumAuszgabeRekursiv(Knoten<T> knoten)
34     {
35
36         baumausgabe += ${einrueckung}{knoten.daten}\r\n";
37
38         if (Unterknoten.Count > 0)
39         {
40             tiefe++;
41             SetzeEinrueckung();
42
43             foreach (var unterknoten in knoten.Unterknoten)
44             {
45                 BaumAuszgabeRekursiv(unterknoten);
46             }
47
48             tiefe--;
49             SetzeEinrueckung();
50         }
51     }
52
53     public Knoten<T> Finde(T nutzDaten)
54     {
55         return FindeRekursiv(this, nutzDaten);
56     }
57
58     public Knoten<T> FindeRekursiv(Knoten<T>knoten,
59     T nutzDaten)
60     {
61         if(knoten.daten.Equals(nutzDaten))
62         {
63             return knoten;
64         }
65         else
66         {
67             foreach(var unterknoten in knoten.Unterknoten)
68             {
69                 var ergebnis = FindeRekursiv(unterknoten,
70                 nutzDaten);
71                 if (ergebnis != null)
72                 {
73                     return ergebnis;
74                 }
75             }
76         }
77
78         return null;
79     }
```

## 10 Objektorientierung für Fortgeschrittene

```

81     private void SetzeEinrueckung()
82     {
83         einrueckung = "";
84         for (int i = 0; i < tiefte; i++)
85         {
86             einrueckung += "    ";
87         }
88     }
89 }
90 }
```

Der neuen Methode `Finde()` übergeben wir den Parameter `nutzDaten` vom Typ `T`. Die Methode ruft die rekursive Methode `FindeRekursiv()` auf und übergibt ihr den aktuellen Knoten mit dem Schlüsselwort `this` und dem Parameter `nutzDaten`. Die Methode `FindeRekursiv()` prüft zuerst, ob die Nutzdaten der Klasse `Knoten<T>` mit dem Wert des Parameters `nutzDaten` übereinstimmen.

Und dabei erleben wir bereits die erste Überraschung: Eigentlich hätten wir erwartet, dass wir den Vergleich mit dem Ausdruck `knoten.daten == nutzDaten` erledigen könnten, aber wenn wir das versuchen, erklärt uns der Compiler, dass er den Operator `==` nicht auf Operanden vom Typ `T` anwenden kann. Stattdessen müssen wir die Methode `Equals()` verwenden, die jede Klasse vom Typ `object` erbt. `Equals()` führt den Vergleich durch und gibt bei Gleichheit `true` zurück. Dieses Verhalten des Compilers werden wir später noch genauer durchleuchten.

Wenn die Nutzdaten des Knotens mit dem Parameter `nutzDaten` übereinstimmen, haben wir den gesuchten Knoten gefunden und geben ihn zurück. Wenn nicht, laufen wir mit einer Schleife über die Unterknoten und suchen dort mit der rekursiven Methode `FindeRekursiv()` in den Unterknoten. Damit durchsuchen wir rekursiv den ganzen Baum und wenn wir unseren gesuchten Knoten finden, geben wir ihn zurück, wenn wir ihn im gesamten Baum nicht finden, geben wir null zurück.

Zuerst wollen wir die neue Methode `Finde()` mit einem Baum von Typ `Knoten<string>` testen.

```

1  using System;
2
3  namespace GenerischeKlassen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var wurzel = new Knoten<string>("Wurzel");
10             var knoten1 = new Knoten<string>("Knoten1");
11             var knoten11 = new Knoten<string>("Knoten11");
12             var knoten12 = new Knoten<string>("Knoten12");
13             var knoten2 = new Knoten<string>("Knoten2");
14             var knoten21 = new Knoten<string>("Knoten21");
```

```

15     var knoten22 = new Knoten<string>("Knoten22");
16
17     knoten1.UnterKnoten.Add(knoten11);
18     knoten1.UnterKnoten.Add(knoten12);
19     knoten2.UnterKnoten.Add(knoten21);
20     knoten2.UnterKnoten.Add(knoten22);
21
22     wurzel.UnterKnoten.Add(knoten1);
23     wurzel.UnterKnoten.Add(knoten2);
24
25     Console.WriteLine(wurzel.Finde("Knoten2"));
26 }
27 }
28 }
```

Im Hauptprogramm bauen wir zuerst unseren Baum vom Typ `Knoten<string>` auf und dann suchen wir einen Knoten, dessen Nutzdaten vom Typ `string` den Wert „Knoten2“ enthalten.

```

Microsoft Visual Studio-Debugging-Konsole
Knoten2
  Knoten21
  Knoten22

C:\Users\rober\source\repos\CSharpBuch\GenerischeKlassen\GenerischeKlassen\bin\Debug\netcoreapp3.1\GenerischeKlassen.exe
(Prozess "21968") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.7.6 Der gesuchte Knoten „Knoten2“

10

Das sieht so aus, als hätten wir die Aufgabenstellung gelöst. Aber machen wir noch einen Versuch. Diesmal testen wir Methode `Finde()` mit einem Baum vom Typ `Knoten<Person>`.

```

1 using System;
2
3 namespace GenerischeKlassen
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var personWurzel = new Knoten<Person>(new
10                Person("Robert", "Schiefele"));
11             var personKnoten1 = new Knoten<Person>(new
12                Person("Max", "Mustermann"));
13             var personKnoten11 = new Knoten<Person>(new
14                Person("Theo", "Tester"));
15             var personKnoten12 = new Knoten<Person>(new
16                Person("Thea", "Testerin"));
17             var personKnoten2 = new Knoten<Person>(new
18                Person("Anna", "Müller"));
19             var personKnoten21 = new Knoten<Person>(new
20                Person("Fritz", "Fischer"));
```

## 10 Objektorientierung für Fortgeschrittene

```

21         var personKnoten22 = new Knoten<Person>(new
22             Person("Erna", "Müller"));
23
24             personKnoten1.UnterKnoten.Add(personKnoten11);
25             personKnoten1.UnterKnoten.Add(personKnoten12);
26
27             personKnoten2.UnterKnoten.Add(personKnoten21);
28             personKnoten2.UnterKnoten.Add(personKnoten22);
29
30             personWurzel.UnterKnoten.Add(personKnoten1);
31             personWurzel.UnterKnoten.Add(personKnoten2);
32
33             var ergebnis = personWurzel.Finde(new
34                 Person("Anna", "Müller"));
35             Console.WriteLine(ergebnis);
36         }
37     }
38 }
```

Diesmal erzeugen wir eine Baumstruktur mit Person-Objekten als Nutzdaten und suchen die Person Anna Müller.

The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The console output is as follows:

```
C:\Users\rober\source\repos\CSharpBuch\GenerischeKlassen\GenerischeKlassen\bin\Debug\netcoreapp3.1\GenerischeKlassen.exe
(Prozess "5192") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.7.7** Die Suche liefert kein Ergebnis.

Obwohl wir ein Person-Objekt für Anna Müller in die Baumstruktur eingefügt haben, liefert die Methode `Finde()` für Anna Müller kein Ergebnis. Egal, nach welchem Person-Objekt wir suchen, wir werden nie ein Ergebnis erhalten.

Das liegt an der Methode `Equals()`, mit der wir überprüfen, ob ein Knoten dem gesuchten Knoten entspricht. Die Methode `Equals()` wird von `object` geerbt und da wir in der Klasse `Person` die Methode `Equals()` nicht überschrieben haben, wird die Originalmethode von `object` aufgerufen. Die Methode `Equals()` von `object` kann aber nicht mit Person-Objekten umgehen. Mit String-Objekten funktioniert die Methode `Finde()`, da die Klasse `String` eine geeignete Überschreibung der Methode `Equals()` enthält. Wie können wir die Methode `Finde()` der generischen Klasse `Knoten<T>` so schreiben, dass Sie für verschiedene Nutzdaten funktioniert. Dafür gibt es verschiedene Varianten.

### Variante 1:

Wir führen den Vergleich über die von `Person` überschriebene `ToString()`-Methode durch.

```
1 knoten.daten.ToString() == nutzDaten.ToString()
```

Der Vorteil dieser Variante ist, dass sie am schnellsten implementiert ist. Der Nachteil ist, dass sie unzuverlässig ist. Nur die Properties, die in der `ToString()`-Methode verwendet werden, fließen in den Vergleich mit ein. Wenn die Klasse `Person` noch Properties für `Strasse` und `Ort` bekommt, aber in der `ToString()`-Methode nur Vorname und Nachname verwendet werden, kann es Anna Müller mehrmals mit verschiedenen Adressen im Baum geben und die Methode `Finde()` kann eine andere Anna Müller als die gesuchte liefern.

### Variante 2:

Wir überschreiben die Methode `Equals()` für die Klasse `Person`.

Der Vorteil dieser Variante ist, dass wir genau kontrollieren können, welche Daten in den Vergleich einfließen und welche nicht. Der Nachteil ist, dass wenn wir eine Klasse als Nutzdaten in einer Baumstruktur verwenden und vergessen, die `Equals()`-Methode zu überschreiben, dann funktioniert die `Finde()`-Methode nicht mehr.

### Variante 3:

Wir verwenden ein generisches Interface. Diese Variante werden wir im Folgenden näher betrachten. Der Vorteil dieser Variante ist, dass sie am zuverlässigsten funktioniert. Der Nachteil dieser Variante ist, dass sie nur bei Klassen funktioniert, die wir selbst geschrieben haben.

Definieren wir nun ein generisches Interface für eine Vergleichsmethode:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace GenerischeKlassen
6 {
7     public interface IVergleichbar<T>
8     {
9         bool IstGleich(T einObjekt);
10    }
11 }
```

Ein generisches Interface wird fast wie ein normales Interface definiert. Nach dem Namen des Interface schreiben wir den generischen Typ-Parameter `T` in spitzen Klammern. Damit können wir `T` als Typ im Interface verwenden. Das Interface `IVergleichbar` definiert die Methode `IstGleich()`. Ihr wird ein Parameter vom Typ `T` übergeben und bei Gleichheit liefert sie `true` und ansonsten `false` zurück. Als nächstes sorgen wir dafür, dass die Klasse `Person` das Interface `IVergleichbar` implementiert.

## 10 Objektorientierung für Fortgeschrittene

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Person : IVergleichbar<Person>
8      {
9          public Person(string vorname, string nachname)
10         {
11             Vorname = vorname;
12             Nachname = nachname;
13         }
14
15         public string Vorname { get; set; }
16         public string Nachname { get; set; }
17
18         public override string ToString()
19         {
20             return $"{Vorname} {Nachname}";
21         }
22
23         public bool IstGleich(Person einObjekt)
24         {
25             if(einObjekt.Vorname == Vorname &&
26             einObjekt.Nachname == Nachname)
27             {
28                 return true;
29             }
30             return false;
31         }
32     }
33 }
```

Die Klasse `Person` implementiert das Interface `IVergleichbar<Person>`. Da wir den generischen Typ Parameter als Typ `Person` festlegen, ist die zu implementierende Methode `IstGleich()` eine ganz normale Methode, die einen Parameter vom Typ `Person` entgegennimmt und einen Boole'schen Wert zurückgibt. In der Methode `IstGleich()` legen wir fest, dass zwei `Person`-Objekte gleich sind, wenn Sie in den Properties `Vorname` und `Nachname` übereinstimmen.

In der generischen Klasse `Knoten<T>` definieren wir jetzt noch eine sogenannte Typeinschränkung, damit wir die Methode `IstGleich()` in der Methode `Finde()` verwenden können.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T> where T : IVergleichbar<T>
8      {
```

```
9     private T daten;
10
11    private int tiefe;
12    private string einrueckung;
13    private string baumausgabe;
14
15    public Knoten(T daten)
16    {
17        this.daten = daten;
18        UnterKnoten = new List<Knoten<T>>();
19    }
20
21    public List<Knoten<T>> UnterKnoten;
22
23    public override string ToString()
24    {
25        tiefe = 0;
26        baumausgabe = "";
27        SetzeEinrueckung();
28
29        BaumAusbabeRekursiv(this);
30        return baumausgabe;
31    }
32
33    private void BaumAusbabeRekursiv(Knoten<T> knoten)
34    {
35
36        baumausgabe += $"{einrueckung}{knoten.daten}\r\n";
37
38        if (UnterKnoten.Count > 0)
39        {
40            tiefe++;
41            SetzeEinrueckung();
42
43            foreach (var unterknoten in knoten.UnterKnoten)
44            {
45                BaumAusbabeRekursiv(unterknoten);
46            }
47
48            tiefe--;
49            SetzeEinrueckung();
50        }
51
52
53        public Knoten<T> Finde(T nutzDaten)
54        {
55            return FindeRekursiv(this, nutzDaten);
56        }
57
58        public Knoten<T> FindeRekursiv(Knoten<T>knoten,
59        T nutzDaten)
60        {
61            if(knoten.daten.IstGleich(nutzDaten))
62            {
63                return knoten;
64            }
65        }
66    }
67
68    private void SetzeEinrueckung()
69    {
70        einrueckung = new string(' ', tiefe * 4);
71    }
72}
```

## 10 Objektorientierung für Fortgeschrittene

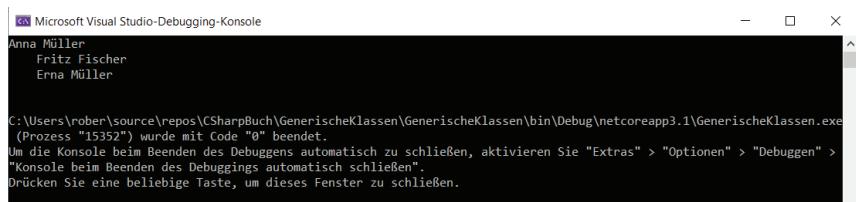
```

65         else
66         {
67             foreach(var unterknoten in knoten.UnterKnoten)
68             {
69                 var ergebnis = FindeRekursiv(unterknoten,
70                     nutzDaten);
71                 if (ergebnis != null)
72                 {
73                     return ergebnis;
74                 }
75             }
76         }
77
78         return null;
79     }
80
81     private void SetzeEinrueckung()
82     {
83         einrueckung = "";
84         for (int i = 0; i<tiefe; i++)
85         {
86             einrueckung += "    ";
87         }
88     }
89 }
90 }
```

Für die Typeinschränkung schreiben wir nach dem Namen der Klasse das Schlüsselwort `where` gefolgt vom generischen Typ-Parameter `T`, einem Doppelpunkt und dem Typen, auf den wir einschränken wollen. In unserem Fall `IVergleichbar<T>`. Damit legen wir fest, dass alle Klassen, die wir als Nutzdaten für `Knoten<T>` verwenden wollen, das generische Interface `IVergleichbar<T>` implementieren müssen.

Das heißt auch, dass alle Variablen innerhalb der Klasse `Knoten<T>`, die vom Typ `T` sind, genauso verwendet werden können, als wären Sie vom Typ `IVergleichbar<T>`. Dadurch können wir für den Vergleich in der Methode `Finde()`, den Ausdruck: `knoten.daten.IstGleich(nutzDaten)` verwenden.

Wenn wir jetzt das Programm starten, wird auch der Person-Knoten Anna Müller gefunden.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The content of the console is as follows:

```

Anna Müller
Fritz Fischer
Erna Müller

C:\Users\rober\source\repos\CSharpBuch\GenerischeKlassen\GenerischeKlassen\bin\Debug\netcoreapp3.1\GenerischeKlassen.exe
(Prozess "15352") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 10.7.8** Ausgabe der Methode `Finde()` mit Interface `IVergleichbar<T>`

Das Interface `IVergleichbar<T>` habe ich hier vorgestellt, um das Prinzip von generischen Interfaces verständlich zu machen. In der Praxis benötigen wir dieses Interface nicht, da es im .NET-Framework bereits existiert. Dort hat es keine deutsche Bezeichnung, sondern heißt `IEquatable<T>` und definiert die Methode `Equals()`. Ansonsten funktioniert es genauso wie das von uns selbst definierte Interface `IVergleichbar<T>`. Die Klasse `Knoten<T>` kann damit `IEquatable<T>` als Typeinschränkung verwenden und die Methode `Finde()` kann für den Vergleich den Ausdruck `knoten.daten.Equals(nutzDaten)` verwenden.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeKlassen
6  {
7      public class Knoten<T> where T : IEquatable<T>
8      {
9          private T daten;
10
11         private int tiefe;
12         private string einrueckung;
13         private string baumausgabe;
14
15         public Knoten(T daten)
16         {
17             this.daten = daten;
18             UnterKnoten = new List<Knoten<T>>() ;
19         }
20
21         public List<Knoten<T>> UnterKnoten;
22
23         public override string ToString()
24         {
25             tiefe = 0;
26             baumausgabe = "";
27             SetzeEinrueckung();
28
29             BaumAusbabeRekursiv(this);
30             return baumausgabe;
31         }
32
33         private void BaumAusbabeRekursiv(Knoten<T> knoten)
34         {
35
36             baumausgabe += $"{einrueckung}{knoten.daten}\r\n";
37
38             if (UnterKnoten.Count > 0)
39             {
40                 tiefe++;
41                 SetzeEinrueckung();
42
43                 foreach (var unterknoten in knoten.UnterKnoten)
44                 {
```

## 10 Objektorientierung für Fortgeschrittene

```
45             BaumAuszgabeRekursiv(unterknoten);
46         }
47
48         tiefe--;
49         SetzeEinrueckung();
50     }
51 }
52
53 public Knoten<T> Finde(T nutzDaten)
54 {
55     return FindeRekursiv(this, nutzDaten);
56 }
57
58 public Knoten<T> FindeRekursiv(Knoten<T>knoten,
59 T nutzDaten)
60 {
61     if(knoten.daten.Equals(nutzDaten))
62     {
63         return knoten;
64     }
65     else
66     {
67         foreach(var unterknoten in knoten.UnterKnoten)
68         {
69             var ergebnis = FindeRekursiv(unterknoten,
70 nutzDaten);
71             if (ergebnis != null)
72             {
73                 return ergebnis;
74             }
75         }
76     }
77
78     return null;
79 }
80
81 private void SetzeEinrueckung()
82 {
83     einrueckung = "";
84     for (int i = 0; i<tiefe; i++)
85     {
86         einrueckung += "    ";
87     }
88 }
89 }
90 }
```

Die Klasse Person kann statt IVergleichbar<T> das Interface IEquatable<T> implementieren:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace GenerischeKlassen
```

```

6  {
7      public class Person : IEquatable<Person>
8      {
9          public Person(string vorname, string nachname)
10         {
11             Vorname = vorname;
12             Nachname = nachname;
13         }
14
15         public string Vorname { get; set; }
16         public string Nachname { get; set; }
17
18         public override string ToString()
19         {
20             return $"{Vorname} {Nachname}";
21         }
22
23         public bool Equals(Person einObjekt)
24         {
25             if(einObjekt.Vorname == Vorname &&
26             einObjekt.Nachname == Nachname)
27             {
28                 return true;
29             }
30             return false;
31         }
32     }
33 }
```

Da die Klasse `String` des .NET-Frameworks ebenfalls das Interface `IEquatable<T>` implementiert, funktioniert die Methode `Finde()` jetzt auch wieder mit Nutzdaten vom Typ `String`.

```

1  using System;
2
3  namespace GenerischeKlassen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var wurzel = new Knoten<string>("Wurzel");
10             var knoten1 = new Knoten<string>("Knoten1");
11             var knoten11 = new Knoten<string>("Knoten11");
12             var knoten12 = new Knoten<string>("Knoten12");
13             var knoten2 = new Knoten<string>("Knoten2");
14             var knoten21 = new Knoten<string>("Knoten21");
15             var knoten22 = new Knoten<string>("Knoten22");
16
17             knoten1.UnterKnoten.Add(knoten11);
18             knoten1.UnterKnoten.Add(knoten12);
19             knoten2.UnterKnoten.Add(knoten21);
20             knoten2.UnterKnoten.Add(knoten22);
21
22             wurzel.UnterKnoten.Add(knoten1);
```

## 10 Objektorientierung für Fortgeschrittene

```

23             wurzel.UnterKnoten.Add(knoten2);
24
25             Console.WriteLine(wurzel.Finde("Knoten2"));
26         }
27     }
28 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```

Microsoft Visual Studio-Debugging-Konsole
Knoten2
Knoten21
Knoten22

C:\Users\rober\source\repos\CSharpBuch\GenerischeKlassen\GenerischeKlassen\bin\Debug\netcoreapp3.1\GenerischeKlassen.exe
(Prozess "16768") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.7.9** Ausgabe der Methode Finde() mit dem Interface IEquatable<T>

## 10.8 Generische Methoden

Manchmal benötigen wir keine ganze generische Klasse, sondern nur eine einzige generische Methode. C# erlaubt uns, in einer nicht-generischen Klasse eine generische Methode zu deklarieren.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace GenerischeMethoden
6  {
7      public class MeineKlasse
8      {
9          public void Vertausche<T>(ref T parameter1,
10             ref T parameter2)
11          {
12              T merke = parameter1;
13              parameter1 = parameter2;
14              parameter2 = merke;
15          }
16      }
17 }
```

Die Klasse `MeineKlasse` ist ohne generischen Typ-Parameter deklariert. Aber die Methode `Vertausche()` soll generisch sein. Daher schreiben wir nach dem Methodennamen den generischen Typ-Parameter `T` in spitzen Klammern. Generische Methoden können den generischen Typ-Parameter als Typ für Übergabeparameter, Rückgabewerte und für lokale Variablen verwenden. In unserem Beispiel benötigen wir keinen Rückgabewert. Die Methode `Vertausche<T>` nimmt zwei `ref` Parameter vom Typ `T` entgegen und tauscht mit Hilfe der lokalen Variable `merke` vom Typ `T` die Werte der beiden Übergabeparameter `parameter1` und `parameter2` aus.

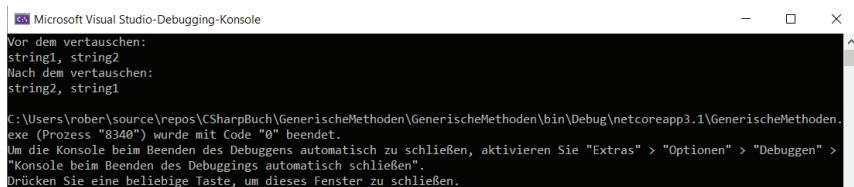
Im Hauptprogramm testen wir die generische Methode `Vertausche<T>()`.

```

1  using System;
2
3  namespace GenerischeMethoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var string1 = "string1";
10             var string2 = "string2";
11
12             Console.WriteLine("Vor dem Vertauschen:");
13             Console.WriteLine($"{string1}, {string2}");
14
15             var meineKlasse = new MeineKlasse();
16             meineKlasse.Vertausche(ref string1, ref string2);
17
18             Console.WriteLine("Nach dem Vertauschen:");
19             Console.WriteLine($"{string1}, {string2}");
20         }
21     }
22 }
```

Zuerst deklarieren wir die beiden String-Variablen `string1` und `string2` und weisen ihnen die Werte „String1“ und „String2“ zu. Danach zeigen wir die Inhalte der beiden Variablen vor dem Vertauschen am Bildschirm an. Und zwar zuerst `string1` und dann `string2`. Danach erzeugen wir eine Instanz der Klasse `MeineKlasse`, rufen die Methode `Vertausche ()` auf und übergeben ihr die Variablen `string1` und `string2` als `ref`-Parameter. Beachten Sie dabei, dass wir die Methode `Vertausche ()` nur mit ihrem Namen `Vertausche ()` rufen, ohne den generischen Typ-Parameter zu konkretisieren. Es ist zwar möglich, die Methode mit `Vertausche<string> ()` zu rufen, dies ist aber nicht nötig, da sich der Compiler den konkreten Typ des generischen Typ-Parameters vom Typ der übergebenen Parameter selbst ableiten kann.

10



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```

Microsoft Visual Studio-Debugging-Konsole
Vor dem vertauschen:
string1, string2
Nach dem vertauschen:
string2, string1

C:\Users\rober\source\repos\CSharpBuch\GenerischeMethoden\GenerischeMethoden\bin\Debug\netcoreapp3.1\GenerischeMethoden.exe (Prozess "8340") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 10.8.1 Ergebnis der Methode `Vertausche<T>()` mit Strings

Die Methode `Vertausche ()` funktioniert auch, wenn wir ihr zwei Variablen vom Typ `int` übergeben.

```

1  using System;
2
```

## 10 Objektorientierung für Fortgeschrittene

```

3  namespace GenerischeMethoden
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var integer1 = 1;
10             var integer2 = 2;
11
12             Console.WriteLine("Vor dem Vertauschen:");
13             Console.WriteLine($"{integer1}, {integer2}");
14
15             var meineKlasse = new MeineKlasse();
16             meineKlasse.Vertausche(ref integer1, ref integer2);
17
18             Console.WriteLine("Nach dem Vertauschen:");
19             Console.WriteLine($"{integer1}, {integer2}");
20         }
21     }
22 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```

Microsoft Visual Studio-Débogging-Konsole
Vor dem vertauschen:
1, 2
Nach dem vertauschen:
2, 1
C:\Users\rober\source\repos\CSharpBuch\GenerischeMethoden\GenerischeMethoden\bin\Debug\netcoreapp3.1\GenerischeMethoden.exe (Prozess "11444") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 10.8.2** Ergebnis der Methode `Vertausche<T>()` mit Int-Variablen

Wir können der Methode `Vertausche()` zwei beliebige Variablen des gleichen Typs übergeben, die Methode wird immer die Inhalte der beiden Variablen vertauschen. Ohne die Möglichkeit einer generischen Methode hätten wir die Methode `Vertausche()` für jeden benötigten Variablentyp speziell implementieren müssen.

Zum Schluss des Kapitels über generische Methoden möchte ich noch einen Spezialfall für generische Methoden vorstellen, der gelegentlich benötigt wird. Dazu betrachten wir folgende einfache generische Methode:

```

1  public static T Finde<T>(this List<T> liste, T wert)
2  {
3      if(liste.Contains(wert))
4      {
5          return wert;
6      }
7      else
8      {
9          return null;
10     }
11 }
```

Die Methode `Finde<T>()` ist eine generische Erweiterungsmethode für `List<T>`. Wenn der übergebene Parameter `wert` vom Typ `T` in der Liste enthalten ist, geben wir diesen Parameter wieder zurück, wenn nicht, geben wir `null` zurück. Da der generische Typ Parameter `T` aber jeder theoretisch mögliche Typ sein kann und damit auch ein Typ, der keine Null-Werte erlaubt, meldet uns der Compiler einen Fehler.

Für dieses Problem gibt es in C# eine einfache Lösung:

```

1 public static T Finde<T>(this List<T> liste, T wert)
2 {
3     if(liste.Contains(wert))
4     {
5         return wert;
6     }
7     else
8     {
9         return default;
10    }
11 }
```

Anstelle von `null` schreiben wir das Schlüsselwort `default`. Damit geben wir den Standardwert des generischen Typ-Parameters `T` zurück. Ist `T` eine Klasse, so wird `null` zurückgegeben. Ist `T` ein primitiver Typ, wie zum Beispiel `int`, wird dessen Standardwert, bei `int` also `0`, zurückgegeben.

10

## 10.9 Generische Properties

Nachdem wir gelernt haben, dass eine nicht generische Klasse generische Methoden haben kann, könnten wir vermuten, dass sie auch generische Properties haben kann. Das wird von C# aber nicht direkt unterstützt.

Wenn wir in einer Klasse eine oder mehrere generische Properties benötigen, müssen wir die ganze Klasse generisch deklarieren. Wenn wir generische Properties mit unterschiedlichen Typen benötigen, können wir die zugehörige generische Klasse mit zwei oder mehr generischen Typ-Parametern deklarieren. Dazu schreiben wir zwei oder mehrere generische Typ-Parameter, durch Kommata getrennt, in die spitzen Klammern nach dem Klassennamen.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace GenerischeProperties
6 {
7     public class MeineKlasse<T,K>
8     {
9         public T Parameter1 { get; set; }
10    }
```

## 10 Objektorientierung für Fortgeschrittene

```

11     public K Parameter2 { get; set; }
12     public MeineKlasse(T parameter1, K parameter2)
13     {
14         Parameter1 = parameter1;
15         Parameter2 = parameter2;
16     }
17 }
18 }
```

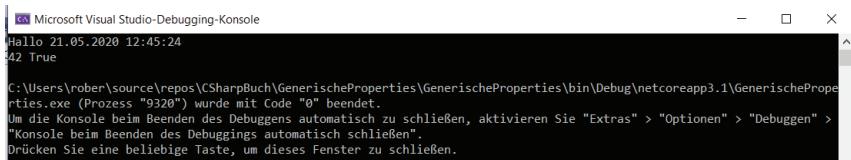
Die Klasse `MeineKlasse<T, K>` deklariert zwei generische Typ-Parameter `T` und `K`, die wir in der Klasse als Typen für die Properties `Parameter1` und `Parameter2` verwenden. Zudem besitzt die Klasse einen Konstruktor, der mit den Übergabeparametern `parameter1` und `parameter2`, mit den Typen `T` und `K` die beiden generischen Properties initialisiert.

Im Hauptprogramm testen wir die Klasse `MeineKlasse<T, K>`:

```

1 using System;
2
3 namespace GenerischeProperties
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var meineKlasse1 = new MeineKlasse<string,
10                DateTime>("Hallo", DateTime.Now);
11             var meineKlasse2 = new MeineKlasse<int,
12                bool>(42, true);
13
14             Console.WriteLine(${meineKlasse1.Parameter1}
15                ${meineKlasse1.Parameter2});
16             Console.WriteLine(${meineKlasse2.Parameter1}
17                ${meineKlasse2.Parameter2});
18         }
19     }
20 }
```

Wir erzeugen zwei Instanzen der Klasse `meine MeineKlasse<T, K>`, und legen dabei die generischen Typ-Parameter `T` und `K` auf `string` und `DateTime` für die erste Instanz und `int` und `bool` für die zweite Instanz fest. Dann geben wir die generischen Properties `Parameter1` und `Parameter2` der beiden Objekte `meineKlasse1` und `meineKlasse2` am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the following text:

```

Microsoft Visual Studio-Debugging-Konsole
Hallo 21.05.2020 12:45:24
42 True
C:\Users\rober\source\repos\CSharpBuch\GenerischeProperties\GenerischeProperties\bin\Debug\netcoreapp3.1\GenerischeProperties.exe (Prozess "9320") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 10.9.1** Parameter1 und Parameter2 von `MeineKlasse<T,K>`

## 10.10 Methoden und Variablen: Delegaten verwischen die Unterschiede

Zu Beginn dieses Buches haben wir uns mit Variablen im Allgemeinen beschäftigt und dabei die primitiven Variablentypen kennengelernt. Später haben wir uns mit strukturierten Daten in Form von Arrays und Listen beschäftigt. Und in der Objektorientierten Programmierung haben wir die Menge der Variablentypen auf Objekte mit selbstdefinierten Properties und Methoden erweitert. In diesem Kapitel werden wir mit den Delegaten einen weiteren Datentyp für Variablen kennenlernen und damit auch die Möglichkeiten, die C# zur sogenannten Funktionalen Programmierung bietet. Mit Funktionaler Programmierung bezeichnet man ein Programmierparadigma, das keinen essentiellen Unterschied zwischen Funktionen und Daten macht. Das heißt, Funktionen können wie Daten Variablen zugewiesen werden. Sie können als Übergabeparameter an andere Funktionen übergeben werden und Funktionen können auch Funktionen zurückgeben. Da C# keine reine funktionale Programmiersprache ist, sondern lediglich die Verwendung funktionaler Programmierung neben der Objektorientierten Programmierung zulässt, werde ich im Folgenden nicht den Begriff der Funktion verwenden, sondern beim objektorientierten Begriff der Methode bleiben.

Wenn wir in C# nun auch funktional programmieren wollen und dabei Variablen verwenden wollen, denen wir Funktionen beziehungsweise Methoden zuweisen wollen, benötigen wir einen Typ für diese Art von Variablen. C# ist schließlich eine typsichere Programmiersprache. So einen Variablentyp nennt man einen Delegaten. Immer, wenn wir eine Methode zur Laufzeit eines Programms für eine bestimmte Aufgabe auswählen wollen, benötigen wir einen Delegaten, damit wir so eine Methode in einer Variablen speichern, als Parameter einer Methode übergeben oder von einer Methode zurückgeben können. Dazu müssen wir zuerst den Typ für eine derartige Variable definieren.

```
1 public delegate int Kalkulation(int a, int b);
```

Für die Deklaration eines Delegaten schreiben wir nur die Signatur einer Methode, also nur den Methodenkopf, und zwischen dem Zugriffsmodifizierer und dem Rückgabetyp fügen wir das Schlüsselwort `delegate` ein. Damit haben wir einen neuen Variablentyp mit dem Namen `Kalkulation` definiert und können damit auch eine Variable deklarieren.

```
1 Kalkulation Rechne;
```

Die Variable `Rechne` hat den Typ `Kalkulation`. Wir können ihr jede Methode zuweisen, die der Signatur des Delegaten `Kalkulation` entspricht. Dies umfasst jede Methode, die einen Wert vom Typ `int` zurückgibt und zwei Übergabeparameter vom Typ `int` entgegennimmt.

```
1 private int Addiere(int a, int b)
2 {
```

## 10 Objektorientierung für Fortgeschrittene

```
3     return a + b;
4 }
```

Die Methode `Addiere()` nimmt zwei `int`-Parameter entgegen, addiert diese und gibt das Ergebnis als `int`-Wert zurück. Damit entspricht sie der Signatur des Delegaten `Kalkulation` und wir können Sie der Variablen `Rechne` zuweisen.

```
1 Rechne = Addiere;
```

Da die Variable `Rechne` jetzt eine Methode enthält, kann die Methode über den Namen der Variable genauso wie eine Methode aufgerufen werden.

```
1 var ergebnis = Rechne(2, 3);
```

Das folgende vollständige Beispielprogramm zeigt die Verwendung des Delegaten `Kalkulation`:

```
1 using System;
2
3 namespace Delegaten
4 {
5     public delegate int Kalkulation(int a, int b);
6
7     class Program
8     {
9
10        static void Main(string[] args)
11        {
12            new Program().Start();
13        }
14
15        public void Start()
16        {
17            var a = 2;
18            var b = 3;
19
20            Kalkulation Rechne;
21
22            Rechne = Addiere;
23            var summe = Rechne(a, b);
24
25            Rechne = Multipliziere;
26            var produkt = Rechne(a, b);
27
28            Console.WriteLine($"{a} + {b} = {summe}");
29            Console.WriteLine($"{a} * {b} = {produkt}");
30        }
31
32        private int Addiere(int a, int b)
33        {
34            return a + b;
35        }
36    }
```

## 10.10 Methoden und Variablen: Delegaten verwischen die Unterschiede

```

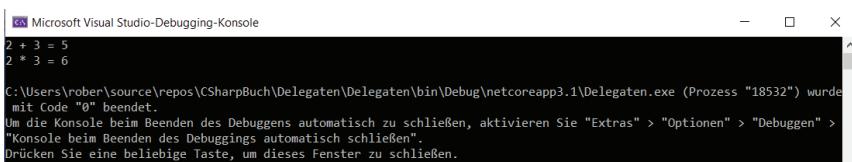
37     private int Multipliziere(int a, int b)
38     {
39         return a * b;
40     }
41 }
42 }
```

Der Delegat `Kalkulation` ist kein Bestandteil der Klasse `Program`, er ist ein eigener Typ und muss daher außerhalb der Klasse deklariert werden. Die Deklaration von Delegaten innerhalb einer Klasse wird von C# nicht unterstützt.

Die statische Hauptmethode `Main()` erzeugt eine anonyme Instanz der Klasse `Program` und ruft die Methode `Start()` auf.

Die Methode `Start()` deklariert die beiden lokalen `int`-Variablen `a` und `b` und weist Ihnen die Werte 2 und 3 zu. Danach deklariert die Methode `Start` die Variable `Rechne`. `Rechne` hat den Delegaten-Typ `Kalkulation`. Der Variablen wird die Methode `Addiere()` zugewiesen. `Addiere` entspricht der Signatur des Delegaten `Kalkulation` und addiert zwei `int`-Parameter.

Danach ruft das Programm die Methode `Addiere()` über die Variable `Rechne` auf und übergibt ihr die beiden Variablen `a` und `b` als Parameter. Das Ergebnis von `Rechne` wird in der Variablen `summe` gespeichert. Dann wird der Variablen `Rechne` die Methode `Multipliziere()` zugewiesen, die zwei übergebene `int`-Parameter miteinander multipliziert. Das Programm ruft danach über die Variable `Rechne` die Methode `Multipliziere()` auf und weist das Ergebnis der Variablen `produkt` zu. Zum Schluss werden die beiden Variablen `summe` und `produkt` am Bildschirm ausgegeben.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the command-line output of a .NET Core application named 'Delegaten.exe'. The console shows the calculation  $2 + 3 = 5$  and  $2 * 3 = 6$ . Below this, it shows the application's exit message: 'C:\Users\rober\source\repos\CSharpBuch\Delegaten\Delegaten\bin\Debug\netcoreapp3.1\Delegaten.exe (Prozess "18532") wurde mit Code "0" beendet.' It also includes instructions for closing the console automatically at exit: 'Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen". Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.'

**Abb. 10.10.1** Die Ergebnisse der Methoden „Addiere“ und „Multipliziere“

Für die Deklaration einer Variablen, der eine Methode zugewiesen werden kann, gibt es auch eine verkürzte Schreibweise als Syntactic Sugar. Wir können auf die Deklaration des Delegaten `Kalkulation` verzichten und die Variable `Rechne` wie folgt direkt deklarieren:

```
1 Func<int, int, int> Rechne;
```

Wir beginnen mit dem Schlüsselwort `Func` gefolgt von den verwendeten Typebezeichnungen in spitzen Klammern und mit Komma trennt - wobei die ersten beiden

## 10 Objektorientierung für Fortgeschrittene

Typbezeichnungen den Typen für die Übergabeparameter entsprechen und die letzte Typbezeichnung dem Rückgabewert der Methode entspricht. Damit ist `Rechne` wieder eine Variable, der jede Funktion zugewiesen werden kann, die zwei `int`-Parameter entgegen nimmt und einen `int`-Wert zurückgibt.

Der Einfachheit halber werde ich ab jetzt solche Variablen als `Func`-Variablen bezeichnen. C# kennt für `Func`-Variablen genauso wie für andere Variablen auch eine Syntax, um ihnen einen Wert mit einem Literal zuweisen zu können. Ein Literal für eine `Func`-Variable wird auch Lambda-Ausdruck genannt.

Wir können die Methode `addiere` als Lambda-Ausdruck der `Func`-Variablen `Rechne` mit folgender Syntax zuweisen.

```
1 Func<int, int, int> Rechne;
2 Rechne = (int a, int b) =>
3 {
4     return a + b;
5 };
```

Der Lambda-Ausdruck beginnt mit den beiden durch ein Komma getrennten Übergabeparametern in runden Klammern. Danach kommen die Zeichen `=>`, die Lambda-Operator genannt werden. Nach dem Lambda-Operator kommt der Methodenrumpf, der genauso aussieht, als würden wir eine normale Methode deklarieren.

Da sich der Compiler durch die Typangabe `Func<int, int, int>` der Variablen `Rechne` die Typen der Übergabeparameter selbst ableiten kann, können wir auf diese Typangaben verzichten.

```
1 Func<int, int, int> Rechne;
2 Rechne = (a, b) =>
3 {
4     return a + b;
5 };
```

Wenn der Programmcode innerhalb eines Methodenrumpfs nur aus einer Zeile besteht, können wir zusätzlich auf die geschweiften Klammern und das Schlüsselwort `return` verzichten. Da wir auch die Deklaration und die Initialisierung einer Variablen auf einmal vornehmen dürfen, können wir die Zuweisung einer Methode an eine Variable mit einem Lambda-Ausdruck sehr kompakt schreiben.

```
1 Func<int, int, int> Rechne = (a, b) => a + b;
```

Falls unsere Funktion nur einen Übergabeparameter besitzt, können wir einen Lambda-Ausdruck noch etwas kompakter schreiben und die runden Klammern um den Übergabeparameter auch noch weglassen. Eine Methode `Erhoehe`, die einen übergebenen `int`-Parameter um eins erhöht, könnte wie folgt deklariert werden:

## 10.10 Methoden und Variablen: Delegaten verwischen die Unterschiede

```
1 Func<int, int> Erhoehe = a => a++;
```

Da bei der Deklaration einer Variablen mit `Func` der letzte Typbezeichner in den spitzen Klammern immer der Rückgabetyp der Methode ist und in den spitzen Klammern mindestens ein Typbezeichner stehen muss, erlaubt diese Syntax keine Deklaration einer Variablen, der wir eine Methode mit dem Rückgabetyp `void` zuweisen können.

Aber manchmal brauchen wir in der Programmierung genau das. Daher gibt es für diesen Spezialfall in C# eine spezielle Syntax.

```
1 Action meineVoidMethode;
```

Mit dem Variablentyp `Action` können wir eine Variable deklarieren, der wir eine Methode zuweisen können, die keinen Wert entgegennimmt und auch keinen Wert zurückgibt.

Falls wir eine Variable für eine `void`-Methode mit Übertragungsparametern benötigen, können wir die Typbezeichnungen nach dem Typbezeichner `Action` in spitzen Klammern und mit Komma trennt angeben.

```
1 Action<int, string> meineVoidMethodeMitZweiParameter;
```

Der Variablen `meineVoidMethodeMitZweiParameter` können wir eine `void`-Methode mit zwei Übertragungsparametern zuweisen.

10

Zum Abschluss dieses Kapitels möchte ich unser Beispielprogramm für die Variable `Rechne` noch einmal in kompakter Schreibweise vorstellen.

```
1 using System;
2
3 namespace Delegaten
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             new Program().Start();
10        }
11
12        public void Start()
13        {
14            var a = 2;
15            var b = 3;
16
17            Func<int, int, int> Rechne = (a, b) => a + b;
18            var summe = Rechne(a, b);
19
20            Rechne = (a, b) => a * b;
```

## 10 Objektorientierung für Fortgeschrittene

```
21         var produkt = Rechne(a, b);
22
23         Console.WriteLine(${a} + {b} = {summe}");
24         Console.WriteLine(${a} * {b} = {produkt}");
25     }
26 }
27 }
```

### 10.11 Übungsaufgabe: Fortgeschrittene Programmierung mit Objekten

In dieser Übung wollen wir Schritt für Schritt ein paar Klassen entwerfen, um die Mitarbeiter und Organisationseinheiten in einem Unternehmen in einer Baumstruktur ablegen zu können. Die einzelnen Teilaufgaben bauen aufeinander auf. Versuchen Sie beim Lösen der einzelnen Teilaufgaben immer, wenn möglich, die Klassen aus vorhergehenden Teilaufgaben wieder zu verwenden.

#### Teilaufgabe 1:

Schreiben Sie die Klasse Person mit den public String-Properties Vorname und Nachname. Der Konstruktor der Klasse nimmt zwei String-Parameter entgegen und initialisiert damit die Properties Vorname und Nachname. Implementieren Sie für die Klasse Person das Interface IEquatable<T> und überschreiben Sie die von der Klasse object geerbte Methode ToString().

#### Teilaufgabe 2:

Schreiben Sie die Klasse Mitarbeiter, die von der Klasse Person abgeleitet ist. Die Klasse deklariert die public String-Property Arbeitsplatz und hat einen Konstruktor, der drei String-Parameter entgegennimmt und damit die Property Arbeitsplatz und die geerbten Properties Vorname und Nachname initialisiert. Zudem implementiert die Klasse Mitarbeiter ebenfalls das Interface IEquatable<T> und überschreibt die von Person geerbte Methode ToString().

#### Teilaufgabe 3:

Schreiben Sie die Klasse OrgEinheit. Die Klasse fungiert als Knotenklasse für eine Baumstruktur, da jede Organisationseinheit wieder Untereinheiten haben kann. Die Untereinheiten werden in der Property Untereinheiten vom Typ List<OrgEinheit> gespeichert. Die Nutzdaten eines Knotens bestehen aus der String Property Name, die den Namen der Organisationseinheit enthält, und der Property OeLeitung vom Typ Mitarbeiter, die den Mitarbeiter enthält, der die Einheit leitet. Weitere Nutzdaten sind in der Property Kollegen vom Typ List<Mitarbeiter> enthalten. Dort werden alle Mitarbeiter gespeichert, die der Organisationseinheit zugeordnet sind.

Der Konstruktor der Klasse nimmt einen `String`-Parameter entgegen, der den Namen der Organisationseinheit enthält und initialisiert damit die Property `Name`. Zudem initialisiert der Konstruktor die Properties `Kollegen` und `Untereinheiten` mit einer leeren Liste.

#### Teilaufgabe 4:

Erweitern Sie die Klasse `Mitarbeiter`. Fügen Sie ihr die Property `Organisationseinheit` mit dem Typ `OrgEinheit` hinzu.

Erweitern Sie die Klasse `OrgEinheit`. Schreiben sie die `public`-Methode `NeuerMitarbeiter()`. Die Methode `NeuerMitarbeiter()` nimmt einen Parameter vom Typ `Mitarbeiter` entgegen.

Sie fügt den neuen Mitarbeiter an die Listen-Property `Kollegen` an und weist der Property `Organisationseinheit` der übergebenen Mitarbeiter-Klasse die aktuelle Instanz von `OrgEinheit` zu.

#### Teilaufgabe 5:

Erweitern Sie die Klasse `Organisationseinheit`. Schreiben Sie die `public`-Methode `ErstelleListeAllerMitarbeiter()`. Die Methode erstellt eine Liste aller Mitarbeiter der Organisationseinheit. Die Liste soll auch die Mitarbeiter aller Untereinheiten sowie die Mitarbeiter deren Untereinheiten und so weiter enthalten. Der Rückgabewert der Methode ist vom Typ `List<Mitarbeiter>`. Schreiben Sie, wenn nötig, eine zusätzliche private Hilfsmethode.

10

#### Teilaufgabe 6:

Erweitern Sie die Klasse `OrgEinheit`. Schreiben Sie die generische Methode mit der folgenden Signatur:

```
1 public T SucheOrgEinheitAls<T>(Func<OrgEinheit, T> erzeuge,
2 Func<OrgEinheit, bool> wenn)
```

Die Methode soll die Organisationseinheit rekursiv durchsuchen und zur Überprüfung eines Treffers die übergebene Methode `Func<OrgEinheit, bool> wenn()` aufrufen. Dabei wird der Methode `wenn()` die aktuelle Organisationseinheit übergeben. Sobald der erste Treffer gefunden wird, ist die Rekursion beendet und es wird die übergebene Methode `Func<OrgEinheit, T> erzeuge` aufgerufen. Der Methode `erzeuge` wird die aktuelle Organisationseinheit übergeben. Das Ergebnis der Methode `erzeuge` ist vom Typ `T` und wird dann als Ergebnis der Methode `SucheOrgEinheitAls<T>` zurückgegeben.

**Teilaufgabe 7:**

Schreiben Sie ein Hauptprogramm, das die bisher erstellten Klassen testet. Erzeugen Sie einige Instanzen der Klasse `OrgEinheit`. Erzeugen Sie Instanzen der Klasse `Mitarbeiter`. Weisen Sie Mitarbeiter den Organisationseinheiten zu und hängen Sie die Instanzen der Klasse `OrgEinheit` untereinander, damit eine Baumstruktur entsteht. Geben Sie eine Liste aller Mitarbeiter der Organisation am Bildschirm aus. Verwenden Sie Methode `SucheOrgEinheitAls<T>()`, um den Vorgesetzten eines Mitarbeiters zu finden und geben Sie dann den Vorgesetzten am Bildschirm aus.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics.CodeAnalysis;
4  using System.Text;
5
6  namespace Uebung_10_11
7  {
8      public class Person : IEquatable<Person>
9      {
10         public string Vorname { get; set; }
11         public string Nachname { get; set; }
12
13         public Person (string vorname, string nachname)
14         {
15             Vorname = vorname;
16             Nachname = nachname;
17         }
18
19         public bool Equals(Person other)
20         {
21             return Vorname == other.Vorname && Nachname ==
22                   other.Nachname;
23         }
24
25         public override string ToString()
26         {
27             return $"{Vorname} {Nachname}";
28         }
29     }
30 }
```

## 10 Objektorientierung für Fortgeschrittene

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics.CodeAnalysis;
4  using System.Text;
5
6  namespace Uebung_10_11
7  {
8      public class Mitarbeiter : Person, IEquatable<Mitarbeiter>
9      {
10         public string Arbeitsplatz { get; set; }
11
12         public Mitarbeiter(string vorname, string nachname,
13                             string arbeitsplatz) : base(vorname, nachname)
14         {
15             Arbeitsplatz = arbeitsplatz;
16         }
17
18         public bool Equals(Mitarbeiter other)
19         {
20             return
21                 Arbeitsplatz == other.Arbeitsplatz &&
22                 Vorname == other.Vorname &&
23                 Nachname == other.Nachname;
24         }
25
26         public override string ToString()
27         {
28             return $"{base.ToString()} {Arbeitsplatz}";
29         }
30     }
31 }
```

**Musterlösung für Teilaufgabe 3:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Uebung_10_11
6  {
7      public class OrgEinheit
8      {
9          public string Name { get; set; }
10         public Mitarbeiter OeLeitung { get; set; }
11         public List<Mitarbeiter> Kollegen {get; set;}
12
13         public List<OrgEinheit> Untereinheiten { get; set; }
14
15         public OrgEinheit(string name)
16         {
17             Name = name;
18             Kollegen = new List<Mitarbeiter>();
19             Untereinheiten = new List<OrgEinheit>();
20         }
21
22     }
23 }
```

## 10 Objektorientierung für Fortgeschrittene

**Musterlösung für Teilaufgabe 4:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics.CodeAnalysis;
4  using System.Text;
5
6  namespace Uebung_10_11
7  {
8      public class Mitarbeiter : Person, IEquatable<Mitarbeiter>
9      {
10         public string Arbeitsplatz { get; set; }
11
12         public OrgEinheit OrganisationsEinheit;
13
14         public Mitarbeiter(string vorname, string nachname,
15                            string arbeitsplatz) : base(vorname, nachname)
16         {
17             Arbeitsplatz = arbeitsplatz;
18         }
19
20         public bool Equals(Mitarbeiter other)
21         {
22             return
23                 Arbeitsplatz == other.Arbeitsplatz &&
24                 Vorname == other.Vorname &&
25                 Nachname == other.Nachname;
26         }
27
28         public override string ToString()
29         {
30             return $"{base.ToString()} {Arbeitsplatz}";
31         }
32     }
33 }
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Uebung_10_11
6  {
7      public class OrgEinheit
8      {
9          public string Name { get; set; }
10         public Mitarbeiter OeLeitung { get; set; }
11         public List<Mitarbeiter> Kollegen { get; set; }
12
13         public List<OrgEinheit> Untereinheiten { get; set; }
14
15         public OrgEinheit(string name)
16         {
17             Name = name;
18             Kollegen = new List<Mitarbeiter>();
19             Untereinheiten = new List<OrgEinheit>();
```

## 10.11 Übungsaufgabe: Fortgeschrittene Programmierung mit Objekten

```
20     }
21
22     public void NeuerMitarbeiter(Mitarbeiter mitarbeiter)
23     {
24         mitarbeiter.OrganisationsEinheit = this;
25         Kollegen.Add(mitarbeiter);
26     }
27 }
28 }
```

## 10 Objektorientierung für Fortgeschrittene

**Musterlösung Teilaufgabe 5:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Uebung_10_11
6  {
7      public class OrgEinheit
8      {
9          public string Name { get; set; }
10         public Mitarbeiter OeLeitung { get; set; }
11         public List<Mitarbeiter> Kollegen {get; set;}
12
13         public List<OrgEinheit> Untereinheiten { get; set; }
14
15         public OrgEinheit(string name)
16         {
17             Name = name;
18             Kollegen = new List<Mitarbeiter>();
19             Untereinheiten = new List<OrgEinheit>();
20         }
21
22         public void NeuerMitarbeiter(Mitarbeiter mitarbeiter)
23         {
24             mitarbeiter.OrganisationsEinheit = this;
25             Kollegen.Add(mitarbeiter);
26         }
27
28         public List<Mitarbeiter> ErstelleListeAllerMitarbeiter()
29         {
30             return ErstelleListeAllerMitarbeiterRekursiv(this);
31         }
32
33         private List<Mitarbeiter>
34         ErstelleListeAllerMitarbeiterRekursiv(OrgEinheit
35         orgseinheit)
36         {
37             var mitarbeiterListe = new List<Mitarbeiter>();
38             mitarbeiterListe.AddRange(orgseinheit.Kollegen);
39
40             foreach (var untereinheit in
41             orgseinheit.Untereinheiten)
42             {
43                 mitarbeiterListe.
44                 AddRange(ErstelleListeAllerMitarbeiterRekursiv
45                 (untereinheit));
46             }
47
48             return mitarbeiterListe;
49         }
50     }
51 }
```

**Musterlösung Teilaufgabe 6:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Uebung_10_11
6  {
7      public class OrgEinheit
8      {
9          public string Name { get; set; }
10         public Mitarbeiter OeLeitung { get; set; }
11         public List<Mitarbeiter> Kollegen { get; set; }
12
13         public List<OrgEinheit> Untereinheiten { get; set; }
14
15         public OrgEinheit(string name)
16         {
17             Name = name;
18             Kollegen = new List<Mitarbeiter>();
19             Untereinheiten = new List<OrgEinheit>();
20         }
21
22         public void NeuerMitarbeiter(Mitarbeiter mitarbeiter)
23         {
24             mitarbeiter.OrganisationsEinheit = this;
25             Kollegen.Add(mitarbeiter);
26         }
27
28         public List<Mitarbeiter> ErstelleListeAllerMitarbeiter()
29         {
30             return ErstelleListeAllerMitarbeiterRekursiv(this);
31         }
32
33         private List<Mitarbeiter>
34         ErstelleListeAllerMitarbeiterRekursiv(OrgEinheit
35         orgeinheit)
36         {
37             var mitarbeiterListe = new List<Mitarbeiter>();
38             mitarbeiterListe.AddRange(orgeinheit.Kollegen);
39
40             foreach (var untereinheit in
41             orgeinheit.Untereinheiten)
42             {
43                 mitarbeiterListe.
44                 AddRange(ErstelleListeAllerMitarbeiterRekursiv
45                 (untereinheit));
46             }
47
48             return mitarbeiterListe;
49         }
50
51         public T SucheOrgEinheitAls<T>(Func<OrgEinheit,T>
52         erzeuge, Func<OrgEinheit, bool> wenn)
53         {
54             return SucheOrgEinheitAlsRekursiv(this,
```

## 10 Objektorientierung für Fortgeschrittene

```
55         erzeuge, wenn);
56     }
57
58     private T SucheOrgEinheitAlsRekursiv<T>(OrgEinheit
59     orgseinheit, Func<OrgEinheit, T> erzeuge,
60     Func<OrgEinheit,
61     bool> wenn)
62     {
63         if (wenn(orgseinheit))
64         {
65             return erzeuge(orgseinheit);
66         }
67         else
68         {
69             foreach (var untereinheit in orgseinheit.
70             Untereinheiten)
71             {
72                 var ergebnis =
73                 SucheOrgEinheitAlsRekursiv(untereinheit,
74                 erzeuge, wenn);
75                 if (ergebnis != null)
76                 {
77                     return ergebnis;
78                 }
79             }
80         }
81
82         return default;
83     }
84 }
85 }
```

**Musterlösung für Teilaufgabe 7:**

```
1  using System;
2
3  namespace Uebung_10_11
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var oeWurzel = new OrgEinheit("OrgEinheit1");
10             oeWurzel.OeLeitung = new Mitarbeiter("Leiter",
11                 "OE1", "Chefbüro OE1");
12             oeWurzel.NeuerMitarbeiter(new Mitarbeiter("MA1",
13                 "OE1", "Bürol OE1"));
14             oeWurzel.NeuerMitarbeiter(new Mitarbeiter("MA2",
15                 "OE1", "Büro2 OE1"));
16
17             var oe11 = new OrgEinheit("OrgEinheit11");
18             oe11.OeLeitung = new Mitarbeiter("Leiter", "OE11",
19                 "Chefbüro OE11");
20             oe11.NeuerMitarbeiter(new Mitarbeiter("MA1", "OE11",
21                 "Bürol OE11"));
22             oe11.NeuerMitarbeiter(new Mitarbeiter("MA2", "OE11",
23                 "Büro2 OE11"));
24
25             var oe12 = new OrgEinheit("OrgEinheit12");
26             oe12.OeLeitung = new Mitarbeiter("Leiter", "OE12",
27                 "Chefbüro OE12");
28             oe12.NeuerMitarbeiter(new Mitarbeiter("MA1", "OE12",
29                 "Bürol OE12"));
30             oe12.NeuerMitarbeiter(new Mitarbeiter("MA2", "OE12",
31                 "Büro2 OE12"));
32
33             oeWurzel.Untereinheiten.Add(oe11);
34             oeWurzel.Untereinheiten.Add(oe12);
35
36             var mitarbeiterListe = oeWurzel.
37             ErstelleListeAllerMitarbeiter();
38
39             foreach (var mitarbeiter in mitarbeiterListe)
40             {
41                 Console.WriteLine($"{mitarbeiter} ({mitarbeiter.
42                     OrganisationsEinheit.Name})");
43             }
44
45             var ergebnis = oeWurzel.SucheOrgEinheitAls(
46                 oe => oe.OeLeitung,
47                 oe => oe.Kollegen.Contains(new Mitarbeiter("MA1",
48                     "OE12", "Bürol OE12")));
49
50             Console.WriteLine(ergebnis);
51         }
52     }
53 }
```

## 10 Objektorientierung für Fortgeschrittene



```
Microsoft Visual Studio-Debugging-Konsole

MA1 OE1 Büro1 OE1 (OrgEinheit1)
MA2 OE1 Büro2 OE1 (OrgEinheit1)
MA1 OE11 Büro1 OE11 (OrgEinheit11)
MA2 OE11 Büro2 OE11 (OrgEinheit11)
MA1 OE12 Büro1 OE12 (OrgEinheit12)
MA2 OE12 Büro2 OE12 (OrgEinheit12)
Leiter OE12 Chefbüro OE12

C:\Users\rober\source\repos\CSharpBuch\Uebung_10_11\Uebung_10_11\bin\Debug\netcoreapp3.1\Uebung_10_11.exe (Prozess "2471
6") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 10.11.1 Bildschirmausgabe der Musterlösung für Teilaufgabe 7

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 11

# Objekte verarbeiten mit „Linq to Objects“

Bis jetzt haben wir uns auf die Programmiersprache C# an sich konzentriert und sind auf Klassenbibliotheken des .NET-Frameworks nur sporadisch eingegangen. Hier werden wir zum ersten Mal eine .NET-Klassenbibliothek näher betrachten: Die Bibliothek Linq. Mit Linq können die Daten beliebiger Objektstrukturen abgefragt werden.

### 11.1 Was ist Linq?

Linq steht für Language Integrated Query und wurde zum ersten Mal mit dem .NET-Framework 3.5 zusammen mit Visual Studio 2008 eingeführt. Ein großer Teil aller Geschäftsanwendungen speichert seine Daten in relationalen Datenbanken, wie zum Beispiel MS SQL-Server oder Oracle.

Relationale Datenbanken können aber über höhere Programmiersprachen wie C# oder VB.NET nicht direkt abgefragt werden. Datenbanken können aber mit einer eigenen Abfragesprache, der Structured Query Language, oder kurz SQL, abgefragt werden. Wenn nun eine Applikation in C# Daten aus einer Datenbank abrufen möchte, muss sie den Umweg über SQL gehen, das heißt, der Programmierer muss eine Abfrage, die beschreibt, welche Daten er von der Datenbank haben möchte, in SQL formulieren und wie einen gewöhnlichen Text in einer Zeichenkettenvariable speichern. Diese SQL-Abfrage schickt das Programm an die Datenbank und, wenn alles gut geht, dann erhält das Programm ein Ergebnis in einer Datenstruktur.

Da aber der C#-Compiler keine Ahnung von SQL hat, kann der Programmierer jeden beliebigen Text an die Datenbank schicken. Wenn der Text nicht der Syntax von SQL entspricht, bricht das Programm zur Laufzeit mit einem Fehler ab. Wäre es da nicht besser, wenn der Compiler oder - noch besser - die Entwicklungsumgebung eine falsche SQL-Syntax erkennen und den Programmierer darauf aufmerksam machen würde? So könnte er seinen Fehler sofort korrigieren und müsste nicht erst das Programm kompilieren, starten und den Teil ausprobieren, der die SQL-Abfrage durchführt, um zu wissen, ob er seine Abfrage korrekt formuliert hat.

Mit Linq hat Microsoft einen allgemeinen Lösungsansatz für dieses Problem geschaffen. Damit kann man nicht nur auf Datenbanken, sondern auch auf XML-Daten, SharePoint-Daten und beliebige Objekt-Strukturen zugreifen. Linq-Ausdrücke werden über sogenannte Linq-Provider in die Datenzugriffsmethoden der jeweiligen Datenquelle übersetzt. Das .NET-Framework stellt folgende Linq-Provider zur Verfügung:

„LINQ to Objects“ für den Zugriff auf Objektlisten im Hauptspeicher

„LINQ to SQL“ für die Abfrage und Verarbeitung von Daten des MS SQL-Servers

„LINQ to Entities“ zur Abfrage von Daten aus relationalen Datenbanken (SQL-Server, Oracle etc.)

„LINQ to XML“ zum Verarbeiten von XML-Daten

„LINQ to DataSet“ für den Zugriff auf ADO.NET-DataSets und -Tabellen

„LINQ to SharePoint“ um auf SharePoint-Daten zuzugreifen

Das Herz eines Linq-Providers ist ein generisches Interface. Bei LINQ to Objects ist das das Interface `IEnumerable<T>` und bei LINQ to Entities das Interface `IQueryable<T>`. Damit nun die üblichen Objekt-Strukturen, wie `List<T>`, Arrays, Dictionaries etc., die in bestehenden .NET-Anwendungen häufig verwendet wurden, von LINQ profitieren können, hat Microsoft ab dem .NET-Framework 3.5 das Interface `IEnumerable<T>` in diesen Strukturen implementiert.

Im Folgenden betrachten wir den Linq-Provider „LINQ to Objects“ etwas näher. Er stellt Erweiterungsmethoden für das Interface `IEnumerable<T>` zur Verfügung. Die Erweiterungsmethode „Where“ ist die in Linq am häufigsten benutzte Methode. Linq-Abfragen werden nativ in C# erstellt, und bereits beim Eingeben des Quellcodes im Editor werden eventuelle Syntaxfehler erkannt und angezeigt.

In C# können Linq-Abfragen in der Abfrage- und in der Methodensyntax formuliert werden. Die Abfragesyntax wird vom Compiler in Aufrufe der Methodensyntax übersetzt. Da die Abfragesyntax formal eine gewisse Ähnlichkeit mit SQL besitzt, ist sie für Programmierer, die SQL beherrschen, leichter zu lesen. In professionellen Projekten dagegen ist es heute „State of the Art“, Linq-Abfragen direkt in der Methodensyntax zu schreiben.

## 11.2 Ein erstes Beispiel für eine Linq-Abfrage

Für ein erstes Linq-Beispiel betrachten wir einmal folgende einfache Liste:

```
1 var Woche = new List<string> { "Montag", "Dienstag",
2   "Mittwoch", "Donnerstag", "Freitag" };
```

Die Variable `Wochentage` enthält eine Liste mit den Namen der Wochentage von Montag bis Freitag. Wenn wir jetzt daraus eine Liste mit allen Wochentagen, die mit dem Buchstaben M beginnen, erstellen wollen, könnte der Quellcode dazu folgendermaßen aussehen:

## 11 Objekte verarbeiten mit „Linq to Objects“

```
1 var TageMitM = new List<string>();  
2  
3 foreach(var tag in Woche)  
4 {  
5     if (tag.StartsWith("M"))  
6     {  
7         TageMitM.Add(tag);  
8     }  
9 }
```

Mit der Abfragesyntax von Linq lässt sich das kompakter und eleganter formulieren:

```
1 var TageMitM = from tag in Woche where tag.  
2 StartsWith("M") select tag;
```

Für Leser, die mit SQL vertraut sind, sieht das ziemlich bekannt aus. Der Variablen `TageMitM` wird eine Linq-Abfrage zugewiesen. Die Abfrage startet mit dem Schlüsselwort `from`, dann folgt die Laufvariable, deren Namen frei wählbar ist. In unserem Beispiel heißt sie `tag`. Danach folgt das Schlüsselwort `in` und die Listenstruktur, die wir abfragen wollen, in unserem Beispiel die Variable `woche` vom Typ `List<string>`. Da die Variable `Wocche` vom Typ `List<string>` ist, ist die Laufvariable `tag` vom Typ `string`.

Als nächstes folgt eine Bedingung, die definiert, welche Elemente der Liste zur Ergebnismenge der Abfrage gehören und welche nicht. Die Bedingung beginnt mit dem Schlüsselwort `where`, gefolgt von einem Boole'schen Ausdruck, der für jedes Element der Liste ausgewertet wird. Nur wenn der Ausdruck wahr ist, gehört das Element zum Ergebnis. Sinnvollerweise bezieht sich der Boole'sche Ausdruck auf das aktuelle Listenelement. In unserem Beispiel können wir das erreichen, indem wir im Ausdruck die Laufvariable `tag` verwenden. Der Ausdruck `tag.StartsWith("M")` wählt alle Elemente der Liste aus, die mit dem Buchstaben „M“ beginnen. Am Ende der Abfrage steht der `select`-Ausdruck. Damit wird festgelegt, wie die Elemente der Ergebnisliste aussehen. Nach dem Schlüsselwort `select` folgt ein Ausdruck, der ein Element der Ergebnisliste bestimmt. Dabei kann auch die Laufvariable `tag` verwendet werden. In unserem Beispiel wollen wir die Laufvariable `tag` als Element der Ergebnisliste verwenden, daher heißt der `select`-Ausdruck: `select tag`. Eine weitere Anwendungsmöglichkeit für den `select`-Ausdruck lernen wir im Kapitel „Daten konvertieren mit Linq“ kennen.

Die Abfragesyntax von Linq hat in der Praxis inzwischen sehr stark an Bedeutung verloren. Meistens wird Linq in der sogenannten Methodensyntax eingesetzt. Da das Behandeln beider Syntaxen für alle Features von Linq den Rahmen dieses Buches sprengen würde, werden wir ab jetzt nur noch die Methodensyntax verwenden.

Mit der Methodensyntax geht es noch kompakter und eleganter:

## 11.2 Ein erstes Beispiel für eine Linq-Abfrage

```
1 var tageMitM = Woche.Where(t => t.StartsWith("M"));
```

Hier weisen wir der Variablen `TageMitM` das Ergebnis der Methode `Where` des Objekts `Wöche` zu. Die Variable `Wöche` ist vom Typ `List<string>`, also einem Spezialfall von `List<T>` und da `List<T>` das Interface `IEnumerable<T>` implementiert, können wir für `Wöche` die Erweiterungsmethoden für `IEnumerable<T>` verwenden. Dazu müssen wir nur die Linq-Bibliothek mit der Anweisung

```
1 using System.Linq;
```

einbinden. Die Methode `Where()` ist eine Erweiterungsmethode für `IEnumerable<T>` und nimmt einen Übergabeparameter vom Typ `Func<T, bool>` entgegen. Das heißt, wenn wir `Where()` für eine Variable vom Typ `List<string>` aufrufen, erwartet `Where()` einen Übergabeparameter vom Typ `Func<string, bool>` also eine Methode, die einen Wert vom Typ `bool` zurückgibt und einen String-Parameter entgegennimmt. Genauso so eine Methode liefern wir mit dem Lambda-Ausdruck:

```
1 t => t.StartsWith("M")
```

Die Methode `Where()` führt die via Lambda-Ausdruck übergebene Methode für jedes Element der Liste aus und gibt eine Liste zurück, die alle Elemente enthält, für die die übergebene Methode den Boole'schen Wert `true` zurückgibt.

Wenn wir den Mauszeiger auf die Variable `tageMitM` bewegen, sehen wir im Tooltip, dass `tageMitM` den Typ `IEnumerable<string>` hat. Diesen Typ können wir mit einer `foreach`-Schleife genauso durchlaufen wie `List<string>`.

11

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4
5 namespace ErstesLinqBeispiel
6 {
7     class Program
8     {
9         static void Main(string[] args)
10        {
11            var Woche = new List<string> { "Montag", "Dienstag",
12                                         "Mittwoch", "Donnerstag", "Freitag" };
13
14            var tageMitM = Woche.Where(t => t.StartsWith("M"));
15
16            foreach(var tag in tageMitM)
17            {
18                Console.WriteLine(tag);
19            }
20        }
21    }
22 }
```

## 11 Objekte verarbeiten mit „Linq to Objects“

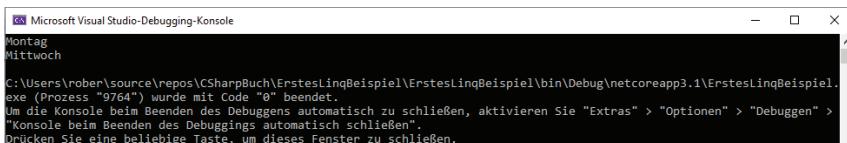


Abb. 11.2.1 Bildschirmausgabe der Wochentage mit M

### 11.3 Daten abfragen mit Linq

In diesem Kapitel werden wir einige der Erweiterungsmethoden von `IEnumerable<T>` kennenlernen. Ein vollständiger Überblick über alle Methoden würde den Rahmen dieses Buchs sprengen, daher werden wir uns auf die wichtigsten und in der Praxis am häufigsten benötigten Methoden beschränken.

Als Datenbasis für unsere Linq-Abfragen werden wir die Klasse `Person` verwenden, die uns schon öfter gute Dienste geleistet hat.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace LinqAbfragen
6  {
7      public class Person
8      {
9          public Person(string vorname, string nachname)
10         {
11             Vorname = vorname;
12             Nachname = nachname;
13         }
14
15         public string Vorname { get; set; }
16         public string Nachname { get; set; }
17
18         public override string ToString()
19         {
20             return $"{Vorname} {Nachname}";
21         }
22
23         public static List<Person> AllePersonen
24         {
25             get
26             {
27                 var personen = new List<Person>();
28
29                 personen.Add(new Person("Konradi", "Zuse"));
30                 personen.Add(new Person("Grace", "Hopper"));
31                 personen.Add(new Person("Alan", "Turing"));
32                 personen.Add(new Person("Ada", "Lovalce"));
33             }
34         }
35     }
36 }
```

```

33     personen.Add(new Person("Charles", "Babbage"));
34     personen.Add(new Person("Rózsa", "Péter"));
35     personen.Add(new Person("Herrman", "Holerith"));
36     personen.Add(new Person("Hedy", "Lamarr"));
37     personen.Add(new Person("Edgar F.", "Codd"));
38     personen.Add(new Person("Marilyn", "Meltzer"));
39     personen.Add(new Person("Heinz", "Nixdorf"));
40     personen.Add(new Person("Gertrude", "Blanch"));

41     return personen;
42 }
43 }
44 }
45 }
46 }
47 }
```

Unsere Klasse Person hat zudem die statische Property AllePersonen, die uns eine Struktur vom Typ `List<Person>` zurückliefert, die bereits einige Instanzen der Klasse Person enthält. Bei den Namen handelt es sich um Personen, die sich in besonderem Maße um die Informatik verdient gemacht haben.

Die erste Erweiterungsmethode von `IEnumerable<T>`, die ich Ihnen vorstellen möchte, ist die Methode:

```
1 T IEnumerable<T>.First<T>()
```

Für den Typ Person:

```
1 Person IEnumerable<Person>.First<Person>()
```

11

Sie liefert das erste Element einer Listenstruktur zurück, vorausgesetzt die Listenstruktur implementiert das Interface `IEnumerable<T>`. Zudem gibt es eine weitere Überladung der Methode `First()`:

```
1 T IEnumerable<T>.First<T>(Func<T, bool> predicate)
```

Für den Typ Person:

```
1 Person IEnumerable<Person>.First<Person>(Func<
2 <Person, bool> predicate)
```

Diese Überladung nimmt eine Methode entgegen, die den generischen Typ-Parameter `T` als Übergabewert erwartet und einen Bool'schen Wert zurückgibt. Solche Methoden werden allgemein als Predicates bezeichnet.

Die Methode `predicate` wird solange auf die Elemente der Listenstruktur angewendet, bis sie den Wert `true` zurückgibt. Damit wird das erste Element zurückgeliefert, bei dem die Methode `predicate` den Wert `true` zurück liefert.

## 11 Objekte verarbeiten mit „Linq to Objects“

Im folgenden Beispiel wenden wir die beiden Überladungen der Methode `First` auf unsere Datenbasis vom Typ `List<Person>` an.

```

1  using System;
2  using System.Linq;
3
4  namespace LinqAbfragen
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              Console.WriteLine(Person.AllePersonen.First());
11              Console.WriteLine(Person.AllePersonen.First(p =>
12                  p.Vorname.StartsWith("A")));
13          }
14      }
15 }
```

Microsoft Visual Studio-Debugging-Konsole

Konrad Zuse  
Adri Lovlace

C:\Users\rober\source\repos\CSharpBuch\LinqAbfragen\LinqAbfragen\bin\Debug\netcoreapp3.1\LinqAbfragen.exe (Prozess "1240") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

**Abb. 11.3.1** Ergebnisse der beiden Überladungen der Methode `First`

**Achtung:** Wenn Sie die Methode `First()` verwenden sollten in Ihrem Programm sichergestellt werden, dass die Methode `First()` auch ein Element in ihrer Datenbasis findet. Wenn die Methode `First()` kein Element findet, bricht ihr Programm mit einem Fehler ab.

Alternativ können Sie die Methode `FirstOrDefault()` anstelle der Methode `First()` verwenden. Die Methode `FirstOrDefault()` gibt den Standardwert der Klasse zurück, auf der die verwendete Listenstruktur basiert, wenn die Methode kein Element findet. Wenn Sie zum Beispiel eine Struktur vom Typ `IEnumerable<Person>` verwenden, gibt die Methode `FirstOrDefault()` den Wert `null` zurück. Verwenden Sie `IEnumerable<int>`, gibt sie `FirstOrDefault()` den Wert `0` zurück.

Die nächste der Methoden, die wir betrachten, ist die Methode

```
1  bool IEnumerable<T>.Any<>()
```

beziehungsweise

```
1  bool IEnumerable<Person>.Any<Person>()
```

(Wenn wir sie auf Strukturen anwenden, die auf dem Typ Person basieren.).

Die Methode Any() liefert einen Boole'schen Wert zurück der true ist, wenn die Struktur, auf die sie angewendet wird, Elemente enthält und false wenn nicht. Linq bietet auch die folgende Überladung für die Methode Any():

```
1 bool IEnumerable<T>.Any<T>(Func<T, bool> predicate)
```

beziehungsweise

```
1 bool IEnumerable<Person>.Any<Person>(Func<Person,
2 bool> predicate)
```

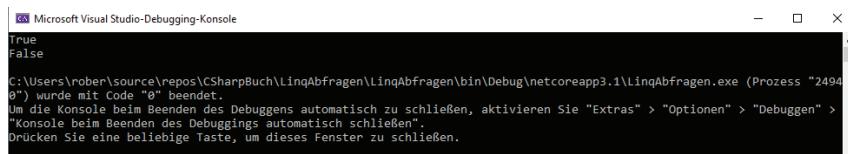
(Für Strukturen, die auf die Klasse Person basieren.).

Mit dieser Überladung liefert die Methode Any() nur dann true zurück, wenn die Struktur auf die Any() angewendet wird, Elemente enthält, für die die übergebene Methode predicate den Wert true ergibt.

Als nächstes wenden wir beide Überladungen der Methode Any() auf unsere Datenbasis vom Typ List<Person> an:

```
1 using System;
2 using System.Linq;
3
4 namespace LinqAbfragen
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Console.WriteLine(Person.AllePersonen.Any());
11             Console.WriteLine(Person.AllePersonen.Any(p =>
12 p.Vorname.StartsWith("X")));
13         }
14     }
15 }
```

11



```
C:\Users\rober\source\repos\CSharpBuch\LinqAbfragen\LinqAbfragen\bin\Debug\netcoreapp3.1\LinqAbfragen.exe (Prozess "2494")
True
False
C:\Users\rober\source\repos\CSharpBuch\LinqAbfragen\LinqAbfragen\bin\Debug\netcoreapp3.1\LinqAbfragen.exe (Prozess "2494"
") wurde mit Code "0" beendet
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.3.2 Ergebnisse der beiden Überladungen der Methode Any()

Der erste Wert ist true, da unsere Datenbasis Elemente enthält. Und der zweite Wert ist false, da unsere Datenbasis keine Person enthält, deren Vorname mit dem Buchstaben X beginnt.

## 11 Objekte verarbeiten mit „Linq to Objects“

Für nächste Methode `SelectMany()` reicht unsere Datenbasis nicht mehr aus, daher werden wir sie wie folgt erweitern: Zuerst erstellen wir die Klasse `Auto`.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace LingAbfragen
6  {
7      public class Auto
8      {
9          public Auto(string marke, string modell)
10         {
11             Marke = marke;
12             Modell = modell;
13         }
14
15         public string Marke;
16         public string Modell;
17
18         public override string ToString()
19         {
20             return $"{Marke} {Modell}";
21         }
22     }
23 }
```

Die Klasse `Auto` hat die beiden Properties `Marke` und `Modell` und einen Konstruktor, der die beiden Properties initialisiert. Außerdem überschreibt die Klasse `Auto` die Methode `ToString()`.

Zudem ändern wir die Klasse `Person`:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace LingAbfragen
6  {
7      public class Person
8      {
9          public Person(string vorname, string nachname)
10         {
11             Vorname = vorname;
12             Nachname = nachname;
13             Autos = new List<Auto>();
14         }
15
16         public string Vorname { get; set; }
17         public string Nachname { get; set; }
18
19         public List<Auto> Autos { get; }
20 }
```

```

21     public override string ToString()
22     {
23         return $"{Vorname} {Nachname}";
24     }
25
26     public static List<Person> AllePersonen
27     {
28         get
29         {
30             var personen = new List<Person>();
31             var auto1 = new Auto("Ford", "Focus");
32             var auto2 = new Auto("VW", "Golf");
33             var auto3 = new Auto("Opel", "Astra");
34             var auto4 = new Auto("Fiat", "Punto");
35             var auto5 = new Auto("Renault", "Megane");
36             var auto6 = new Auto("Seat", "Leon");
37
38             var person1 = new Person("Lisa", "Müller");
39             person1.Autos.Add(auto1);
40             person1.Autos.Add(auto2);
41
42             var person2 = new Person("Robert", "Schiefele");
43             person2.Autos.Add(auto3);
44             person2.Autos.Add(auto4);
45
46             var person3 = new Person("Jacqueline", "Meier");
47             person3.Autos.Add(auto5);
48             person3.Autos.Add(auto6);
49
50             personen.Add(person1);
51             personen.Add(person2);
52             personen.Add(person3);
53
54             return personen;
55         }
56     }
57
58 }
59 }
```

11

Die Klasse Person erhält die Property Autos vom Typ List<Auto>, in der die Autos einer Person gespeichert werden. Der Konstruktor der Klasse Person initialisiert die Property Autos mit einer leeren Liste. Die statische Property AllePersonen gibt eine Liste mit drei Personen zurück, wobei eine Person jeweils zwei Autos besitzt.

Mit der Methode SelectMany() können wir eine Liste aller Autos aller Personen zusammenstellen. Die Signatur von SelectMany() sieht wie folgt aus:

```

1 I Enumerable<TResult> I Enumerable<T>.
2 SelectMany<T, TResult>(Func<T, I Enumerable<TResult>> selector)
```

Wenn wir statt den generischen Typ-Variablen T und TResult die Typen Person und Auto aus unserer Datenbasis einsetzen, erhalten wir.

## 11 Objekte verarbeiten mit „Linq to Objects“

```
1 IEnumerable<Auto> IEnumerable<Person>.
2 SelectMany<Person,Auto>(Func<Person,IEnumerable<Auto>> selector)
```

Der Rückgabewert der Methode `SelectMany()` ist vom Typ `IEnumerable<Auto>`. Wir wollen ja eine Liste von Autos erhalten. Die Methode erwartet einen Übergabeparameter vom Typ `Func<Person,IEnumerable<Auto>>`. Das ist eine Methode, der ein Parameter vom Typ `Person` übergeben wird und die den Typ `IEnumerable<Auto>` zurückgibt. Diese Methode wird auf alle Elemente der Basisstruktur, die in unserem Fall den Typ `IEnumerable<Person>` hat, angewendet. Die Rückgabewerte der Methode für jedes Element werden der Ergebnisliste hinzugefügt.

In unserem Beispiel übergeben wir der Methode `SelectMany()` den Lambda-Ausdruck `p => p.Autos`, damit selektieren wir alle Autos einer Person und fügen die Autos der Ergebnisliste hinzu. Damit ergibt sich das folgende Hauptprogramm.

```
1 using System;
2 using System.Linq;
3
4 namespace LinqAbfragen
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var alleAutos = Person.AllePersonen.SelectMany(p =>
11                 p.Autos);
12             foreach(var auto in alleAutos)
13             {
14                 Console.WriteLine(auto);
15             }
16         }
17     }
18 }
```

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
Ford Focus
VW Golf
Opel Astra
Fiat Punto
Renault Megane
Seat Leon

C:\Users\rober\source\repos\CsharpBuch\LinqAbfragen\LinqAbfragen\bin\Debug\netcoreapp3.1\LINQAbfragen.exe (Prozess "23032") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.3.3 Bildschirmausgabe der Liste aller Autos aller Personen

## 11.4 Daten konvertieren mit Linq

In komplexeren Programmen kommt man öfter in die Situation, dass es eine komplexe, aber gut funktionierende Methode gibt, die eine Listenstruktur der Klasse A verar-

beitet, zum Beispiel `List<A>`. Zudem haben wir in unserem fiktiven Programm Daten in Form einer Listenstruktur der Klasse B, zum Beispiel `List<B>`. Die Strukturen der Klassen A und B sind nicht identisch, aber ähnlich. Die Methode unseres Programms, die den Typ `List<A>` entgegennimmt, würde genau die Verarbeitung machen, die wir für den Typ `List<B>` benötigen. Leider können wir ihr kein Objekt vom Typ `List<B>` übergeben. Eine naheliegende Lösung des Problems wäre es, eine Vorverarbeitung zu programmieren, in der wir mit einer Schleife `List<B>` durchlaufen. In der Schleife erzeugen wir ein Objekt vom Typ A, weisen ihm Daten des Objekts vom Typ B zu und fügen es einem Objekt vom Typ `List<A>` hinzu. Damit haben wir ein Objekt vom Typ `List<A>`, das wir an die Methode, die die von uns gewünschte Verarbeitung durchführt, übergeben können. Eine derartige Vorverarbeitung nennt man eine Datenkonvertierung. Mit Linq lässt sich so eine Datenkonvertierung wesentlich eleganter und kompakter formulieren als mit einer Schleife. Linq stellt dazu die Erweiterungsmethode `Select()` zur Verfügung. Aber um `Select()` vernünftig verwenden zu können, müssen wir erst eine weitere Variante zur Instanziierung von Objekten betrachten: die sogenannten Objektinitialisierer. Mit Objektinitialisierern kann man eine Instanz einer Klasse mit einer einzigen Anweisung erstellen und gleichzeitig einer oder mehreren Properties des Objekts Werte zuweisen. Ein Konstruktor, der Werte entgegennimmt, um sie an Properties weiterzugeben, wird damit überflüssig. Objektinitialisierer können prinzipiell immer verwendet werden, wenn wir eine Instanz einer Klasse erzeugen wollen. Das führt zu einem verständlichen und kompakten Quellcode. Eine besondere Rolle spielen Objektinitialisierer allerdings bei Linq-Ausdrücken. Wenn ein Lambda-Ausdruck nur ein Objekt erstellen und zurückgeben soll, benötigen wir eine Möglichkeit, das vollständige Objekt mit einer einzigen Anweisung zu erzeugen. Denn ein Lambda-Ausdruck ist kompakter und besser lesbar, wenn er mit nur einer einzigen Anweisung formuliert werden kann. Die meisten Linq-Ausdrücke in einem Programm können als eine einzige Anweisung formuliert werden, welche dann zur besseren Übersichtlichkeit umgebrochen und eingerückt werden sollte. Bevor wir die Erweiterungsmethode `Select()` näher betrachten, werden wir uns zuerst ansehen, wie Objektinitialisierer funktionieren.

11

Dazu ändern wir die Klasse `Person` wie folgt:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace KonvertierenMitLinq
6  {
7      public class Person
8      {
9          public string Vorname { get; set; }
10         public string Nachname { get; set; }
11
12         public static List<Person> AllePersonen
13         {
```

## 11 Objekte verarbeiten mit „Linq to Objects“

```

14         get
15     {
16         var personenListe = new List<Person>();
17
18         var person1 = new Person
19         {
20             Vorname = "Max",
21             Nachname = "Mustermann"
22         };
23
24         var person2 = new Person
25         {
26             Vorname = "Marta",
27             Nachname = "Musterfrau"
28         };
29
30         personenListe.Add(person1);
31         personenListe.Add(person2);
32
33         return personenListe;
34     }
35 }
36 }
37 }
```

Die Klasse Person hat jetzt keinen Konstruktor mehr, dafür haben die Properties Vorname und Nachname einen public-Setter. In der statischen Property AllePersonen werden die Instanzen von Person jetzt mit einem Objektinitialisierer erzeugt. Der Objektinitialisierer beginnt mit dem Schlüsselwort new, gefolgt vom Namen der Klasse, dann folgen die Zuweisungen zu den einzelnen Properties der Klasse durch Kommata getrennt und in geschweifte Klammern eingeschlossen.

Objektinitialisierer können auch zur Erzeugung anonymer Objekte verwendet werden. Damit können wir die Klasse Person weiter vereinfachen.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonvertierenMitLinq
6 {
7     public class Person
8     {
9         public string Vorname { get; set; }
10        public string Nachname { get; set; }
11
12        public static List<Person> AllePersonen
13        {
14            get
15            {
16                var personenListe = new List<Person>();
17
18                personenListe.Add(new Person
```

```
19     {
20         Vorname = "Max",
21         Nachname = "Mustermann"
22     });
23
24     personenListe.Add(new Person
25     {
26         Vorname = "Marta",
27         Nachname = "Musterfrau"
28     });
29
30     return personenListe;
31 }
32 }
33 }
34 }
```

Wir übergeben der Methode `Add()` der Variablen `personenListe` direkt einen Objektinitialisierer. Es ist auch möglich, eine ganze Struktur wie `List<Person>` über einen Objektinitialisierer als anonymes Objekt zu erzeugen. Das führt uns dann zu einer noch kompakteren Variante der Klasse `Person`.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonvertierenMitLinq
6 {
7     public class Person
8     {
9         public string Vorname { get; set; }
10        public string Nachname { get; set; }
11
12        public static List<Person> AllePersonen
13        {
14            get
15            {
16                return new List<Person>
17                {
18                    new Person
19                    {
20                        Vorname = "Max",
21                        Nachname = "Mustermann"
22                    },
23                    new Person
24                    {
25                        Vorname = "Marta",
26                        Nachname = "Musterfrau"
27                    }
28                };
29            }
30        }
31    }
32 }
```

## 11 Objekte verarbeiten mit „Linq to Objects“

Der Objektinitialisierer für eine Listenstruktur enthält die Objektinitialisierer für die einzelnen Elemente der Liste durch Kommata getrennt.

Um die Funktionsweise der Methode `Select()` zu zeigen, benötigen wir noch die Klasse `Adresse`

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonvertierenMitLinq
6 {
7     public class Adresse
8     {
9         public string Ort { get; set; }
10        public string Strasse { get; set; }
11    }
12 }
```

Des Weiteren lassen wir die Klasse `Person` von der Klasse `Adresse` erben.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonvertierenMitLinq
6 {
7     public class Person : Adresse
8     {
9         public string Vorname { get; set; }
10        public string Nachname { get; set; }
11
12        public static List<Person> AllePersonen
13        {
14            get
15            {
16                return new List<Person>
17                {
18                    new Person
19                    {
20                        Vorname = "Max",
21                        Nachname = "Mustermann",
22                        Ort = "Musterhausen",
23                        Strasse = "Musterstr. 1"
24                    },
25                    new Person
26                    {
27                        Vorname = "Marta",
28                        Nachname = "Musterfrau",
29                        Ort = "Musterstadt",
30                        Strasse = "Musterweg 7"
31                    }
32                };
33            }
34        }
35    }
36 }
```

```

34     }
35 }
36 }
```

In der statischen Property `AllePersonen` weisen wir den von `Adresse` geerbten Properties `Ort` und `Strasse` der einzelnen Element Werte zu.

Jetzt haben wir eine Listenstruktur, die für die Verwendung mit der Methode `Select` geeignet ist.

Die Methode `Select()` hat folgende Signatur:

```

1 IEnumerable<TResult> IEnumerable<T>.
2 Select<T, TResult>(Func<T, TResult> selector)
```

beziehungsweise

```

1 IEnumerable<TResult> IEnumerable<Person>.
2 Select<Person, TResult>(Func<Person, TResult> selector)
```

In unserem Beispiel wollen wir Objekte vom Typ `Person` in Objekte vom Typ `Adresse` konvertieren. Daher ergibt sich folgende konkrete Signatur:

```

1 IEnumerable<Adresse> IEnumerable<Person>.
2 Select<Person, Adresse>(Func<Person, Adresse> selector)
```

Damit erwartet die Methode `Select()` eine Konvertier-Methode, die einen Übergabeparameter vom Typ `Person` entgegennimmt und ein Objekt vom Typ `Adresse` zurückgibt. Die Methode `Select()` wendet die übergebene Methode auf alle Elemente der zugrundeliegenden Listenstruktur an.

11

Der Lambda Ausdruck `p => new Adresse { Ort = p.Ort, Strasse = p.Strasse }` stellt eine Methode dar, wie sie von der Methode `Select` erwartet wird. Sie ruft einen Objektinitialisierer für die Klasse `Adresse` auf und weist dabei den Properties `Ort` und `Strasse` der Klasse `Adresse` die Properties `Ort` und `Strasse` des übergebenen Objekts vom Typ `Person` zu.

Mit dem folgenden Hauptprogramm können wir die Methode `Select()` testen:

```

1 using System;
2 using System.Linq;
3
4 namespace KonvertierenMitLinq
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
```

## 11 Objekte verarbeiten mit „Linq to Objects“

```

10
11         var adressen = Person.AllePersonen.Where(p =>
12             p.Nachname.StartsWith("M")).Select(p => new Adresse
13             {Ort = p.Ort, Strasse = p.Strasse });
14
15         foreach(var adresse in adressen)
16         {
17             Console.WriteLine(adresse);
18         }
19     }
20 }
21 }
```

Natürlich könnte man statt der Methode `Select()` auch eine `foreach`-Schleife verwenden und innerhalb der Schleife ein Objekt vom Typ `Adresse` erzeugen, die Properties `Ort` und `Strasse` zuweisen und das `Adresse`-Objekt an eine Ergebnisliste anhängen. Aber der Linq-Ausdruck mit `Select()` ist deutlich kompakter und übersichtlicher. Zudem gibt `Select()` den Typ `IEnumerable<T>` zurück. Das heißt, wir könnten an `Select()` eine weitere Linq-Methode anhängen. Außerdem ist `Select()` flexibler, wenn es darum geht, wann genau eine Linq-Abfrage ausgeführt wird. Diese Aspekte werden wir in den Kapiteln „Linq-Methoden verketten“ und „Wann wird eine Linq-Abfrage ausgeführt“ näher beleuchten.

The screenshot shows the Microsoft Visual Studio Debugging Console window. The output text is:

```

Microsoft Visual Studio-Debugging-Konsole
Musterhausen Musterstr. 1
Musterstadt Musterweg 7

C:\Users\rober\source\repos\CSharpBuch\KonvertierenMitLinq\KonvertierenMitLinq\bin\Debug\netcoreapp3.1\KonvertierenMitLinq.exe (Prozess "8536") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 11.4.1** 11.41 Bildschirmausgabe der Konvertierung von Person zu Adresse

Die zweite Klasse ist für die Methode `Select()` nicht wirklich notwendig, da man einen Objektinitialisierer auch verwenden kann, ohne dafür eine Klasse zu definieren.

```
1 var objekt = new { a = 5, b = "Hallo" };
```

Die Variable `objekt` ist eine Instanz einer sogenannten anonymen Klasse. Sie hat zwei Properties `a` und `b` vom Typ `int` und `string`. So eine anonyme Klasse kann auch in einem Lambda-Ausdruck, der der Methode `Select()` übergeben wird, verwendet werden.

```

1 using System;
2 using System.Linq;
3
4 namespace KonvertierenMitLinq
5 {
6     class Program
7     {
8         static void Main(string[] args)
```

```
9     {
10
11     var personen = Person.AllePersonen.Select(p => new
12         {Vorname = p.Vorname, Nachname = p.Nachname });
13
14     foreach(var person in personen)
15     {
16         Console.WriteLine($"{person.Vorname} {person.
17             Nachname}");
18     }
19 }
20 }
21 }
```

Jetzt erzeugen wir mit `Select()` eine Liste von Objekten, deren Typ eine anonyme Klasse ist. Den Lambda-Ausdruck kann man auch etwas kompakter schreiben.

```
1 p => new {p.Vorname, p.Nachname}
```

Bei dieser Variante erzeugt der Compiler die Namen der Properties der anonymen Klasse selbst und verwendet dabei die Namen der Properties der zugewiesenen Werte.

## 11.5 Linq-Methoden verketten

In diesem Kapitel beschäftigen wir uns mit der Technik des Verkettens zweier oder mehrerer Linq-Methoden. Viele Linq-Methoden geben den Typ `IEnumerable<T>` zurück. Das heißt, wir können an den Aufruf einer Erweiterungsmethode von `IEnumerable<T>` mit einem Punkt eine weitere Erweiterungsmethode von `IEnumerable<T>` anhängen. Falls diese angehängte Methode auch `IEnumerable<T>` zurückgibt, können wir noch eine Erweiterungsmethode von `IEnumerable<T>` anhängen und so weiter. Dieses Aneinanderreihen von Methoden-Aufrufen nennt man Verkettung von Methoden.

In Linq gibt es Methoden, mit denen man eine mehrstufige Sortierung von Objektstrukturen implementieren kann. Dazu werden mehrere Aufrufe dieser Sortiermethoden miteinander verkettet.

Doch bevor wir uns das Sortieren von Listen anschauen, passen wir die statische Property „Alle Personen“ etwas an, um eine geeignete Datenbasis zu erhalten.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KonvertierenMitLinq
6 {
7     public class Person : Adresse
8     {
```

## 11 Objekte verarbeiten mit „Linq to Objects“

```

 9     public string Vorname { get; set; }
10    public string Nachname { get; set; }
11
12    public override string ToString()
13    {
14        return $"{Vorname} {Nachname} {Ort} {Strasse}";
15    }
16
17    public static List<Person> AllePersonen
18    {
19        get
20        {
21            return new List<Person>
22            {
23                new Person
24                {
25                    Vorname = "Max",
26                    Nachname = "Musterperson",
27                    Ort = "Musterhausen",
28                    Strasse = "Musterstr. 1"
29                },
30                new Person
31                {
32                    Vorname = "Marta",
33                    Nachname = "Musterperson",
34                    Ort = "Musterstadt",
35                    Strasse = "Musterweg 7"
36                },
37                new Person
38                {
39                    Vorname = "Tim",
40                    Nachname = "Testperson",
41                    Ort = "Testhausen",
42                    Strasse = "Testgasse 5"
43                },
44                new Person
45                {
46                    Vorname = "Tina",
47                    Nachname = "Testperson",
48                    Ort = "Testdorf",
49                    Strasse = "Testallee 8"
50                },
51            };
52        }
53    }
54}
55

```

Die erste Sortiermethode, die wir betrachten, heißt `OrderBy()` und hat die folgende Signatur:

```

1 IOrderedEnumerable<T> IEnumerable<T>.
2 OrderBy<T, TKey>(Func<T, TKey> keySelector)

```

beziehungsweise

```
1 IOrderedEnumerable<Person> IEnumerable<Person>.  
2 OrderBy<Person, TKey>(Func<Person, TKey> keySelector)
```

Wir wollen in einem ersten Beispiel unsere Liste vom Typ `List<Person>` nach dem Vornamen der Person sortieren. Da die Property `Person` vom Typ `string` ist, ergibt sich für unsere `OrderBy()`-Methode die konkrete Signatur:

```
1 IOrderedEnumerable<Person> IEnumerable<Person>.  
2 OrderBy<Person, string>(Func<Person, string> keySelector)
```

Die Methode `OrderBy()` erwartet eine Methode, der ein Parameter von Typ `Person` übergeben wird und die einen Wert vom Typ `String` zurückgibt. Diese Methode wird von `OrderBy()` für jedes Element der Liste aufgerufen, um den für die Sortierung relevanten Wert zu bestimmen.

Mit dem Lambda-Ausdruck `p => p.Nachname` erhalten wir eine Methode mit der passenden Signatur und können damit unsere Personenliste nach dem Vornamen der Personen sortieren.

Alle Linq-Methoden, die wir bis jetzt kennengelernt haben, die eine Listenstruktur zurückgeben, geben den Typ `IEnumerable<T>` zurück. Die Methode `OrderBy()` gibt den Typ `IOrderedEnumerable<T>` zurück. `IOrderedEnumerable<T>` ist von `IEnumerable<T>` abgeleitet. Das heißt, wir können Objekte vom Typ `IOrderedEnumerable<T>` genauso verwenden wie Objekte vom Typ `IEnumerable<T>`.

Aber für Objekte vom Typ `IOrderedEnumerable<T>` existiert eine weitere Erweiterungsmethode: `ThenBy()`. Mit dieser Methode können wir auf einer Listenstruktur geschachtelte Sortierungen vornehmen. Das heißt, wir sortieren unsere Personenliste zuerst nach Nachnamen und dann nach Vornamen. Alle Personen mit dem gleichen Nachnamen stehen dadurch nacheinander in der Liste. Aber jede Gruppe der Personen mit dem gleichen Nachnamen ist sortiert nach dem Vornamen.

Die Methode `ThenBy()` hat die gleiche Signatur wie die Methode `OrderBy()`.

```
1 IOrderedEnumerable<T> IOrderedEnumerable<T>.  
2 ThenBy<T, TKey>(Func<T, TKey> keySelector)
```

Da die Methode `OrderBy()` ein Objekt vom Typ `IOrderedEnumerable` zurückgibt, kann man direkt an die Methode `OrderyBy` mit einem Punkt die Methode `ThenBy()` anhängen.

```
1 var sortiertePersonen = Person.AllePersonen.OrderBy(p =>  
2 p.Nachname).ThenBy(p => p.Vorname);
```

## 11 Objekte verarbeiten mit „Linq to Objects“

Diese Technik nennt man Verketten von Methoden. Sie funktioniert für alle Methoden, sofern sie einen geeigneten Rückgabewert liefern. Zum Beispiel können wir mit der Methode `Where()` eine Liste filtern, dann mit Methode `Select()` die Elemente der Liste konvertieren und dann mit den Methoden `OrderBy()` und `ThenBy()` die Liste geschachtelt sortieren. Und das Ganze kann in einer einzigen Zeile formuliert werden:

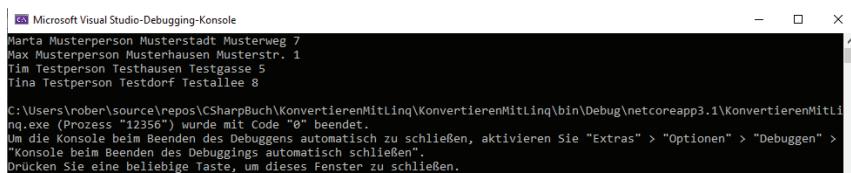
```
1 var ergebnis = Person.AllePersonen.Where(p => p.Nachname.  
2 StartsWith("M")).Select(p => new Adresse { Ort = p.Ort, Strasse =  
3 p.Strasse }).OrderBy(p => p.Ort).ThenBy(p => p.Strasse);
```

Allerdings empfehle ich Ihnen aus Gründen der Lesbarkeit derartige Programmzeilen umgebrochen und eingerückt zu schreiben.

```
1 var ergebnis = Person.AllePersonen  
2 .Where(p => p.Nachname.StartsWith("M"))  
3 .Select(p => new Adresse { Ort = p.Ort, Strasse = p.Strasse  
4 })  
5 .OrderBy(p => p.Ort)  
6 .ThenBy(p => p.Strasse);
```

Zum Abschluss dieses Kapitels betrachten wir noch ein Beispielprogramm, mit dem wir unsere Personenlisten zuerst nach Nachnamen und dann nach Vornamen sortieren.

```
1 using System;  
2 using System.Linq;  
3  
4 namespace KonvertierenMitLinq  
5 {  
6     class Program  
7     {  
8         static void Main(string[] args)  
9         {  
10             var sortiertePersonen = Person.AllePersonen.OrderBy(p  
11             => p.Nachname).ThenBy(p => p.Vorname);  
12  
13             foreach(var person in sortiertePersonen)  
14             {  
15                 Console.WriteLine(person);  
16             }  
17         }  
18     }  
19 }
```



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays a sorted list of people from a database. The output is as follows:

```
C:\Users\rrober\source\repos\CSharpBuch\KonvertierenMitLinq\KonvertierenMitLinq\bin\Debug\netcoreapp3.1\KonvertierenMitLinq.exe (Prozess "12356") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.5.1 Die Personenliste sortiert nach Nachnamen und Vornamen

## 11.6 Verschachtelte Linq-Ausdrücke

Linq-Ausdrücke können nicht nur verkettet, sondern auch ineinander verschachtelt werden. Bei der Verkettung reihen wir mehrere Linq-Methoden aneinander. Das heißt, eine Linq-Methode wird auf den Rückgabewert der vorhergehenden Linq-Methode angewendet.

Bei der Verschachtelung dagegen starten wir mit einer Linq-Methode, der ein Lambda-Ausdruck als Parameter übergeben wird. Der übergebene Lambda-Ausdruck enthält wieder einen Linq-Ausdruck.

So eine Verschachtelung kann auch tiefer als nur zwei Ebenen gehen.

Bevor wir mit verschachtelten Linq-Ausdrücken arbeiten können, benötigen wir natürlich wieder eine Datenbasis. Die erste Klasse für unsere Datenbasis ist die Klasse Auto.

11

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace VerschachtelteLinqAusdrucke
6  {
7      public class Auto
8      {
9          public string Marke { get; set; }
10         public string Modell { get; set; }
11
12         public override string ToString()
13         {
14             return $"{Marke} {Modell}";
15         }
16     }
17 }
```

Des Weiteren benötigen wir die Klasse Person:

```
1  using System;
2  using System.Collections.Generic;
```

**11 Objekte verarbeiten mit „Linq to Objects“**

```
3  using System.Text;
4
5  namespace VerschachtelteLinqAusdruecke
6  {
7      public class Person
8      {
9          public string Nachname { get; set; }
10         public string Vorname { get; set; }
11
12         public IEnumerable<Auto> Autos { get; set; }
13
14         public override string ToString()
15         {
16             var autos = "";
17             foreach (var auto in Autos)
18             {
19                 autos += $"{auto} ";
20             }
21
22             return $"{Vorname} {Nachname} {autos}";
23         }
24
25         public static IEnumerable<Person> AllePersonen
26     {
27         get
28     {
29         return new List<Person>
30     {
31         new Person
32     {
33             Nachname = "Müller",
34             Vorname = "Hans",
35             Autos = new List<Auto>
36     {
37             new Auto {Marke = "BMW", Modell =
38             "320i"}, 
39             new Auto {Marke = "Fiat", Modell =
40             "Panda"}
41         }
42     },
43     new Person
44     {
45         Nachname = "Meier",
46         Vorname = "Julia",
47         Autos = new List<Auto>
48     {
49         new Auto {Marke = "Audi", Modell =
50         "A4"}, 
51         new Auto {Marke = "Fiat", Modell =
52         "Punto"}
53     }
54 },
55     new Person
56     {
57         Nachname = "Huber",
58         Vorname = "Gerd",
59     }
60 }
```

```

59         Autos = new List<Auto>
60         {
61             new Auto {Marke = "Mercedes", Modell
62             = "E240"}, 
63             new Auto {Marke = "Ford", Modell =
64             "Fiesta"}
65         }
66     },
67     new Person
68     {
69         Nachname = "Mahler",
70         Vorname = "Sonja",
71         Autos = new List<Auto>
72         {
73             new Auto {Marke = "VW", Modell =
74             "Phaeton"}, 
75             new Auto {Marke = "Ford", Modell =
76             "Focus"}
77         }
78     };
79 }
80 }
81 }
82 }
83 }
84 }
```

Die Klasse Person hat die Properties Vorname und Nachname und die Property Autos vom Typ `IEnumerable<Autos>`, um die der Person gehörenden Autos abzuspeichern. Die statische Property AllePersonen liefert uns eine initialisierte Personenliste vom Typ `IEnumerable<Person>`, die einige Person-Objekte inklusive den ihnen zugeordneten Auto-Objekten enthält.

Wenn wir jetzt zum Beispiel eine Liste der Personen benötigen, die ein Auto der Marke Fiat besitzen, können wir das mit einem geschachtelten Linq-Ausdruck erreichen.

```

1  using System;
2  using System.Linq;
3
4  namespace VerschachtelteLinqAusdruecke
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var fiatBesitzer = Person.AllePersonen.Where(p =>
11             p.Autos.Any(a => a.Marke == "Fiat"));
12
13             foreach (var person in fiatBesitzer)
14             {
15                 Console.WriteLine(person);
16             }
17         }
18     }
```

## 11 Objekte verarbeiten mit „Linq to Objects“

```
18     }
19 }
```

Wir rufen für die Property `AllePersonen` die Methode `Where()` auf. Der Methode `Where()` übergeben wir mit einem Lambda-Ausdruck eine Methode, die ein `Person`-Objekt entgegennimmt. Für die Property `Autos` des übergebenen `Person`-Objekts rufen wir die Linq-Methode `Any()` auf. Der Methode `Any()` wird eine Methode übergeben, der ein Objekt vom Typ `Auto` übergeben wird. Diese Methode gibt den Boole'schen Wert `true` zurück, wenn die Property `Marke` des übergebenen `Auto`-Objekts den Wert „Fiat“ enthält. Damit können wir mit dem Linq-Ausdruck:

```
1 Person.AllePersonen.Where(p => p.Autos.Any(a => a.Marke ==
2 "Fiat"));
```

alle Fiat-Besitzer selektieren.

```
Microsoft Visual Studio-Debugging-Konsole
Hans Müller BMW 320i Fiat Panda
Julia Meier Audi A4 Fiat Punto

C:\Users\rober\source\repos\CSharpBuch\VerschachteltingAusdruecke\VerschachteltingAusdruecke\bin\Debug\netcoreapp3.1\VerschachteltingAusdruecke.exe (Prozess "14440") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.6.1 Bildschirmausgabe der Fiat-Besitzer

## 11.7 Wann wird eine Linq Abfrage ausgeführt?

Linq-Abfragen werden verzögert ausgeführt. Um das zu verstehen, stellen Sie sich vor, Sie fragen keine Objektliste ab, die fünf Zeichenketten enthält, sondern eine Objektliste die 100.000 komplexe Objekte enthält. Trotzdem würde Ihr Computer für die Zuweisung eines Linq-Ausdrucks für diese Liste weniger als eine Millisekunde benötigen. Die Zuweisung eines Linq-Ausdrucks an eine Variable führt keine Abfrage aus, sondern definiert nur eine Abfrage. Zur Veranschaulichung betrachten wir das folgende kleine Beispielprogramm:

```
1 using System;
2 using System.Linq;
3
4 namespace AusfuerenVonLinqAbfragen
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var arbeitsWoche = new string[]{ "Montag",
11                 "Dienstag", "Mittwoch", "Donnerstag", "Freitag" };
12
13             var abfrageTageMitM = arbeitsWoche.Where(t =>
14                 t.StartsWith("M"));
```

## 11.7 Wann wird eine Linq Abfrage ausgeführt?

```

15 }
16 }
17 }
```

Wenn Sie in Visual Studio die Maus über die Variable `abfrageTageMitM` bewegen, sehen Sie, dass der Typ der Variablen nicht `List<string>` ist, sondern `IEnumerable<string>`.

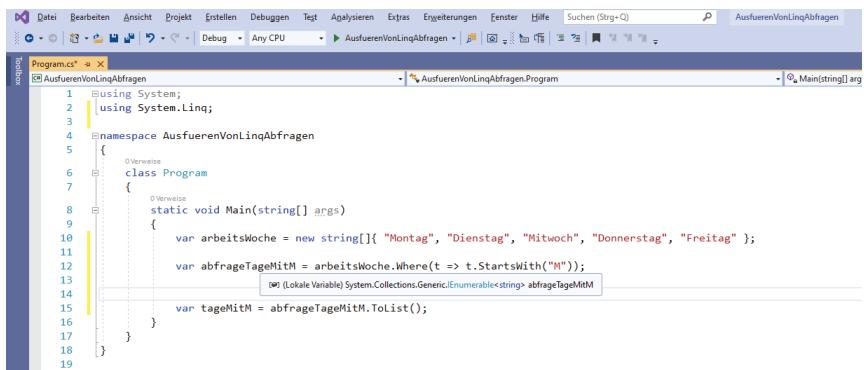


Abb. 11.7.1 Die Zuweisung eines Linq-Ausdrucks führt noch keine Abfrage aus.

Wenn sie das Ergebnis ihrer Linq-Abfrage jetzt auf dem Bildschirm ausgeben wollen, können Sie das wie folgt tun:

```

1 foreach(var tag in abfrageTageMitM)
2 {
3     Console.WriteLine(tag);
4 }
```

11

Erst wenn die Daten wirklich benötigt werden, also beim Ausführen der `foreach`-Schleife, wird die Abfrage ausgeführt.

Eine weitere, häufig verwendete Methode zum Ausführen einer Linq-Abfrage ist die Methode `ToList()`:

```
1 var tageMitM = abfrageTageMitM.ToList();
```

Mit dieser Codezeile wird die Linq-Abfrage ausgeführt. Wenn Sie jetzt die Maus in Visual Studio über die Variable `liste` bewegen, dann sehen Sie: Die Variable ist nicht mehr von Typ `IEnumerable<string>`, sondern vom Typ `List<string>`.

## 11 Objekte verarbeiten mit „Linq to Objects“

```

1  using System;
2  using System.Linq;
3
4  namespace AusfuerenVonLinqAbfragen
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var arbeitsWoche = new string[]{"Montag", "Dienstag", "Mitwoch", "Donnerstag", "Freitag"};
11
12             var abfrageTageMitM = arbeitsWoche.Where(t => t.StartsWith("M"));
13
14
15         }
16     }
17 }
18

```

The screenshot shows the Microsoft Visual Studio interface with the code editor open. The code uses LINQ's `Where` method to filter an array of strings representing days of the week based on whether they start with the letter 'M'. A tooltip for the `ToList()` method is visible at the bottom of the code editor.

Abb. 11.7.2 Die Methode `ToList()` führt eine Linq-Abfrage aus

## 11.8 Parallele Verarbeitung mit Linq

Heutzutage verfügt ein Computer über mehrere Prozessorkerne. Jeder dieser Kerne kann als eigener kleiner Computer aufgefasst werden, wobei die einzelnen Kerne gleichzeitig arbeiten können und sich den Hauptspeicher teilen. Normalerweise bekommen wir als Benutzer nicht viel davon mit. Denn welcher Kern, wann und mit welchem anderen Kern gleichzeitig aktiv ist, das regelt das Betriebssystem. Da aber das gleichzeitige Verwenden mehrerer Kerne bei sogenannten parallelisierbaren Problemstellungen sehr vorteilhaft sein kann, gibt es im .NET-Framework eine Möglichkeit, die Verwendung der Kerne zu beeinflussen. Um eine Listenstruktur parallelisiert zu verarbeiten, benötigen wir die Methode `AsParallel` mit folgender Signatur:

```
1 ParallelQuery<T> IEnumerable<T>.AsParallel()
```

Die Methode `AsParallel()` hat keinen Übergabeparameter und gibt ein Objekt vom Typ `ParallelQuery<T>` zurück. `ParallelQuery<T>` implementiert `IEnumerable<T>`, daher stehen uns für Objekte vom Typ `ParallelQuery<T>` die gleichen Linq-Methoden zur Verfügung wie für `IEnumerable<T>`. Wenn wir `ParallelQuery<T>` verwenden, versucht Linq die Abfrage so gut wie möglich zu parallelisieren und dann alle Prozessorkerne gleichzeitig einzusetzen, was - je nach Problemstellung und Anzahl der verfügbaren Prozessorkerne - zu einem beträchtlichen Performancegewinn führen kann. Außerdem gibt es für `ParallelQuery<T>` noch die sehr praktische Erweiterungsmethode `ForAll()`. Mit dieser Methode kann man mit einem Aufruf eine Aktion für alle Elemente einer Struktur vom Typ `ParallelQuery<T>` durchführen. Die Methode hat die Signatur:

```
1 void ParallelQuery<T>.ForAll(Action<T> action)
```

Die Methode `ForAll()` hat keinen Rückgabewert und erwartet als Übergabeparameter eine Methode vom Typ `Action<T>`, das heißt eine Methode ohne Rückgabe, die auf alle Elemente der zu verarbeitenden Struktur angewendet wird.

Nun wollen wir die Parallelverarbeitung in der Praxis betrachten:

```
1  using System;
2  using System.Linq;
3
4  namespace Linqtutorials
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             int[] intWerte = Enumerable.Range(1, 10000).
11             ToArray();
12
13             var startZeit = DateTime.Now.Ticks;
14             for(var i = 0; i < intWerte.Length; i++)
15             {
16                 intWerte[i]++;
17             }
18             Console.WriteLine($"Verarbeitungszeit mit
19             klassischem Linq: {TimeSpan.FromTicks(DateTime.Now.
20             Ticks-startZeit). TotalMilliseconds} ms");
21
22             startZeit = DateTime.Now.Ticks;
23             intWerte.AsParallel().ForAll((wert) => wert++);
24             Console.WriteLine($"Verarbeitungszeit mit Parallel
25             Linq: {TimeSpan.FromTicks(DateTime.Now.Ticks -
26             startZeit). TotalMilliseconds} ms");
27         }
28     }
29 }
```

11

Zuerst erzeugen wir mit Hilfe der statischen Methode `Range` der Klasse `Enumerable` aus dem .NET-Framework Integer-Werte von 1 bis 10.000 und legen sie im Array `intWerte` ab. Dann speichern wir für die Zeitmessung den aktuellen Zeitpunkt in der Variablen `startZeit`. Als nächstes laufen wir mit einer `for`-Schleife über das Array `intWerte` und erhöhen jedes Element des Arrays um eins. Nach der Schleife geben wir die bis hierher benötigte Zeit am Bildschirm aus.

Im Anschluss wollen wir die gleiche Aufgabe in parallelisierter Form ausführen. Damit wir die Zeit messen können, speichern wir wieder den aktuellen Zeitpunkt in der Variablen `startZeit`.

Wir rufen für das Array `intWerte` die Linq-Methode `AsParallel()` auf und verketten dann die Methode `ForAll()` und übergeben ihr den Lambda-Ausdruck

## 11 Objekte verarbeiten mit „Linq to Objects“

(wert) => wert++, der das Erhöhen der Array-Elemente erledigt. Danach geben wir die benötigte Zeit am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
Verarbeitungszeit mit klassischem Linq: 7,0156 ms
Verarbeitungszeit mit Parallel Linq: 85,3673 ms

C:\Users\rober\source/repos\CSharpBuch\ParallelVerarbeitungMitLinq\ParallelVerarbeitungMitLinq\bin\Debug\netcoreapp3.1\ParallelVerarbeitungMitLinq.exe (Prozess "3140") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 11.8.1 Mit Parallelisierung dauert es deutlich länger

Dieses Ergebnis ist eine Überraschung: Die parallelisierte Variante ist über zehn Mal so langsam wie die nicht parallelisierte. Die Parallelisierung an sich erzeugt einen gewissen Overhead, der zunächst für einen Zeitverlust sorgt. Wenn dann die Operationen auf den einzelnen Elementen der Listenstruktur parallel erfolgen, wird wieder Zeit gewonnen.

In unserem Beispiel erhöhen wir für jedes Element nur einen Integer-Wert um eins, das ist eine Aufgabe die ein moderner Computer in sehr kurzer Zeit, deutlich unter einer Millisekunde, erledigen kann. Das heißt unser Zeitgewinn durch die Parallelisierung ist eher gering. Daher ist in dieser Konstellation die Parallelisierung kontraproduktiv.

Als nächstes wollen wir in der Parallelverarbeitung für jedes Element eine Aufgabe erledigen, die etwas länger dauert. Dazu ändern wir unser Beispielprogramm etwas ab:

```

1  using System;
2  using System.Linq;
3  using System.Threading;
4
5  namespace Linqtutorials
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             int[] intWerte = Enumerable.Range(1, 10000).
12             ToArray();
13
14             var startZeit = DateTime.Now.Ticks;
15             for(var i = 0; i < intWerte.Length; i++)
16             {
17                 Thread.Sleep(1);
18             }
19             Console.WriteLine($"Verarbeitungszeit mit
20 klassischem Linq: {TimeSpan.FromTicks(DateTime.Now.
21 Ticks-startZeit). TotalMilliseconds} ms");
22
23             startZeit = DateTime.Now.Ticks;
24             intWerte.AsParallel().ForAll((wert) => Thread.

```

```

25     Sleep(1));
26     Console.WriteLine($"Verarbeitungszeit mit Parallel
27     Linq: {TimeSpan.FromTicks(DateTime.Now.Ticks -
28         startZeit).TotalMilliseconds} ms");
29 }
30 }
31 }
```

Anstelle des Hochzählens eines Integer-Wertes führen wir die Anweisung `Thread.Sleep(1)` aus. Die Methode `Sleep()` der Klasse `Thread` wartet die Anzahl an Millisekunden, die dem übergebenen Integer-Parameter entspricht.

The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays two execution times: "Verarbeitungszeit mit klassischem Linq: 19096,1157 ms" and "Verarbeitungszeit mit Parallel Linq: 4906,9035 ms". Below these, there is a command-line output from a .NET Core application named "ParallelVerarbeitungMitLinq.exe". The output includes the execution times again, followed by a message about how to close the console when debugging is finished, and a final instruction to press any key to close the window.

```

Microsoft Visual Studio-Debugging-Konsole
Verarbeitungszeit mit klassischem Linq: 19096,1157 ms
Verarbeitungszeit mit Parallel Linq: 4906,9035 ms

C:\Users\röber\source\repos\CSharpBuch\ParallelVerarbeitungMitLinq\ParallelVerarbeitungMitLinq\bin\Debug\netcoreapp3.1\ParallelVerarbeitungMitLinq.exe (Prozess "5220") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.8.2 Bei zeitintensiven Aufgaben gewinnt die Parallelisierung

Wenn die Aufgaben, die parallel ausgeführt werden sollen, jeweils eine Millisekunde dauern, ist eine Parallelverarbeitung deutlich schneller. Die Parallelverarbeitung ist fast viermal so schnell. Der Faktor vier ergibt sich daraus, dass der Computer, auf dem dieses Beispielprogramm ausgeführt wurde, über vier Kerne verfügt.

Ein Nachteil der Parallelverarbeitung ist, dass die Reihenfolge, in der die Ergebnisse der parallelen Aufgaben eingeliefert werden, nicht beeinflussbar ist. Wenn in Ihrer Verarbeitung also die Reihenfolge eine Rolle spielt, müssen Sie nach der Parallelverarbeitung wieder sortieren, was zusätzlich noch Zeit benötigt.

Detailliertere Betrachtungen über Parallelverarbeitung würden den Rahmen dieses Buches sprengen. Für Ihre Programmierpraxis sollten Sie sich folgende Aspekte der Parallelverarbeitung merken:

Parallelverarbeitung kann einen Vorgang stark beschleunigen, wenn ...

- der Computer, auf dem das Programm laufen soll, viele Prozessorkerne hat.

- die einzelnen Aufgaben, die parallel erledigt werden, nicht zu kurz sind (eine Millisekunde oder mehr).

- die einzelnen Aufgaben möglichst parallelisierbar sind. Das heißt, wenn eine Aufgabe nicht vom Ergebnis einer anderen Aufgabe abhängt.

- die Reihenfolge, in der die Ergebnisse geliefert werden, keine Rolle spielt.

## 11 Objekte verarbeiten mit „Linq to Objects“

Als Richtlinie für die Praxis gilt: Programmieren Sie einen Vorgang erst mal ohne Parallelverarbeitung und wenn die Performance nicht den Erwartungen entspricht, versuchen Sie es mit Parallelverarbeitung.

### 11.9 Übungsaufgabe: Linq verwenden

In dieser Übung werden wir zum Erstellen von Linq-Abfragen die folgenden beiden Klassen Verwenden:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace UebungLinq
6  {
7      public class Schueler
8      {
9          public string Vorname { get; set; }
10
11         public string Nachname { get; set; }
12
13         public override string ToString()
14         {
15             return $"{Vorname} {Nachname}";
16         }
17     }
18 }
19
20 using System;
21 using System.Collections.Generic;
22 using System.Text;
23
24 namespace UebungLinq
25 {
26     public class Schulkasse
27     {
28         public string Bezeichnung { get; set; }
29
30         public List<Schueler> Schueler {get; set;}
31
32         public override string ToString()
33         {
34             return Bezeichnung;
35         }
36     }
37 }
```

#### Teilaufgabe 1:

Schreiben Sie für die Klasse `Schulkasse` die statische Property `AlleKlassen`. Die Property hat den Typ `List<Schulkasse>`. Die Property `AlleKlassen` soll ein

paar Objekte vom Typ `Schulkasse` enthalten. Verwenden Sie folgende Werte für die Schulklassen:

1a

Max Müller

Anna Meier

2b

Fritz Huber

Anna Neumann

3c

Gerda Müller

Martin Mahler

4d

Die Schulkasse 4d hat keine Schüler. Legen Sie daher für die Klasse 4d eine leere Schülerliste an.

Verwenden Sie Objektinitialisierer zum Erstellen der Objektinstanzen.

### **Teilaufgabe 2:**

Bestimmen Sie im Hauptprogramm mit Hilfe einer Linq-Abfrage die Klasse, die keine Schüler hat und geben Sie die Bezeichnung dieser Klasse am Bildschirm aus.

11

### **Teilaufgabe 3:**

Erstellen Sie mit Linq eine Liste aller Klassen, die mindestens einen Schüler haben, dessen Nachname mit dem Buchstaben M beginnt. Geben Sie die Bezeichnungen der Klassen am Bildschirm aus.

### **Teilaufgabe 4:**

Erstellen Sie mit Hilfe einer verketteten Linq-Abfrage eine alphabetisch sortierte Liste der Vornamen aller Schüler aller Klassen und geben Sie diese am Bildschirm aus.

**Musterlösung für Teilaufgabe 1:**

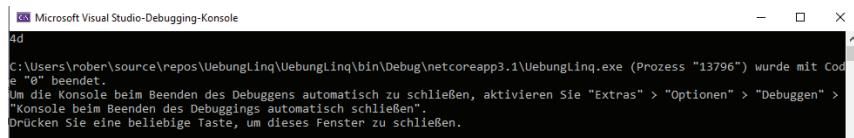
```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace UebungLinq
6  {
7      public class Schulkasse
8      {
9          public string Bezeichnung { get; set; }
10
11         public List<Schueler> Schueler {get; set;}
12
13         public override string ToString()
14         {
15             return Bezeichnung;
16         }
17
18         public static List<Schulkasse> AlleKlassen
19         {
20             get
21             {
22                 return new List<Schulkasse>
23                 {
24                     new Schulkasse
25                     {
26                         Bezeichnung = "1a",
27                         Schueler = new List<Schueler>
28                         {
29                             new Schueler { Vorname = "Max",
30                                 Nachname = "Müller" },
31                             new Schueler { Vorname = "Anna",
32                                 Nachname = "Meier" },
33                         }
34                     },
35                     new Schulkasse
36                     {
37                         Bezeichnung = "2b",
38                         Schueler = new List<Schueler>
39                         {
40                             new Schueler { Vorname = "Fritz",
41                                 Nachname = "Huber" },
42                             new Schueler { Vorname = "Anna",
43                                 Nachname = "Neumann" },
44                         }
45                     },
46                     new Schulkasse
47                     {
48                         Bezeichnung = "3c",
49                         Schueler = new List<Schueler>
50                         {
51                             new Schueler { Vorname = "Gerda",
52                                 Nachname = "Müller" },
53                             new Schueler { Vorname = "Martin",
54                                 Nachname = "Mahler" },
55                         }
56                     }
57                 }
58             }
59         }
60     }
```

```
55         }
56     },
57     new Schulkasse
58     {
59         Bezeichnung = "4d";
60         Schueler = new List<Schueler>()
61     },
62     );
63 }
64 }
65 }
66 }
```

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2  using System.Linq;
3
4  namespace UebungLinq
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              var klasseOhneSchueler = Schulkasse.AlleKlassen
11                  .Where(sk => sk.Schueler.Count() == 0)
12                  .FirstOrDefault();
13
14              Console.WriteLine(klasseOhneSchueler);
15          }
16      }
17 }
```

Die Variable `klasseOhneSchueler` ist vom Typ `Schulkasse`. Da die Klasse `Schulkasse`, die Methode `ToString()` mit einer Methode überschreibt, die den Wert der Property `Bezeichnung` zurückgibt, gibt der Aufruf `Console.WriteLine(klasseOhneSchueler);` die Property `Bezeichnung` des Objekts `klasseOhneSchueler` aus.



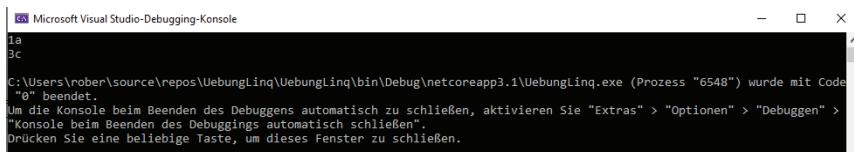
The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The content of the console is as follows:

```
4d
C:\Users\rober\source\repos\UebungLinq\UebungLinq\bin\Debug\netcoreapp3.1\UebungLinq.exe (Prozess "13796") wurde mit Code "q" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 11.9.1 Die Klasse 4d hat keine Schüler.

**Musterlösung für Teilaufgabe 3:**

```
1 using System;
2 using System.Linq;
3
4 namespace UebungLinq
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var klassenMitSchuelerMitM = Schulklasse.AlleKlassen
11                 .Where(sk => sk.Schueler.Any(s => s.Nachname.
12                     StartsWith("M")));
13
14             foreach (var klasse in klassenMitSchuelerMitM)
15             {
16                 Console.WriteLine(klasse);
17             }
18         }
19     }
20 }
```



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar says "Microsoft Visual Studio-Debugging-Konsole". The console output area displays two lines of text: "1a" and "3c". Above the output, there is some explanatory text in German: "C:\Users\rober\source\repos\UebungLinq\UebungLinq\bin\Debug\netcoreapp3.1\UebungLinq.exe (Prozess "6548") wurde mit Code : "0" beendet. Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen". Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.".

**Abb. 11.9.2** Die Klassen 1a und 3c haben Schüler, deren Nachnamen mit M beginnen.

## 11 Objekte verarbeiten mit „Linq to Objects“

**Musterlösung für Teilaufgabe 4:**

```
1  using System;
2  using System.Linq;
3
4  namespace UebungLinq
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              var vornamenAllerSchueler = Schulkasse.AlleKlassen
11                  .SelectMany(sk => sk.Schueler)
12                  .OrderBy(s => s.Vorname)
13                  .Select(s => s.Vorname);
14
15              foreach (var vorname in vornamenAllerSchueler)
16              {
17                  Console.WriteLine(vorname);
18              }
19          }
20      }
21 }
```



The screenshot shows the Microsoft Visual Studio Debugging-Konsole window. It displays a list of student names: Anna, Anna, Fritz, Gerda, Martin, Max. Below the names, there is some diagnostic text from the application's output.

```
Microsoft Visual Studio-Debugging-Konsole
Anna
Anna
Fritz
Gerda
Martin
Max

C:\Users\rober\source\repos\UebungLinq\UebungLinq\bin\Debug\netcoreapp3.1\UebungLinq.exe (Prozess "17832") wurde mit Code
a "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 11.9.3** Eine sortierte Liste der Vornamen aller Schüler

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 12

# Fehlerbehandlung mit Exceptions

Bisher haben wir gelegentlich gesehen, dass es beim Ablauf von Programmen zu Fehlern kommen kann, obwohl diese Programme korrekt kompilierbar sind. Solche Fehler können manchmal auch vom Verhalten des Benutzers abhängen. Das heißt, sie können auftreten, müssen es aber nicht, je nachdem, was der Benutzer tut. Solche Fehler und den Umgang damit wollen wir in den folgenden Kapiteln näher betrachten.

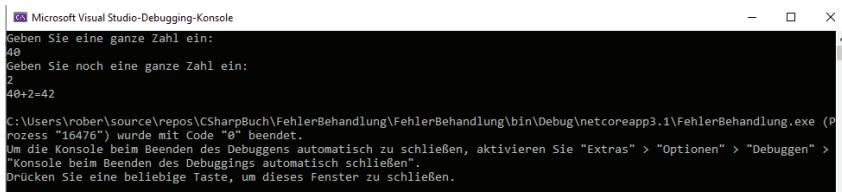
### 12.1 Was ist eine Exception?

Fehler, die zur Laufzeit eines Programms auftreten, werden im .NET-Framework Exceptions (englisch für Ausnahmen) genannt. Eine Exception tritt immer dann auf, wenn es zu einer Situation kommt, in der das Programm nicht mehr normal fortgesetzt werden kann. Wenn der Programmierer nicht durch spezielle Anweisungen festlegt, wie das Programm im Fehlerfall fortgesetzt werden soll, bricht das Programm ab. Betrachten wir hierzu folgendes einfaches Beispiel.

```
1  using System;
2
3  namespace FehlerBehandlung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
10             var eingabeZahl1 = Console.ReadLine();
11
12             Console.WriteLine("Geben Sie noch eine ganze Zahl
13             ein:");
14             var eingabeZahl2 = Console.ReadLine();
15
16             var zahl1 = int.Parse(eingabeZahl1);
17             var zahl2 = int.Parse(eingabeZahl2);
18
19             Console.WriteLine($"{zahl1}+{zahl2}={zahl1 +
20             zahl2}");
21         }
22     }
23 }
```

Wir lesen zwei Benutzereingaben in die beiden string-Variablen `eingabeZahl1` und `eingabeZahl2` ein. Danach konvertieren wir die beiden string-Variablen in die bei-

den int-Variablen zahl1 und zahl2 und geben die Summe dieser beiden Werte am Bildschirm aus.

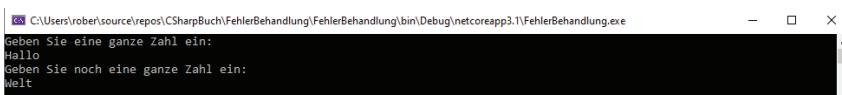


```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
40
Geben Sie noch eine ganze Zahl ein:
2
40+2=42

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (P
rozess "16476") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

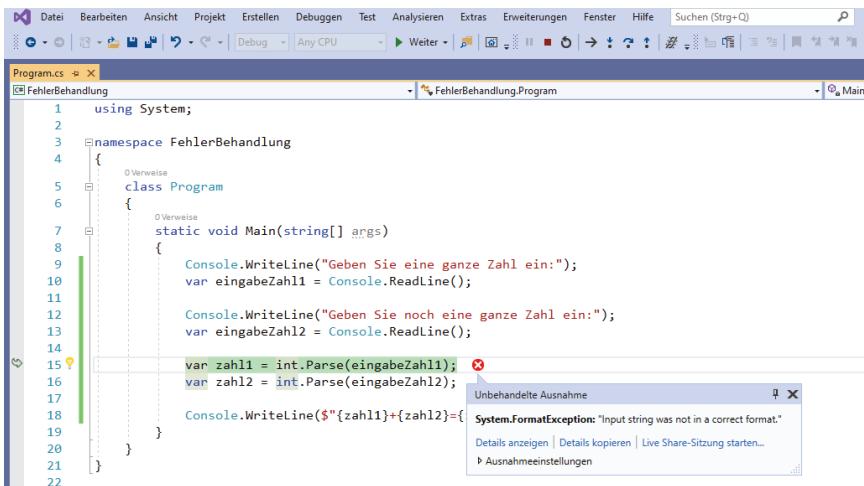
Abb. 12.1.1 Bei der Eingabe von ganzen Zahlen tritt kein Fehler auf

Wenn der Benutzer, wie vom Programm erwartet, zwei ganze Zahlen eingibt, läuft das Programm fehlerfrei zu Ende. Aber was passiert, wenn der Benutzer statt der ganzen Zahlen Texte eingibt?



```
C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe
Geben Sie eine ganze Zahl ein:
Hallo
Geben Sie noch eine ganze Zahl ein:
Welt
```

Abb. 12.1.2 Jetzt geben wir Texte statt Zahlen ein



12

Program.cs

```
using System;
namespace FehlerBehandlung
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Geben Sie eine ganze Zahl ein:");
            var eingabeZahl1 = Console.ReadLine();

            Console.WriteLine("Geben Sie noch eine ganze Zahl ein:");
            var eingabeZahl2 = Console.ReadLine();

            var zahl1 = int.Parse(eingabeZahl1);
            var zahl2 = int.Parse(eingabeZahl2);

            Console.WriteLine($"{zahl1}+{zahl2}={");
        }
    }
}
```

Unbehandelte Ausnahme

System.FormatException: "Input string was not in a correct format."

Details anzeigen | Details kopieren | Live Share-Sitzung starten...  
Ausnahmeeinstellungen

Abb. 12.1.3 Ein Text kann nicht in eine ganze Zahl konvertiert werden

Das Programm bricht bei der Zeile `var zahl1 = int.Parse(eingabeZahl1);` ab, da es logisch nicht möglich ist, den Text „Hallo“ in eine ganze Zahl zu konvertieren. Die gemeldete Ausnahme ist ein Objekt vom Typ `FormatException`, welche in der Klassenbibliothek `System` definiert ist.

## 12.2 Exceptions abfangen mit try – catch

Für den Benutzer sieht so etwas ziemlich unschön aus. Mit der folgenden Änderung an unserem Programm können wir eine solche Fehlersituation erkennen und mit einer für den Benutzer verständlichen Fehlermeldung reagieren.

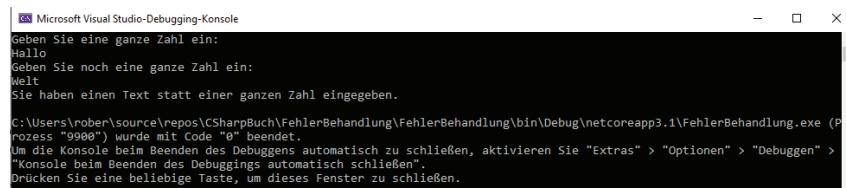
```
1  using System;
2
3  namespace FehlerBehandlung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
10             var eingabeZahl1 = Console.ReadLine();
11
12             Console.WriteLine("Geben Sie noch eine ganze Zahl
13             ein:");
14             var eingabeZahl2 = Console.ReadLine();
15
16             try
17             {
18                 var zahl1 = int.Parse(eingabeZahl1);
19                 var zahl2 = int.Parse(eingabeZahl2);
20
21                 Console.WriteLine($"{{zahl1}}+{{zahl2}}={{zahl1 +
22                 zahl2}}");
23             }
24             catch (FormatException ex)
25             {
26                 Console.WriteLine("Sie haben einen Text statt
27                 einer ganzen Zahl eingegeben.");
28             }
29         }
30     }
31 }
```

Das Einlesen der Benutzereingaben in string-Variablen ist unkritisch, da eine string-Variable alles aufnehmen kann, was ein Benutzer tippt. Die kritischen Programmzeilen bestehen aus der Konvertierung von string-Variablen in int-Variablen und der Ausgabezeile, die nicht ausgeführt werden soll, wenn mindestens eine der EingabevARIABLEN nicht konvertiert werden kann.

Den kritischen Programmteil umschließen wir mit geschweiften Klammern und stellen ihm das Schlüsselwort `try` voran. Der Computer soll versuchen, den Programmteil innerhalb des `try`-Blocks auszuführen. Wenn jetzt eine Exception ausgelöst wird, setzt das Programm die Ausführung im nächsten `catch`-Block fort, der Exceptions vom Typ der ausgelösten Exception fangen kann.

Ein catch-Block beginnt mit dem Schlüsselwort `catch`, gefolgt von runden Klammern. In den runden Klammern steht ein Übergabeparameter vom Typ einer Exception. In unserem Fall vom Typ `FormatException`. Danach folgen geschweifte Klammern. In den geschweiften Klammern stehen die Anweisungen, die ausgeführt werden, wenn der catch-Block ausgelöst wird.

In unserem Beispiel wird die Programmausführung im catch-Block fortgesetzt, wenn eine Exception vom Typ `FormatException` ausgelöst wird.



The screenshot shows a terminal window titled "Microsoft Visual Studio-Debugging-Konsole". It displays the following interaction:

```
Geben Sie eine ganze Zahl ein:  
Hallo  
Geben Sie noch eine ganze Zahl ein:  
Welt  
Sie haben einen Text statt einer ganzen Zahl eingegeben.  
C:\Users\rrober\source\repos\CSsharpBuch\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (P  
rozess "9900") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >  
"Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 12.2.1 Tritt ein Fehler auf, geben wir jetzt eine verständliche Meldung aus

Wenn der Benutzer jetzt einen Text statt einer ganzen Zahl eingibt, wird eine `FormatException` ausgelöst und wir reagieren darauf mit einer benutzerfreundlichen Fehlermeldung und beenden das Programm kontrolliert.

Unser catch-Block reagiert spezifisch nur auf eine `FormatException`. Wenn eine andere Exception ausgelöst wird, reagiert unser catch-Block nicht und das Programm bricht unkontrolliert ab. Um das zu simulieren, fügen wir in unserem Beispielprogramm eine Division durch Null ein.

```
1  using System;  
2  
3  namespace FehlerBehandlung  
4  {  
5      class Program  
6      {  
7          static void Main(string[] args)  
8          {  
9              Console.WriteLine("Geben Sie eine ganze Zahl ein:");  
10             var eingabeZahl1 = Console.ReadLine();  
11  
12             Console.WriteLine("Geben Sie noch eine ganze Zahl  
13             ein:");  
14             var eingabeZahl2 = Console.ReadLine();  
15  
16             try  
17             {  
18                 var zahl1 = int.Parse(eingabeZahl1);  
19                 var zahl2 = int.Parse(eingabeZahl2);  
20  
21                 var zahl3 = zahl1 / 0;  
22             }  
23         }  
24     }  
25 }
```

## 12 Fehlerbehandlung mit Exceptions

```

23             Console.WriteLine($"\"{zahll1}+{zahl2}={zahll1 +
24                 zahl2}\"");
25         }
26         catch (FormatException ex)
27         {
28             Console.WriteLine("Sie haben einen Text statt
29                 einer ganzen Zahl eingegeben.");
30         }
31     }
32 }
33 }
```

```

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe
Geben Sie eine ganze Zahl ein:
8
Geben Sie noch eine ganze Zahl ein:
9

```

Abb. 12.2.2 Bei Eingabe zweier ganzer Zahlen wird keine FormatException ausgelöst.

Wenn wir zwei ganze Zahlen eingeben, wird keine FormatException ausgelöst.

```

1  using System;
2
3  namespace FehlerBehandlung
4  {
4.1    class Program
4.2    {
4.3        static void Main(string[] args)
4.4        {
4.5            Console.WriteLine("Geben Sie eine ganze Zahl ein:");
4.6            var eingabeZahl1 = Console.ReadLine();
4.7
4.8            Console.WriteLine("Geben Sie noch eine ganze Zahl ein:");
4.9            var eingabeZahl2 = Console.ReadLine();
5.0
5.1            try
5.2            {
5.3                var zahll1 = int.Parse(eingabeZahl1);
5.4                var zahll2 = int.Parse(eingabeZahl2);
5.5
5.6                var zahll3 = zahll1 / 0; ✖
5.7
5.8                Console.WriteLine($"\"{zahll1}/{zahll2}={zahll3}\"");
5.9            }
5.10           catch (FormatException ex)
5.11           {
5.12               Console.WriteLine("Sie haben einen Text statt
5.13                 einer ganzen Zahl eingegeben.");
5.14           }
5.15       }
5.16   }
5.17 }
```

Abb. 12.2.3 Eine Division durch Null löst eine DivideByZeroException aus

Die Programmzeile `var zahll3 = zahll1 / 0;` löst eine DevideByZeroException aus, auf die unser catch-Block nicht reagiert und das Programm bricht unkontrolliert ab.

Unser Beispielprogramm ist sehr überschaubar. In realistischen Szenarien werden Programme aber meistens sehr komplex und es kann zu Exceptions kommen, die vom Programmierer nicht vorhergesehen wurden. Solche Exceptions nennt man un-

erwartete Exceptions. Wenn wir unser Beispielprogramm wie folgt erweitern, können wir auch auf unerwartete Exceptions reagieren.

```
1  using System;
2
3  namespace FehlerBehandlung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
10             var eingabeZahl1 = Console.ReadLine();
11
12             Console.WriteLine("Geben Sie noch eine ganze Zahl
13             ein:");
14             var eingabeZahl2 = Console.ReadLine();
15
16             try
17             {
18                 var zahl1 = int.Parse(eingabeZahl1);
19                 var zahl2 = int.Parse(eingabeZahl2);
20
21                 var zahl3 = zahl1 / 0;
22
23                 Console.WriteLine($"{zahl1}+{zahl2}={zahl1 +
24                         zahl2}");
25             }
26             catch (FormatException ex)
27             {
28                 Console.WriteLine("Sie haben einen Text statt
29                         einer ganzen Zahl eingegeben.");
30             }
31             catch (Exception ex)
32             {
33                 Console.WriteLine($"Unerwarteter Fehler: {ex.
34                         Message}");
35             }
36         }
37     }
38 }
```

12

Im Anschluss an den `catch`-Block, der die `FormatException` abfängt, haben wir einen weiteren `catch`-Block hinzugefügt. Dieser reagiert auf Exceptions vom Typ `Exception`. Das heißt, er reagiert auf alle Exceptions. In diesem zweiten `catch`-Block geben wir die `Property Message` des jeweiligen `Exception`-Objekts aus.

## 12 Fehlerbehandlung mit Exceptions

The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the following text:

```

Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
34
Geben Sie noch eine ganze Zahl ein:
55
Unerwarteter Fehler: Attempted to divide by zero.

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (Prozess "12448") wurde mit Code "9" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 12.2.4** Ein catch-Block, der auf Exception reagiert, fängt auch die DivideByZeroException.

Wird innerhalb eines try-Blocks eine Exception ausgelöst, so werden die dem try-Block folgenden catch-Blöcke der Reihe nach durchsucht und, sobald ein passender catch-Block gefunden wird, wird die Programmausführung in diesem catch-Block fortgesetzt.

### 12.3 Exceptions kontrolliert auslösen

Eine Exception kann nicht nur durch eine Fehlersituation ausgelöst werden. Als Programmierer können wir sie auch kontrolliert auslösen. Wir modifizieren unser Beispielprogramm etwas, um diese Technik näher zu betrachten.

```

1  using System;
2
3  namespace FehlerBehandlung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              try
10              {
11                  Start();
12              }
13              catch(InvalidOperationException ex)
14              {
15                  Console.WriteLine(ex.Message);
16              }
17          }
18
19          static void Start()
20          {
21              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
22              var eingabeZahl1 = Console.ReadLine();
23
24              if(eingabeZahl1 == "")
25              {
26                  throw new InvalidOperationException("Sie müssen
27                  eine ganze Zahl eingeben.");
28              }
29

```

```
30         if (int.TryParse(eingabeZahl1, out int zahl1) ==  
31             false)  
32         {  
33             throw new InvalidOperationException  
34             ($"{eingabeZahl1} ist keine ganze Zahl.");  
35         }  
36  
37         Console.WriteLine("Geben Sie noch eine ganze Zahl  
38         ein:");  
39         var eingabeZahl2 = Console.ReadLine();  
40  
41         if (eingabeZahl2 == "")  
42         {  
43             throw new InvalidOperationException("Sie müssen  
44             noch eine ganze Zahl eingeben.");  
45         }  
46  
47         if (int.TryParse(eingabeZahl2, out int zahl2) ==  
48             false)  
49         {  
50             throw new InvalidOperationException  
51             ($"{eingabeZahl2} ist keine ganze Zahl.");  
52         }  
53  
54         Console.WriteLine($"{zahl1}+{zahl2}={(zahl1 +  
55             zahl2)}");  
56     }  
57 }  
58 }
```

Wir lagern unseren Programmcode der Hauptmethode in die statische Methode `Start` aus. Nach jeder Eingabe überprüfen wir, ob der Benutzer eine für unser Programm verwertbare Eingabe gemacht hat. Entspricht die Eingabe nicht unseren Erwartungen, so werfen wir eine Exception.

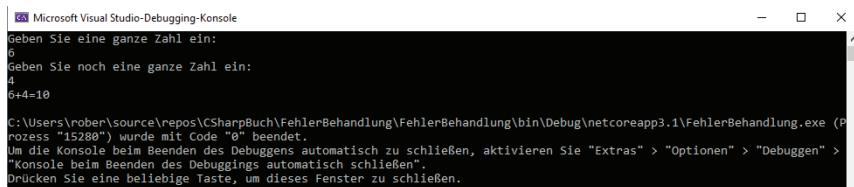
12

Das Werfen einer Exception erfolgt mit dem Schlüsselwort `throw`, gefolgt von der Instanz einer `Exception`-Klasse. `Exception`-Klassen sind alle Klassen, die entweder direkt oder indirekt von der Klasse `Exception` abgeleitet sind. In unserem Beispiel verwenden wir eine anonyme Instanz der Klasse `InvalidOperationException` und übergeben ihr im Konstruktor die Fehlermeldung, die dann über die Property `Message` der `Exception`-Instanz abgerufen werden kann.

Theoretisch könnten wir jede der vielen vom .NET-Framework bereitgestellten Ableitungen der Klasse `Exception` verwenden, aber `InvalidOperationException` trifft den Sachverhalt unseres Beispielprogramms am besten.

In der Hauptmethode rufen wir die Methode `Start` innerhalb eines `try`-Blocks auf. Im zugehörigen `catch`-Block fangen wir die Exceptions vom Typ `InvalidOperationException` ab und geben die zugehörige Property `Message` am Bildschirm aus.

## 12 Fehlerbehandlung mit Exceptions

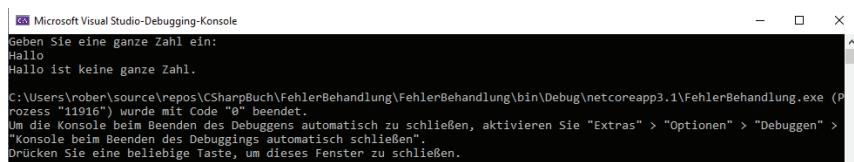


```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
6
Geben Sie noch eine ganze Zahl ein:
4
6+4=10

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (P
rozess "15288") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 12.3.1** Bei Eingabe von ganzen Zahlen kommt es zu keinem Fehler

Wenn wir zwei ganze Zahlen eingeben, funktioniert unser Beispiel ohne Fehlermeldung.



```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
Hallo
Hallo ist keine ganze Zahl.

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (P
rozess "1916") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 12.3.2** Bei einer Fehleingabe erhalten wir die entsprechende Fehlermeldung

Geben wir, statt einer ganzen Zahl, einen Text ein, wird unser Beispielprogramm kontrolliert, mit einer benutzerfreundlichen Fehlermeldung, beendet.

### 12.4 Eigene Exceptions definieren

Alle vom .NET-Framework zur Verfügung gestellten Exceptions sind von der Klasse `Exception` abgeleitet, daher können wir auch eigene Klassen von `Exception` ableiten. Um in unserem Beispielprogramm unterscheiden zu können, ob es sich bei einer Exception um eine vom Programmierer geworfene Exception handelt oder ob eine Exception durch eine Fehlersituation ausgelöst wurde, definieren wir uns unsere eigene Ableitung von `Exception`.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.Serialization;
4  using System.Text;
5
6  namespace FehlerBehandlung
7  {
8      public class FalscheEingabeException : Exception
9      {
10          public FalscheEingabeException() : base("Es wurde eine")
11          falsche Eingabe gemacht.")
12          {
13          }
14
15          public FalscheEingabeException(string message) :
```

```
16     base(message)
17     {
18     }
19
20     public FalscheEingabeException(string message, Exception
21     innerException) : base(message, innerException)
22     {
23     }
24
25     public FalscheEingabeException(SerializationInfo info,
26     StreamingContext context) : base(info, context)
27     {
28     }
29   }
30 }
```

Unsere eigene Exception `FalscheEingabeException` ist von der Klasse `Exception` abgeleitet. Die Klasse `Exception` besitzt vier Konstruktoren, die wir in unserer abgeleiteten `Exception`-Klasse alle überschreiben müssen, um zu gewährleisten, dass unsere eigene Exception in allen Situationen genauso funktioniert wie eine vom .NET-Framework zur Verfügung gestellte Exception.

Der erste Konstruktor ist der parameterlose Konstruktor. Beim Überschreiben rufen wir den Konstruktor der Basisklasse `Exception`, der einen String mit einer Fehlermeldung entgegennimmt und übergeben ihm als Standard-Fehlermeldung den Text „Es wurde eine falsche Eingabe gemacht.“

Den zweiten Konstruktor überschreiben wir und rufen nur seine Entsprechung der Basisklasse auf, ohne eigene Funktionalität hinzuzufügen. Wir benötigen diesen Konstruktor, um eine Instanz unserer eigenen Exception mit einer individuellen Fehlermeldung erzeugen zu können.

Im dritten Konstruktor übernehmen wir ebenfalls nur die Funktionalität der Basisklasse. Dieser Konstruktor nimmt noch einen Parameter vom Typ `Exception` entgegen. Diese Exception wird in unserer eigenen Exception als sogenannte innere Exception verwendet. Wozu innere Exceptions benötigt werden, werden wir im nächsten Beispiel betrachten.

Der letzte Konstruktor implementiert auch nur die Funktionalität der Basisklasse. Er wird nur benötigt, wenn unsere Exception in einem Programm mit Client-Server-Kommunikation verwendet wird und dort vom Server zum Client transportiert werden soll. Der Vollständigkeit halber sollten wir ihn aber immer implementieren, wenn wir eine eigene `Exception`-Klasse definieren. Auf Client-Server Kommunikation wird jedoch nicht weiter eingegangen, da es den Rahmen dieses Buchs sprengen würde.

## 12 Fehlerbehandlung mit Exceptions

Zum Abschluss der Kapitel über Exceptions verwenden wir unsere selbst definierte Exception in unserem Beispielprogramm und sehen uns auch näher an, was eine innere Exception ist. Innere Exceptions treten genau dann auf, wenn der Programmierer eine Exception mit einen catch-Block fängt, und dann via Programmcode eine andere Exception wirft. Wenn die vom Programmierer geworfene Exception später von einem anderen catch-Block gefangen wird, ist die ursprüngliche Exception zunächst verloren. Allerdings kann der Programmierer dafür sorgen, dass die ursprüngliche Exception in der geworfenen Exception als sog. innere Exception erhalten bleibt. Wie das geht, werden wir im Weiteren sehen.

```
1  using System;
2
3  namespace FehlerBehandlung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              try
10             {
11                 Start();
12             }
13             catch(FalscheEingabeException ex)
14             {
15                 if (ex.InnerException == null)
16                 {
17                     Console.WriteLine(ex.Message);
18                 }
19                 else
20                 {
21                     Console.WriteLine($"{ex.Message} {ex.
22                         InnerException.Message}");
23                 }
24             }
25         }
26
27         static void Start()
28         {
29             Console.WriteLine("Geben Sie eine ganze Zahl ein:");
30             var eingabeZahl1 = Console.ReadLine();
31
32             if (eingabeZahl1 == "")
33             {
34                 throw new FalscheEingabeException("Sie müssen
35                     eine ganze Zahl eingeben.");
36             }
37
38             if (int.TryParse(eingabeZahl1, out int zahl1) ==
39                 false)
40             {
41                 throw new FalscheEingabeException
42                     ("'{eingabeZahl1}' ist keine ganze Zahl.");
43             }
44         }
45     }
```

```
45     Console.WriteLine("Geben Sie noch eine ganze Zahl  
46     ein:");
47     var eingabeZahl2 = Console.ReadLine();
48
49     if (eingabeZahl2 == "")  

50     {
51         throw new FalscheEingabeException("Sie müssen  
52         noch eine ganze Zahl eingeben.");
53     }
54
55     if (int.TryParse(eingabeZahl2, out int zahl2) ==  
56     false)
57     {
58         throw new FalscheEingabeException  

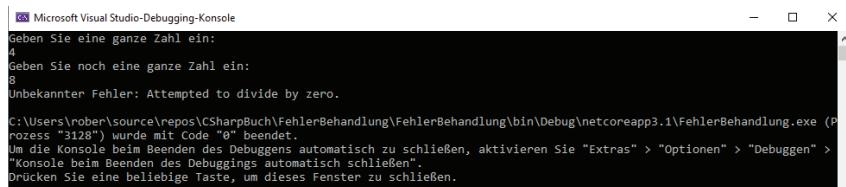
59         ($"{eingabeZahl2} ist keine ganze Zahl.");
60     }
61
62     try
63     {
64         var zahl3 = zahl1 / 0;
65         Console.WriteLine($"{zahl1}+{zahl2}={zahl1 +  
66         zahl2}");
67     }
68     catch(Exception ex)
69     {
70         throw new FalscheEingabeException("Unbekannter  
71         Fehler:", ex);
72     }
73 }
74 }
75 }
```

Statt der `InvalidOperationException` werfen wir jetzt unsere selbst definierte Exception: `FalscheEingabeException`, wenn wir Eingaben erhalten, die von unserem Programm nicht verwendet werden können.

Die Berechnung der Summe der beiden eingegebenen Zahlen und die Ausgabe des Ergebnisses packen wir in einen `try`-Block. Des Weiteren enthält der `try`-Block eine Zeile, die eine Division durch Null durchführt, um einen unerwarteten Fehler zu simulieren. Im zugehörigen `catch`-Block werfen wir eine `FalscheEingabeException` mit der Fehlermeldung „Unbekannter Fehler.“. Zudem übergeben wir dem Konstruktor der Klasse `FalscheEingabeException` die Variable `ex`, also die `Exception`, die ursprünglich ausgelöst wurde. Damit haben wir eine Instanz der Klasse `FalscheEingabeException` mit einer inneren `Exception` erzeugt.

In der Hauptmethode fangen wir alle Exceptions vom Typ `FalscheEingabeException` ab. Wir überprüfen die Property `InnerException`. Wenn sie `null` ist, gibt es keine innere Exception und wir geben einfach nur die Property `Message` am Bildschirm aus. Ist `InnerException` nicht `null`, geben wir die Property `Message` der Exception und auch die Property `Message` der inneren Exception am Bildschirm aus.

## 12 Fehlerbehandlung mit Exceptions



```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
4
Geben Sie noch eine ganze Zahl ein:
8
Unbekannter Fehler: Attempted to divide by zero.

C:\Users\rober\source\repos\CSharpBuch\FehlerBehandlung\FehlerBehandlung\bin\Debug\netcoreapp3.1\FehlerBehandlung.exe (Prozess "3128") wurde mit Code "9" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 12.4.1 Bildschirmausgabe der Exception und der inneren Exception

### 12.5 Übungsaufgabe: Ein Programm mit Fehlerbehandlung erstellen

In dieser Übung schreiben wir ein einfaches Programm, das das Quadrat für eine eingegebene Zahl berechnet. Dabei werden wir eine selbst definierte Exception verwenden, um Eingabefehler des Benutzers zu behandeln.

#### Teilaufgabe 1:

Definieren Sie die Exception `EingabeFehlerException`. Implementieren Sie alle Konstruktoren der Basisklasse `Exception`, sodass die Konstruktoren lediglich die Funktionalität der Konstruktoren der Basisklasse `Exception` bereitstellen.

#### Teilaufgabe 2:

Fügen Sie dem Hauptprogramm die statische Membervariable `fertig` vom Typ `bool` und die statische Methode `LeseZahlEin()` hinzu. Die Methode hat keinen Übergabeparameter und gibt einen Wert vom Typ `int` zurück. Die Methode liest eine ganze Zahl als Benutzereingabe ein und gibt diese zurück. Wenn der Benutzer den Text „fertig“ eingibt, soll die Methode `LeseZahlEin()`, die Membervariable `fertig` auf `true` setzen und die Zahl Null zurückgeben. Falls der Benutzer etwas anderes als eine ganze Zahl oder den Text „fertig“ eingibt, wirft die Methode eine Exception vom Typ `EingabeFehlerException`.

#### Teilaufgabe 3:

Schreiben Sie das Hauptprogramm. Lesen Sie in einer Schleife mit Hilfe der Methode `LeseZahlEin()` eine ganze Zahl ein und geben Sie dann das Quadrat dieser Zahl am Bildschirm aus. Wenn der Benutzer den Text fertig eingibt, soll die Schleife abbrechen und das Programm soll beendet werden. Wenn die Methode `LeseZahlEin()` eine Exception vom Typ `EingabeFehlerException` wirft, soll diese abgefangen werden und die Property `Message` der Exception soll am Bildschirm ausgegeben werden. Beachten Sie bitte, dass das Programm nach einer Fehleingabe weiterlaufen und wieder eine Zahl abfragen soll.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.Serialization;
4  using System.Text;
5
6  namespace UebungExcptions
7  {
8      public class EingabeFehlerException : Exception
9      {
10         public EingabeFehlerException() :base()
11         {
12         }
13
14         public EingabeFehlerException(string
15             message) :base(message)
16         {
17         }
18
19         public EingabeFehlerException(string message, Exception
20             inner) : base(message, inner)
21         {
22         }
23
24         public EingabeFehlerException(SerializationInfo info,
25             StreamingContext context) :base(info, context)
26         {
27         }
28     }
29 }
```

## 12 Fehlerbehandlung mit Exceptions

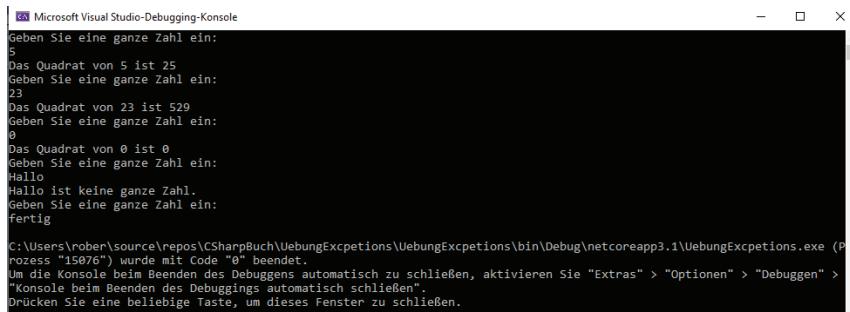
**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2
3  namespace UebungExcpetions
4  {
5      class Program
6      {
7          static bool fertig;
8          static void Main(string[] args)
9          {
10          }
11
12          static int LeseZahlEin()
13          {
14              Console.WriteLine("Geben Sie eine ganze Zahl ein:");
15              var eingabeZahl = Console.ReadLine();
16              if(eingabeZahl == "fertig")
17              {
18                  fertig = true;
19                  return 0;
20              }
21
22              if (int.TryParse(eingabeZahl, out int zahl) ==
23                  false)
24              {
25                  throw new EingabeFehlerException($"(eingabeZahl)
26                  ist keine ganze Zahl.");
27              }
28
29              return zahl;
30          }
31      }
32 }
```

**Musterlösung für Teilaufgabe 3**

```
1 using System;
2
3 namespace UebungExcpetions
4 {
5     class Program
6     {
7         static bool fertig;
8         static void Main(string[] args)
9         {
10            while (!fertig)
11            {
12                try
13                {
14                    var zahl = LeseZahlEin();
15
16                    if (!fertig)
17                    {
18                        Console.WriteLine($"Das Quadrat von
19                        {zahl} ist {zahl * zahl}");
20                    }
21                }
22                catch(EingabeFehlerException ex)
23                {
24                    Console.WriteLine(ex.Message);
25                }
26            }
27        }
28
29        static int LeseZahlEin()
30        {
31            Console.WriteLine("Geben Sie eine ganze Zahl ein:");
32            var eingabeZahl = Console.ReadLine();
33            if(eingabeZahl == "fertig")
34            {
35                fertig = true;
36                return 0;
37            }
38
39            if (int.TryParse(eingabeZahl, out int zahl) ==
40            false)
41            {
42                throw new EingabeFehlerException($"{eingabeZahl}
43                ist keine ganze Zahl.");
44            }
45
46            return zahl;
47        }
48    }
49 }
```

## 12 Fehlerbehandlung mit Exceptions



```
Microsoft Visual Studio-Debugging-Konsole
Geben Sie eine ganze Zahl ein:
5
Das Quadrat von 5 ist 25
Geben Sie eine ganze Zahl ein:
23
Das Quadrat von 23 ist 529
Geben Sie eine ganze Zahl ein:
0
Das Quadrat von 0 ist 0
Geben Sie eine ganze Zahl ein:
Hallo
Hallo ist keine ganze Zahl.
Geben Sie eine ganze Zahl ein:
fertig

C:\Users\rober\source\repos\CSharpBuch\UebungExcptions\UebungExcptions\bin\Debug\netcoreapp3.1\UebungExcptions.exe (Prozess "15076") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 12.5.1 Bildschirmausgabe der Übung für Exceptions

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 13

# Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

In diesem Kapitel sprechen wir nicht über neue Features der Programmiersprache C# und auch nicht über neue Klassenbibliotheken des .NET-Frameworks, sondern über Funktionalitäten von Visual Studio als Entwicklungsumgebung. Diese Funktionalitäten werden umso wichtiger, je größer und komplexer unsere Programmierprojekte werden. Alle Funktionalitäten von Visual Studio vollständig zu beschreiben, würde den Rahmen dieses Buches sprengen. Daher beschränke ich mich hier auf die wichtigsten. In den nächsten Unterkapiteln betrachten wir die folgenden Features von Visual Studio:

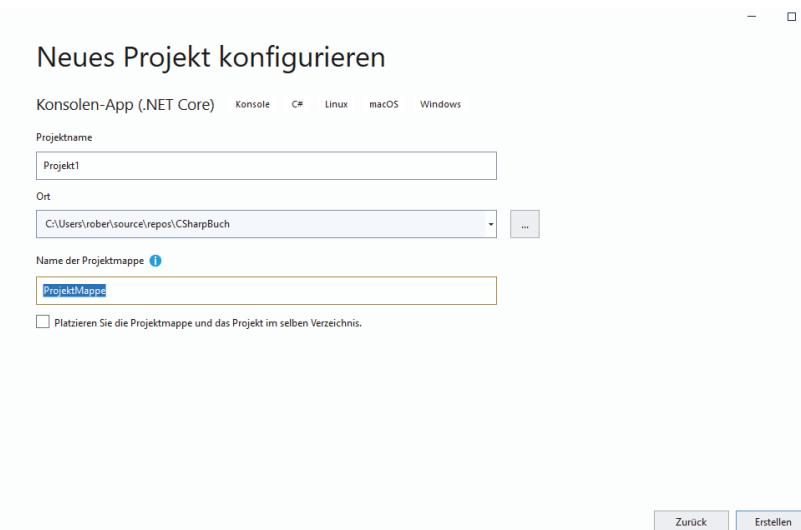
- ▶ Gemeinsames laden von mehreren Projekten
- ▶ Eigene Klassenbibliotheken
- ▶ Navigieren durch umfangreiche Programmcodes
- ▶ Verwenden des integrierten Debuggers von Visual Studio

### 13.1 Ein weiteres Projekt zur Projektmappe hinzufügen

Bisher haben wir in Visual Studio immer nur ein einziges Projekt geöffnet, welches unser Programm repräsentiert hat. Beim Erstellen eines neuen Projektes haben wir immer stillschweigend in Kauf genommen, dass Visual Studio eine sogenannte Projektmappe mit dem gleichen Namen wie das Projekt mit angelegt hat. Projektmappen sind Ordner, die mehrere Projekte enthalten können. Öffnen wir in Visual Studio eine Projektmappe, so werden alle Projekte der Projektmappe geöffnet. Im Projektmappen Explorer von Visual Studio wird als Wurzelknoten des Projektbaums eine Projektmappe angezeigt und als Kind Knoten der Projektmappe ein Projekt oder mehrere Projekte.

Als nächstes wollen wir eine Projektmappe anlegen, die zwei Projekte enthält. Starten Sie Visual Studio und erstellen Sie ein neues Projekt vom Typ Konsolen-App (.NET Core):

## 13.1 Ein weiteres Projekt zur Projektmappe hinzufügen



**Abb. 13.1.1** Erstellen einer Projektmappe mit einem Projekt

Verwenden Sie „Projekt1“ als Projektname. Visual Studio schlägt ihnen vor, auch die Projektmappe „Projekt1“ zu nennen. Vergeben sie stattdessen den Namen „Projekt-Mappe“ für die Projektmappe, da diese Projektmappe zwei Projekte enthalten wird. Klicken Sie auf die Schaltfläche „Erstellen“.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

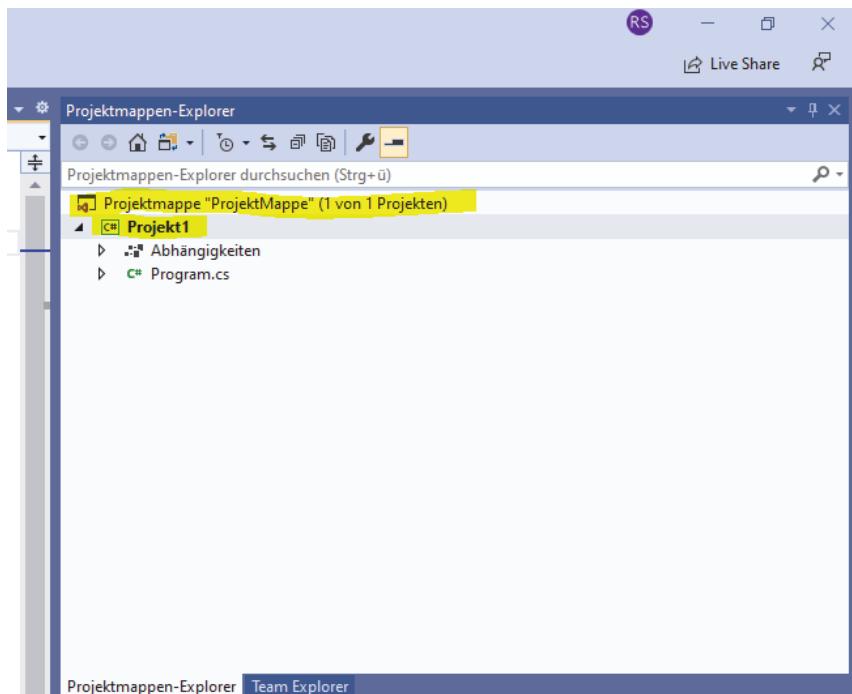


Abb. 13.1.2 Die Projektmappe „Projektmappe“ enthält das Projekt „Projekt1“

Wie zu erwarten war, hat Visual Studio ein „Hello Word“ Projekt mit dem Namen „Projekt1“ erstellt, das unter einer Projektmappe, die „ProjektMappe“ heißt, angezeigt wird.

Als nächste fügen wir der Projektmappe ein zweites Projekt hinzu. Dazu klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf die Projektmappe.

## 13.1 Ein weiteres Projekt zur Projektmappe hinzufügen

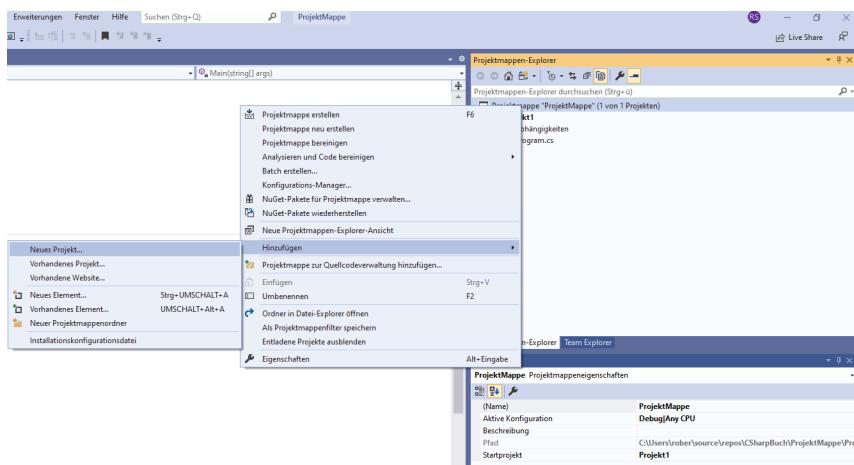


Abb. 13.1.3 Ein weiteres Projekt hinzufügen

Wählen Sie das Kontextmenü „Hinzufügen\Neues Projekt...“ aus.

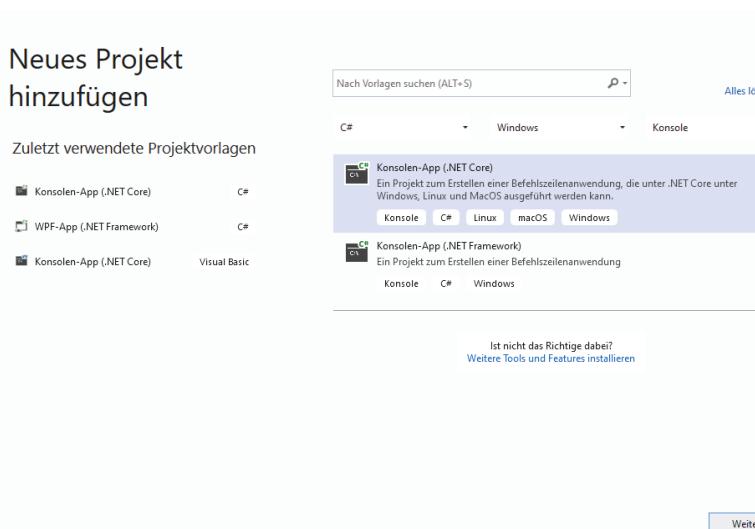


Abb. 13.1.4 Auswählen des Projekttyps

Wählen Sie als Projekttyp „Konsolen-App (.NET Core)“ aus und klicken Sie auf die Schaltfläche „Weiter“.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

## Neues Projekt konfigurieren

Konsolen-App (.NET Core) Konsole C# Linux macOS Windows

Projektname

Projekt2

Ort

C:\Users\rober\source\repos\CSharpBuch\ProjektMappe



Zurück

Erstellen

Abb. 13.1.5 Ein zweites Projekt erstellen

Vergeben Sie „Projekt2“ als Projektname und klicken Sie auf die Schaltfläche erstellen.

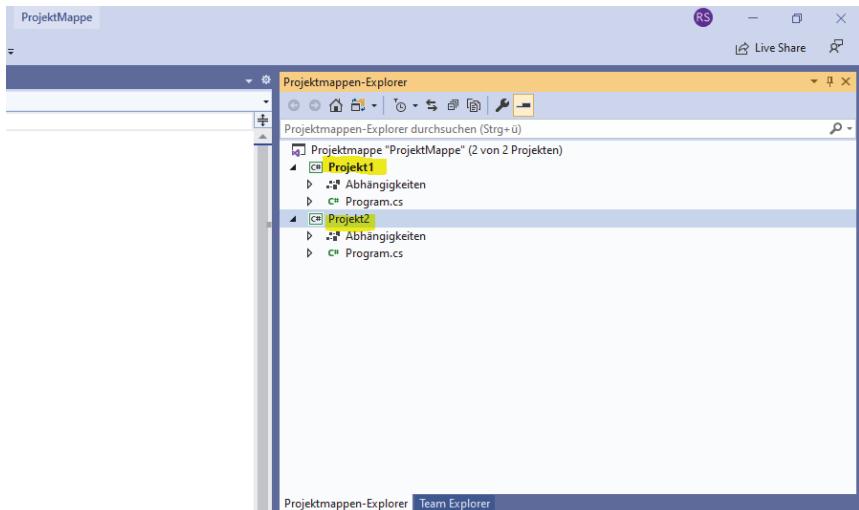


Abb. 13.1.6 Eine Projektmappe mit zwei Projekten

## 13.1 Ein weiteres Projekt zur Projektmappe hinzufügen

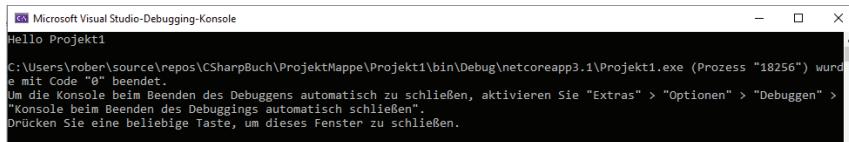
Jetzt haben wir eine Projektmappe erstellt, in der sich zwei „Hello World!“-Programme befinden. Damit wir die beiden besser unterscheiden können, ändern wir in Projekt1 die Bildschirmausgabe zu

```
1 Console.WriteLine("Hello Projekt1");
```

und in Projekt2 zu.

```
1 Console.WriteLine("Hello Projekt2");
```

Wenn wir jetzt in Visual Studio auf das grüne Dreieck zum Starten des aktuellen Programms klicken, startet das Programm Projekt1.



Das liegt daran, dass wir Projekt1 zuerst erstellt haben und deswegen Projekt1 als Startprojekt in unserer Projektmappe festgelegt wurde. Wenn wir Projekt2 starten wollen, müssen wir Projekt2 als Startprojekt festlegen. Dazu klicken wir mit der rechten Maustaste auf Projekt2.

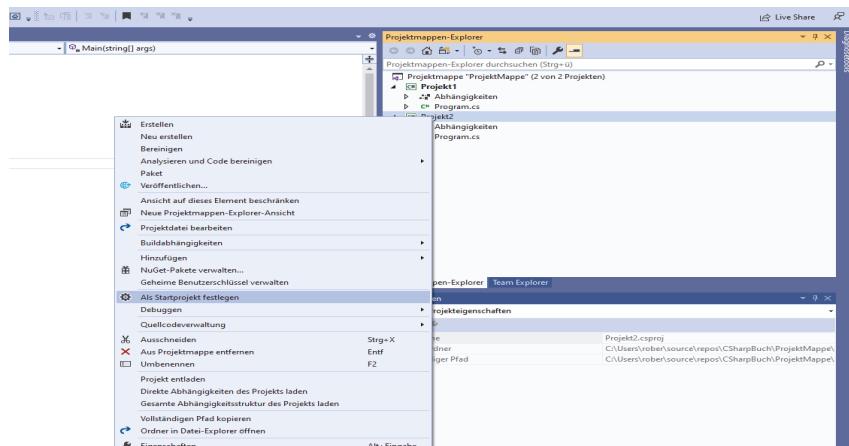
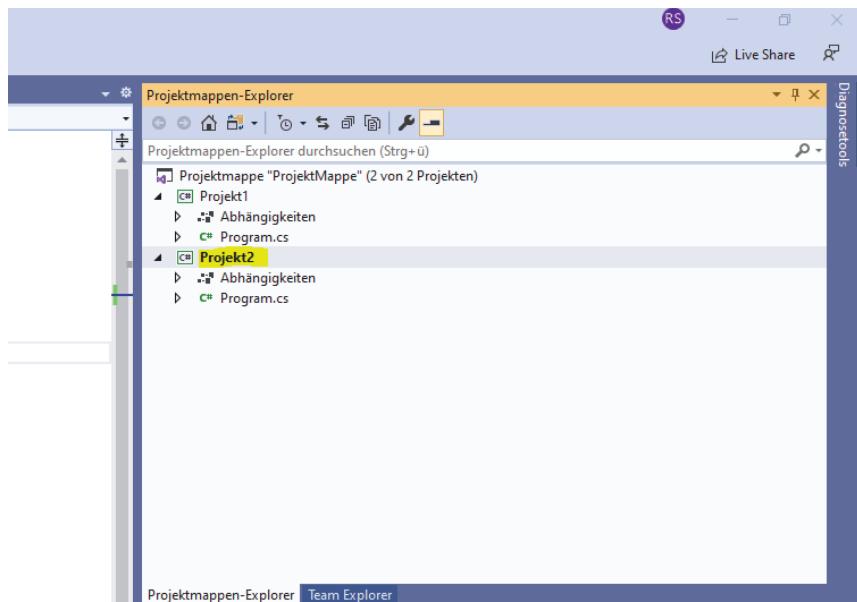


Abb. 13.1.7 Projekt2 wird als Startprojekt festgelegt

Im Kontextmenü klicken wir auf „Als Startprojekt festlegen“.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene



Jetzt ist nicht mehr Projekt1 fett dargestellt, sondern Projekt2. Mit einem Klick auf das grüne Dreieck können wir jetzt Projekt2 starten.

```
Microsoft Visual Studio-Debugging-Konsole
Hello Projekt2
C:\Users\rober\source\repos\CSharpBuch\ProjektMappe\Projekt2\bin\Debug\netcoreapp3.1\Projekt2.exe (Prozess "9728") wurde
mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 13.1.8 Projekt2 wird gestartet

Visual Studio erlaubt uns auch beide Projekte gleichzeitig zu starten. Dazu klicken wir mit der rechten Maustaste auf die Projektmappe.

## 13.1 Ein weiteres Projekt zur Projektmappe hinzufügen

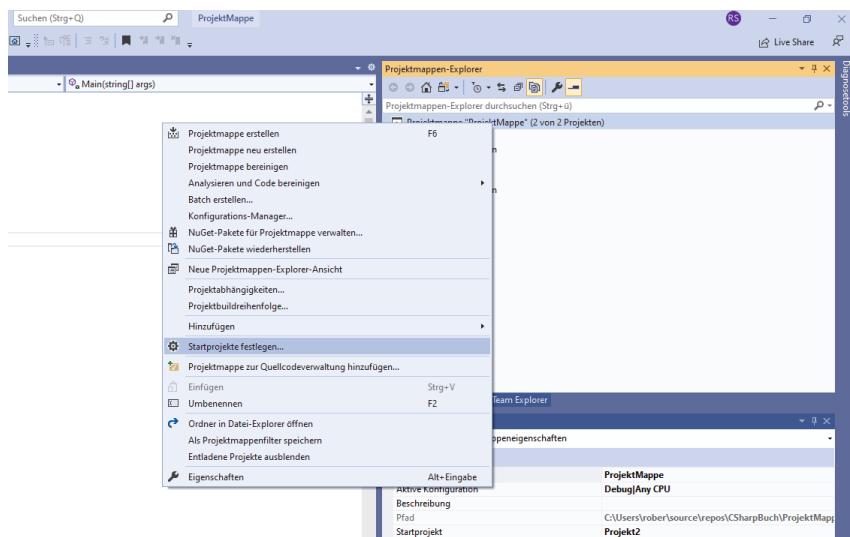


Abb. 13.1.9 Startprojekte einer Projektmappe festlegen

Aus dem Kontextmenü wählen wir „Startprojekte festlegen...“ aus.

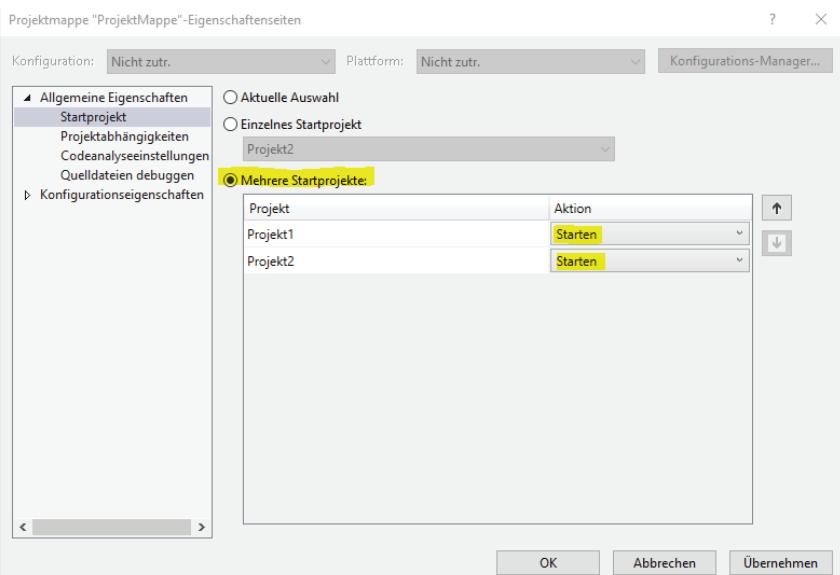


Abb. 13.1.10 Mehrere Startprojekte auswählen

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Wählen Sie die Radioschaltfläche „Mehrere Startprojekte“ aus und wählen Sie für Projekt1 und Projekt2 die Aktion „Starten“ aus. Danach Klicken Sie auf die Schaltfläche „OK“.

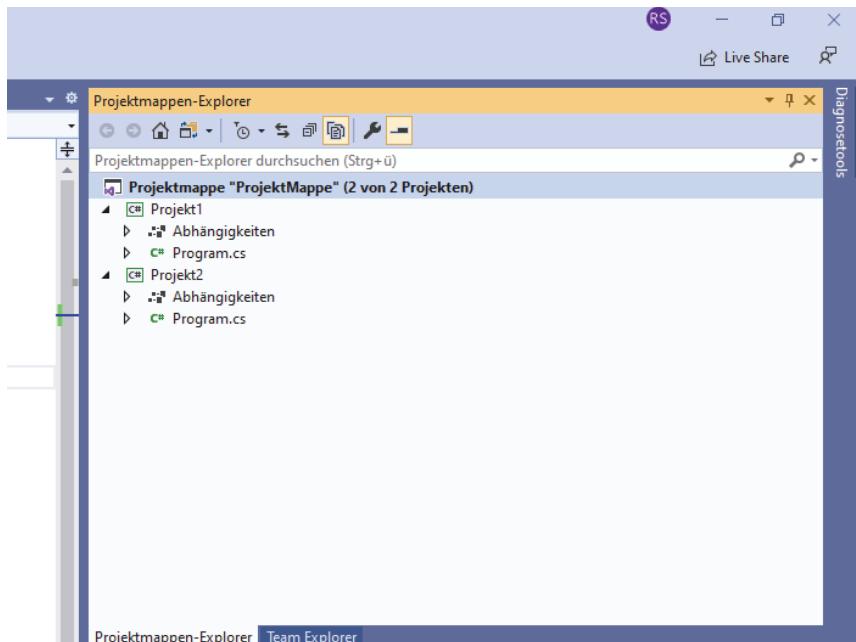


Abb. 13.1.11 Eine Projektmappe mit zwei Startprojekten

Jetzt ist keines unserer beiden Projekte fett dargestellt, da beide gleichzeitig gestartet werden. Damit wir auch sehen können, dass beide Projekte gestartet werden, ergänzen wir in den Hauptmethoden der beiden Programme die Anweisung:

```
1 Console.ReadLine();
```

Jetzt können wir mit einem Klick auf die das grüne Dreieck beide Programme starten.

### 13.2 Eine Klassenbibliothek erstellen

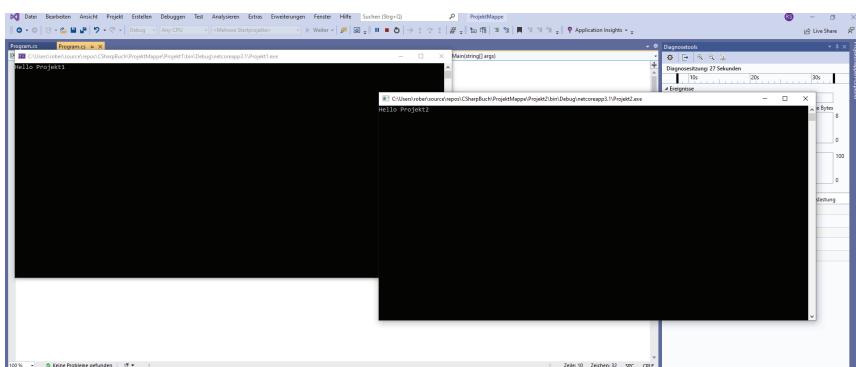
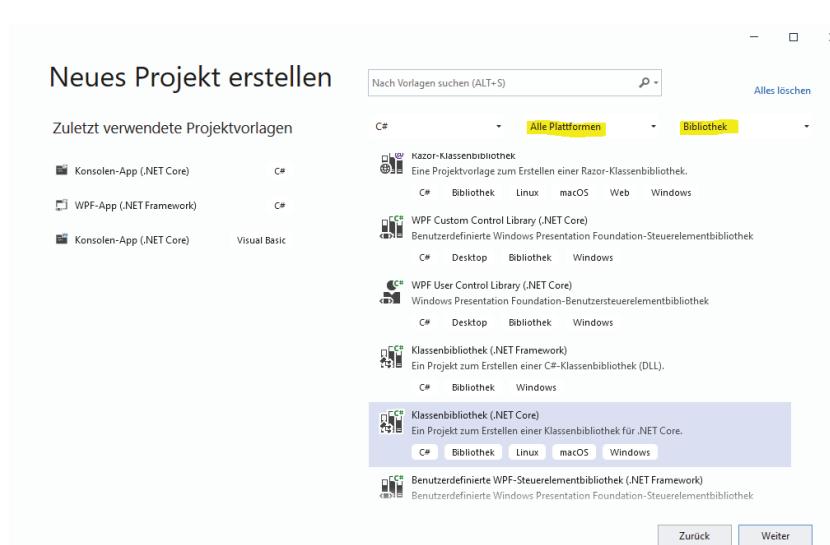


Abb. 13.1.12 Visual Studio startet zwei Programme gleichzeitig.

## 13.2 Eine Klassenbibliothek erstellen

In diesem Unterkapitel beschäftigen wir uns mit Klassenbibliotheken. Wir haben Klassenbibliotheken bisher nur als Anwender kennengelernt. Das heißt, wir haben Klassenbibliotheken verwendet, die andere programmiert haben. Diese Klassenbibliotheken wurden mit C# oder einer anderen .NET-Programmiersprache erstellt und kompiliert. Natürlich verfügt Visual Studio auch über die Möglichkeit, eine Klassenbibliothek zu erstellen, die wir dann in unseren Programmen verwenden oder anderen Programmierern zu Verfügung stellen können. Um eine eigene Klassenbibliothek zu erstellen, starten wir Visual Studio und wählen „Neues Projekt erstellen“ aus.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene



**Abb. 13.2.1** Auswahl des Projekttyps Klassenbibliothek

Als Filter für die Sprache wählen wir C# aus, beim Plattform-Filter wählen wir „Alle Plattformen“ und als Projekttyp-Filter wählen wir Bibliothek. Dann wählen wir den Projekttyp „Klassenbibliothek (.NET Core)“ aus. .NET Core ist die neueste und modernste Variante des .NET-Frameworks, es unterstützt die Betriebssysteme Linux, macOS und Windows.

Klicken sie auf Schaltfläche „Weiter“.

### Neues Projekt konfigurieren

Klassenbibliothek (.NET Core) C# Bibliothek Linux macOS Windows

Projektname

Ort

Name der Projektmappe

Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis.

**Abb. 13.2.2** Eine Klassenbibliothek erstellen

Als Projektnamen vergeben Sie „Mathematik“ und als Namen für die Projektmappe vergeben Sie „EigeneKlassenbibliotheken“. Klicken Sie auf die Schaltfläche „Erstellen“.

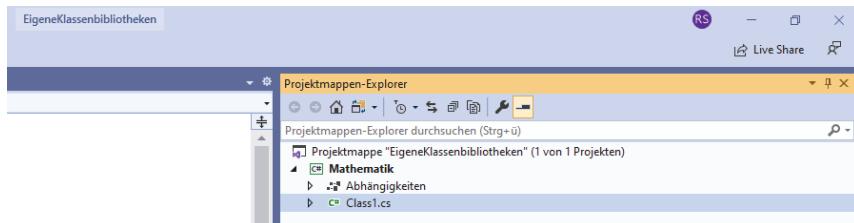


Abb. 13.2.3 Eine Projektmappe mit einer Klassenbibliothek

Visual Studio erstellt eine Projektmappe mit dem Namen „EigeneKlassenbibliotheken“ und unterhalb der Projektmappe ein Projekt mit dem Namen „Mathematik“. Dieses Projekt ist vom Typ Klassenbibliothek, daher hat Visual Studio auch keine Klasse **Program**, sondern eine Klasse mit dem Namen **Class1** erstellt. Der Name **Class1** ist wenig aussagekräftig, daher werden wir ihn ändern. Dazu klicken wir im Projekt-Explorer mit der rechten Maustaste auf die Klasse **Class1**.

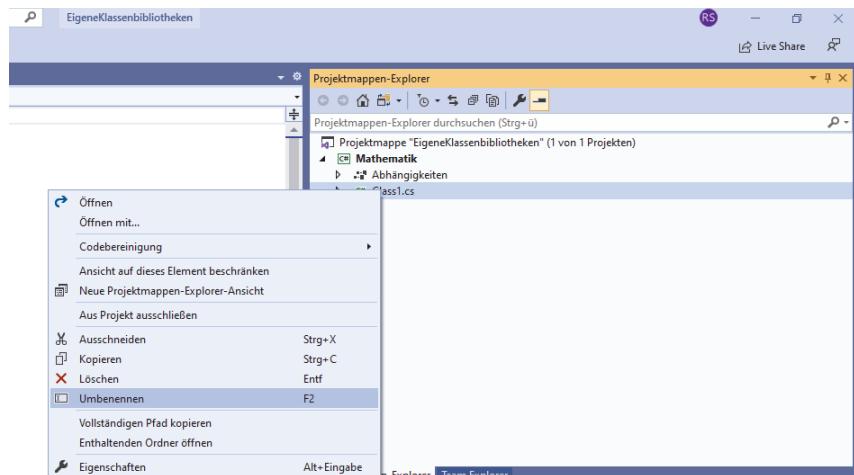


Abb. 13.2.4 Umbenennen einer Programmdatei

Im Kontextmenü klicken wir auf „Umbenennen“. Den Namen **Class1** ersetzen wir durch den Namen **Rechner** und drücken die Enter-Taste. Im darauffolgenden Dialogfenster werden wir gefragt, ob wir den Namen der Klasse **Class1** auch im Programmcode ändern möchten. Wir bestätigen den Dialog mit der Schaltfläche „Ja“.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Die Klasse, die von Visual Studio erzeugt wurde, besteht lediglich aus dem Rumpf der Klasse ohne Konstruktor, Membervariablen oder Methoden.

Als nächstes ändern wir die Klasse wie folgt ab:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Mathematik
6  {
7      public class Rechner
8      {
9          public double Addiere(double a, double b)
10         {
11             return a + b;
12         }
13
14         public double Subtrahiere(double a, double b)
15         {
16             return a - b;
17         }
18
19         public double Multpliziere(double a, double b)
20         {
21             return a * b;
22         }
23
24         public double Dividiere(double a, double b)
25         {
26             return a / b;
27         }
28     }
29 }
30 }
```

Die Klasse `Rechner` ist trivial. Sie besitzt für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division jeweils eine Methode, die jeweils zwei Parameter vom Typ `double` entgegennimmt, die jeweilige Grundrechenart mit den Parametern ausführt und das Ergebnis als `double` zurückgibt.

Eine Klassenbibliothek kann nicht gestartet werden. Wenn wir es trotzdem versuchen, erhalten wir eine Fehlermeldung. Eine Klassenbibliothek kann nur von einem anderen Programm verwendet werden. Wie wir unsere selbst erstellte Klassenbibliothek verwenden können, werden wir im nächsten Unterkapitel sehen.

### 13.3 Die eigene Klassenbibliothek verwenden

Um die Klassenbibliothek, die wir im vorherigen Unterkapitel erstellt haben, zu verwenden, benötigen wir ein Programm. Zunächst werden wir unsere Klassenbiblio-

thek in einem Programm verwenden, das sich in der gleichen Projektmappe wie die Klassenbibliothek befindet. Dazu öffnen wir die Projektmappe „EigeneKlassenbibliotheken“ mit Visual Studio und fügen der Projektmappe, wie im Unterkapitel 13.1 gezeigt, ein weiteres Projekt hinzu. Als Projekttyp wählen wir Konsolen-App (.NET Core) aus und als Projektnamen vergeben wir „TestMathematik“.

Unsere Projektmappe enthält jetzt zwei Projekte: Die Klassenbibliothek Mathematik und die Konsolenapplikation TestMathematik. Damit die Konsolenapplikation die Klassenbibliothek verwenden kann, müssen wir ihr einen sogenannten Projektverweis hinzufügen. Dazu klicken wir mit der rechten Maustaste auf den Knoten „Abhängigkeiten“, der sich unterhalb des Projekts „TestMathematik“ befindet.

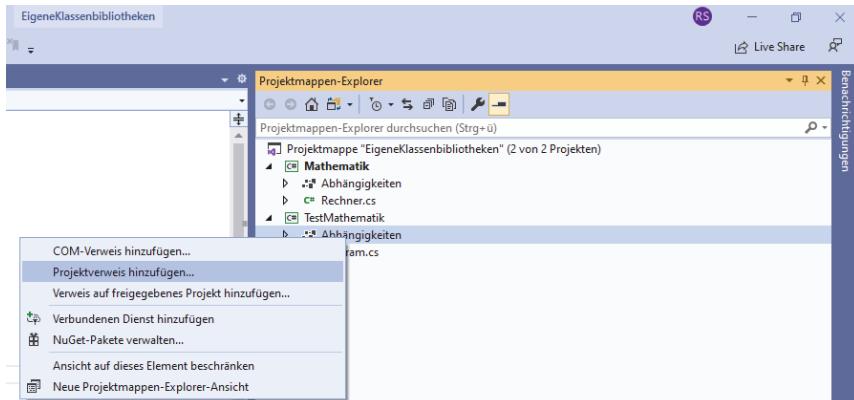


Abb. 13.3.1 Das Kontextmenü Projektverweis hinzufügen ...

Im Kontextmenü wählen wir „Projektverweis hinzufügen...“ aus.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

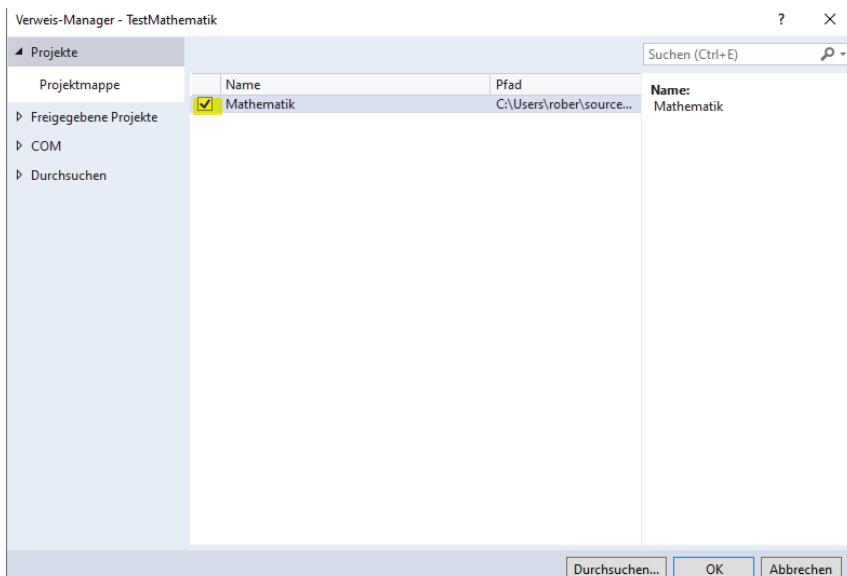


Abb. 13.3.2 Projektverweis auf die Klassenbibliothek Mathematik hinzufügen

Wir markieren die Checkbox neben der Klassenbibliothek Mathematik mit einem Haken und klicken auf die Schaltfläche OK.

In unserer Konsolenapplikation können wir die Klassenbibliothek Mathematik genauso verwenden wie die Klassenbibliothek System aus dem .NET-Framework. Dazu müssen wir nur in der Klasse `Program` eine `using`-Anweisung einfügen. Die `using`-Anweisung für die Klassenbibliothek Mathematik heißt:

```
1 using Mathematik;
```

Jetzt könnte man meinen, der Name nach der `using`-Anweisung ist der Name der Klassenbibliothek. Aber es handelt sich hierbei um den Namespace Mathematik. Wenn wir noch mal den Programmcode der Klasse `Rechner` betrachten, sehen wir, dass der Namespace der Klasse `Rechner` Mathematik heißt.

Falls Sie planen, eine Klassenbibliothek zu veröffentlichen, müssen Sie damit rechnen, dass ein anderer Programmierer auch eine Klassenbibliothek veröffentlicht, die den gleichen Namespace verwendet wie Ihre Klassenbibliothek. Das kann ziemlich verwirrend werden, wenn ein Benutzer Ihrer Klassenbibliothek auch die Klassenbibliothek des anderen Programmierers verwenden will. Um dem vorzubeugen, ist es üblich, dem Namespace einer Klassenbibliothek den Namen des Autors und einen

Punkt voranzustellen. Bei professionellen Projekten verwendet man meist den Namen der Firma, die die jeweilige Klassenbibliothek veröffentlicht.

Daher ändern wir die Klasse Rechner jetzt wie folgt.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MeineFirma.Mathematik
6  {
7      public class Rechner
8      {
9          public double Addiere(double a, double b)
10         {
11             return a + b;
12         }
13
14         public double Subtrahiere(double a, double b)
15         {
16             return a - b;
17         }
18
19         public double Multpliziere(double a, double b)
20         {
21             return a * b;
22         }
23
24         public double Dividiere(double a, double b)
25         {
26             return a / b;
27         }
28     }
29 }
30 }
```

Wenn wir jetzt die Klasse Program der Konsolenapplikation TestMathematik betrachten, dann ist der Name Mathematik nach der using-Anweisung rot unterstrichen. Und wenn wir die Maus auf die rote Wellenlinie bewegen, sehen wir eine Fehlermeldung, die uns sagt, dass der Compiler den Namespace Mathematik nicht finden kann. Wir ändern die using-Anweisung wie folgt:

```
1  using MeineFirma.Mathematik;
```

Dann verschwindet der Fehler. Nach diesem kurzen Ausflug in das Thema Namespace werden wir jetzt die Klasse Rechner aus unserer Klassenbibliothek verwenden.

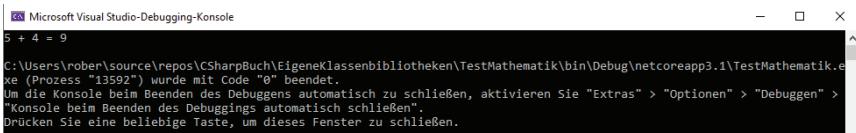
```
1  using System;
2  using MeineFirma.Mathematik;
3
4  namespace TestMathematik
5  {
```

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

```
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var rechner = new Rechner();
11             var a = 5.0d;
12             var b = 4.0d;
13             var c = rechner.Addiere(a, b);
14             Console.WriteLine($"{a} + {b} = {c}");
15         }
16     }
17 }
```

Wir verwenden die Methode `Addiere()` der Klasse `Rechner` um zwei Zahlen zu addieren und geben das Ergebnis am Bildschirm aus.

Damit wir unser Programm auch starten können, müssen wir noch dafür sorgen, dass die Konsolenapplikation `TestMathematik` das Startprojekt der Projektmappe ist. Dazu klicken wir mit der rechten Maustaste auf das Projekt `TestMathematik` und wählen „Als Startprojekt festlegen“ aus dem Kontextmenü aus. Jetzt können wir die Konsolenapplikation `TestMathematik` starten.



**Abb. 13.3.3** Bildschirmausgabe der Konsolenapplikation `TestMathematik`

Das Verwenden einer Klassenbibliothek durch einen Projektverweis auf ein anderes Projekt der gleichen Projektmappe ist ein Verfahren, das nur für den Programmierer der Klassenbibliothek geeignet ist, da wir dazu den Programmcode der Klassenbibliothek benötigen. Dem üblichen Benutzer einer Klassenbibliothek steht diese normalerweise nur in kompilierter Form zur Verfügung. Die kompilierte Klassenbibliothek `Mathematik.dll` finden wir im Verzeichnis `Mathematik\bin\debug\netcoreapp3.1` unterhalb des Verzeichnisses der Projektmappe. Um die Klassenbibliothek `Mathematik.dll` in seiner kompilierten Form zu verwenden, legen wir ein neues Projekt für eine KonsolenApp in einer eigenen Projektmappe an. Als Projektname vergeben wir `TestMathematik2` und den Namen der Projektmappe belassen wir bei `TestMathematik2` – sowie von Visual Studio vorgeschlagen.

Jetzt klicken wir im Projektmappen-Explorer mit der rechten Maustaste auf Abhängigkeiten unterhalb des Projekts `TestMathematik2`. Im Kontextmenü wählen wir „Projektverweis hinzufügen...“ aus.

## 13.3 Die eigene Klassenbibliothek verwenden

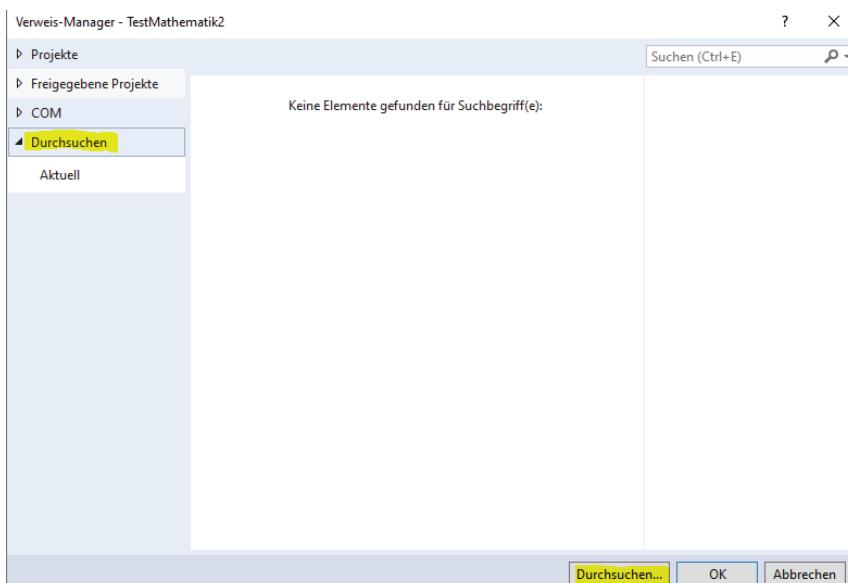


Abb. 13.3.4 Projektverweis auf eine Klassenbibliothek festlegen

Wählen Sie im Menü auf der linken Seite des Dialogs den Menüpunkt „Durchsuchen“ aus und klicken Sie auf die Schaltfläche „Durchsuchen...“.

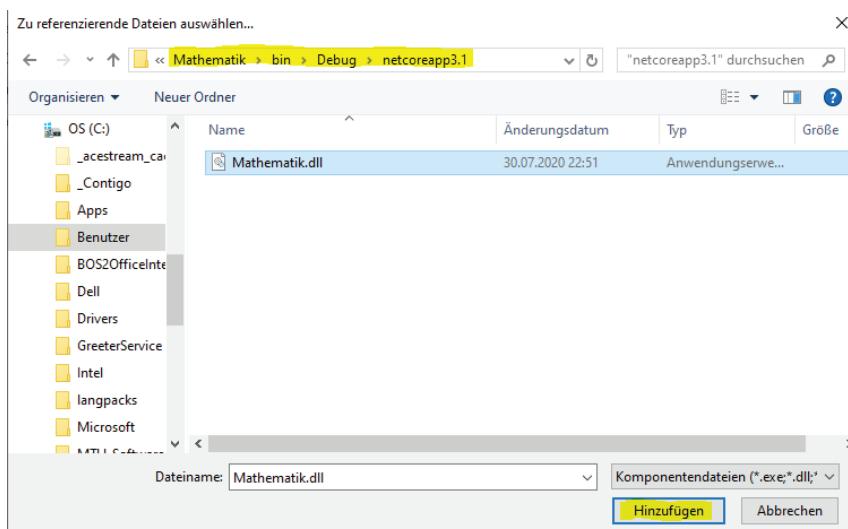


Abb. 13.3.5 Auswahl der Klassenbibliothek Matematik.dll

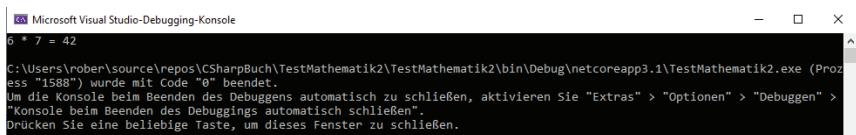
## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Im Dateiauswahl dialog navigieren Sie zu der Datei Mathematik.dll und klicken auf die Schaltfläche „Hinzufügen“ und dann auf Schaltfläche „OK“.

Damit haben wir einen Projektverweis auf die kompilierte Klassenbibliothek Mathematik.dll erstellt und können sie verwenden. Dazu ändern wir unser Hauptprogramm wie folgt:

```
1 using MeineFirma.Mathematik;
2 using System;
3
4 namespace TestMathematik2
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var a = 6;
11             var b = 7;
12             var rechner = new Rechner();
13             var c = rechner.Mulitplizierte(a, b);
14             Console.WriteLine($"{a} * {b} = {c}");
15         }
16     }
17 }
```

Mit der Anweisung `using MeineFirma.Mathematik` binden wir die Klassenbibliothek ein. Wir erstellen eine Instanz der Klasse `Rechner` aus der Klassenbibliothek und verwenden die Methode `Multiplizierte()`, um zwei Zahlen miteinander zu Multiplizieren. Das Ergebnis geben wir am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the command `6 * 7 = 42`. Below this, there is some diagnostic text from the application's output, including the path to the executable and instructions about how to close the console.

```
Microsoft Visual Studio-Debugging-Konsole
6 * 7 = 42
C:\Users\hoben\source\repos\CSharpBuch\TestMathematik2\TestMathematik2\bin\Debug\netcoreapp3.1\TestMathematik2.exe (Prozess "1588") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 13.3.6 Bildschirmausgabe der Konsolenapplikation TestMathematik2

### 13.4 Codenavigation

Unter der sogenannten Codenavigation verstehen wir das effiziente Sich-bewegen durch den Programmcode, der auf viele Dateien verteilt ist. Da unsere Übungs- und Beispielprogramme nicht wirklich groß sind, haben Sie dieses Thema vermutlich auch nicht sonderlich vermisst. Aber stellen Sie sich vor, sie arbeiten in einem ernsthaften Entwicklungsprojekt in einem Unternehmen. Da ist eine Projektmappe mit 10-20 Projekten völlig normal. Und wenn dann jedes Projekt noch aus durchschnittlich 50-100 Dateien mit durchschnittlich 500-1000 Zeilen Programmcode besteht, ist das auch nicht ungewöhnlich. Für diese Fälle gibt es in Visual Studio etliche Hilfsfunk-

tionen für die Codenavigation. Diese Hilfsfunktionen wollen wir in diesem Unterkapitel im Einzelnen betrachten.

Als Beispielprogramm für die Codenavigation benutzen wir eine Variante der größten Programme, die wir im Rahmen dieses Buchs geschrieben haben: Das Telefonverzeichnis aus der Übungsaufgabe im Kapitel 9.8. Die Variante unterscheidet sich von der Musterlösung aus Kapitel 9.8 dadurch, dass die beiden Klassen `Verzeichnis` und `VerzeichnisEintrag` in eine eigene Klassenbibliothek ausgelagert wurden.

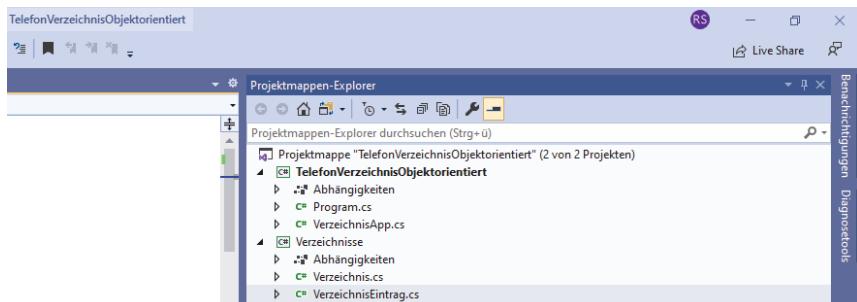


Abb. 13.4.1 Die App Telefonverzeichnis mit eigener Klassenbibliothek

Die erste Funktionalität, die wir betrachten wollen, ist die Suchfunktion. Hierbei handelt es sich um eine sog. Volltextsuche. Das heißt, wir können nach beliebigen Texten in unserem Programmcode suchen. So eine Suchfunktion werden die meisten von Ihnen in einem Texteditor oder einem Textverarbeitungsprogramm gesehen haben. Die Besonderheit der Suchfunktion in Visual Studio ist, dass Sie speziell auf die Bedürfnisse von Programmierern zugeschnitten ist. Die Suchfunktion können wir mit der Maus über den Menüpunkt Bearbeiten\Suchen und Ersetzen\Schnellsuche oder mit der Tastenkombination Strg + F aufrufen. Dann wird in der rechten oberen Ecke des Editorfensters ein kleines Fenster eingeblendet. Der Name des Codeelements, auf dem sich Cursor im aktiven Editor Fenster befindet, wird automatisch als zu suchender Text in das Suchfenster übernommen. In unserem Beispiel ist das der Klassename `Verzeichnis`.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

The screenshot shows the Visual Studio IDE with the 'VerzeichnisApp.cs' file open. The code editor displays the following C# code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Verzeichnisse;
5
6  namespace TelefonVerzeichnisObjektorientiert
7  {
8      public class VerzeichnisApp
9      {
10          private Verzeichnis verzeichnis;
11
12          public VerzeichnisApp()
13          {
14              verzeichnis = new Verzeichnis();
15          }
16
17          private void ZeigeMenu()
18          {
19              Console.Clear();
20              Console.WriteLine("1. Alle Einträge anzeigen.");
21              Console.WriteLine("2. Einen bestimmten Eintrag anzeigen");
22              Console.WriteLine("3. Einträge suchen nach Vornamen.");
23              Console.WriteLine("4. Einträge suchen nach Nachnamen.");
24              Console.WriteLine("5. Eintrag hinzufügen.");
25              Console.WriteLine("6. Eintrag ändern.");
26              Console.WriteLine("7. Eintrag löschen");
27              Console.WriteLine("8. Programm beenden.");
28              Console.WriteLine("-----");
29              Console.WriteLine("Wählen Sie einen Menüpunkt (1-8)");
30          }
31
32          private int LeseMenuPunkt()
33          {
34      }
```

The word 'Verzeichnis' appears highlighted in yellow across multiple lines of code, indicating it is the current search term. The status bar at the bottom right shows 'Zeile: 14 Zeichen: 45 SPC CRLF'. The top menu bar shows 'TelefonVerzeichnisObjektorientiert'.

Abb. 13.4.2 Suchen nach einem Text

Jedes Vorkommen der Zeichenfolge „Verzeichnis“ ist Dunkelgelb hinterlegt. Am rechten Rand des Editorfensters wird mit dunkelgelben Quadraten die Verteilung der Treffer auf die zu durchsuchende Programmdatei angezeigt. Standardmäßig wird in der Visual Studio Suche jedes Auftreten des Suchtextes angezeigt. Egal ob die Groß-/Kleinschreibung übereinstimmt und egal ob es sich beim Treffer um ein ganzes Wort oder nur um einen Teil eines Wortes handelt. Möchte man, dass bei der Trefferanzeige die Groß-/Kleinschreibung beachtet wird, kann man diese Funktionalität mit der Optionsschaltfläche „Aa“ einschalten.

## 13.4 Codenavigation

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes 'Datei', 'Bearbeiten', 'Ansicht', 'Projekt', 'Erstellen', 'Debuggen', 'Test', 'Analysieren', 'Extras', 'Erweiterungen', 'Fenster', 'Hilfe', and 'Suchen (Strg+Q)'. The title bar displays the project name 'TelefonVerzeichnisObjektorientiert' and the file 'VerzeichnisApp.cs'. The code editor window contains C# code for a console application. The code defines a class 'VerzeichnisApp' with methods 'ZeigeMenu()' and 'leseMenuPunkt()'. The word 'Verzeichnis' is highlighted in orange, indicating it is the current search term. The status bar at the bottom shows 'Zeile: 4 Zeichen: 18 SPC CRLF'. On the right side, there is a 'Toolbox' and a 'Solution Explorer' window titled 'Verzeichnis' which lists 'Aktuelles Dokument'.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using Verzeichnisse;
5
6 namespace TelefonVerzeichnisObjektorientiert
7 {
8     public class VerzeichnisApp
9     {
10         private Verzeichnis verzeichnis;
11
12         public VerzeichnisApp()
13         {
14             verzeichnis = new Verzeichnis();
15         }
16
17         private void ZeigeMenu()
18         {
19             Console.Clear();
20             Console.WriteLine("1. Alle Einträge anzeigen.");
21             Console.WriteLine("2. Einen bestimmten Eintrag anzeigen");
22             Console.WriteLine("3. Einträge suchen nach Vornamen.");
23             Console.WriteLine("4. Einträge suchen nach Nachnamen.");
24             Console.WriteLine("5. Eintrag hinzufügen.");
25             Console.WriteLine("6. Eintrag ändern.");
26             Console.WriteLine("7. Eintrag löschen.");
27             Console.WriteLine("8. Programm beenden.");
28             Console.WriteLine("-----");
29             Console.WriteLine("Wählen Sie einen Menüpunkt (1-8)");
30         }
31
32         private int leseMenuPunkt()
33         {
```

Abb. 13.4.3 Suche unter Beachtung der Groß-/Kleinschreibung

Jetzt werden nur noch Treffer angezeigt, bei denen auch die Groß-/Kleinschreibung übereinstimmt. Wenn wir unter den Treffern unserer Suche nur ganze Wörter sehen wollen, können wir das mit der Optionsschaltfläche „AB“ erreichen.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

The screenshot shows the Microsoft Visual Studio interface with the following details:

- Menu Bar:** Datei, Bearbeiten, Ansicht, Projekt, Erstellen, Debuggen, Test, Analysieren, Extras, Erweiterungen, Fenster, Hilfe, Suchen (Strg+Q).
- Toolbars:** Standard, Debug, Task List.
- Status Bar:** Zeile: 10 Zeichen: 28 SPC CRLF.

The code editor displays the file `VerzeichnisApp.cs` from the project `TelefonVerzeichnisObjektorientiert`. The code implements a console application for managing address book entries. The search results pane on the right side highlights matches for the word "Verzeichnis".

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Verzeichnisse;
5
6  namespace TelefonVerzeichnisObjektorientiert
7  {
8      public class VerzeichnisApp
9      {
10         private Verzeichnis verzeichnis;
11
12         public VerzeichnisApp()
13         {
14             verzeichnis = new Verzeichnis();
15         }
16
17         private void ZeigeMenu()
18         {
19             Console.Clear();
20             Console.WriteLine("1. Alle Einträge anzeigen.");
21             Console.WriteLine("2. Einen bestimmten Eintrag anzeigen");
22             Console.WriteLine("3. Einträge suchen nach Vornamen.");
23             Console.WriteLine("4. Einträge suchen nach Nachnamen.");
24             Console.WriteLine("5. Eintrag hinzufügen.");
25             Console.WriteLine("6. Eintrag ändern.");
26             Console.WriteLine("7. Eintrag löschen");
27             Console.WriteLine("8. Programme beenden.");
28             Console.WriteLine("-----");
29             Console.WriteLine("Wählen Sie einen Menüpunkt (1-8)");
30         }
31
32         private int LeseMenuPunkt()
33         {
```

Abb. 13.4.4 Suche nach ganzen Wörtern unter Beachtung der Groß-/Kleinschreibung

Jetzt sehen wir nur Treffer mit ganzen Wörtern, bei denen die Groß-/Kleinschreibung übereinstimmt.

Mit einem Klick auf den horizontalen Pfeil rechts neben dem Suchtext oder der Taste F3 können wir von Treffer zu Treffer springen.

## 13.4 Codenavigation

```

1 //using System;
2 //using System.Collections.Generic;
3 //using System.Text;
4 //using Verzeichnisse;
5
6 namespace TelefonVerzeichnisObjektorientiert
7 {
8     public class VerzeichnisApp
9     {
10         private Verzeichnis verzeichnis;
11
12         public VerzeichnisApp()
13         {
14             verzeichnis = new Verzeichnis();
15         }
16
17         private void ZeigeMenu()
18         {
19             Console.Clear();
20             Console.WriteLine("1. Alle Einträge anzeigen.");
21             Console.WriteLine("2. Einen bestimmten Eintrag anzeigen");
22             Console.WriteLine("3. Einträge suchen nach Vornamen.");
23             Console.WriteLine("4. Einträge suchen nach Nachnamen.");
24             Console.WriteLine("5. Eintrag hinzufügen.");
25             Console.WriteLine("6. Eintrag ändern.");
26             Console.WriteLine("7. Eintrag löschen.");
27             Console.WriteLine("8. Programm beenden.");
28             Console.WriteLine("-----");
29             Console.WriteLine("Wählen Sie einen Menüpunkt (1-8)");
30         }
31
32         private int LeseMenuPunkt()
33         {

```

Abb. 13.4.5 Springen von Treffer zu Treffer

Der aktuelle Treffer wird grau hinterlegt.

Mit der Combobox rechts unten im Suchfenster können wir den Bereich festlegen, in dem die gewünschte Zeichenfolge gesucht werden soll. Es gibt sechs Möglichkeiten, den Suchbereich einzuschränken.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Verzeichnisse;
5
6  namespace TelefonVerzeichnisObjektorientiert
7  {
8      public class VerzeichnisApp
9      {
10         private Verzeichnis verzeichnis;
11
12         public VerzeichnisApp()
13         {
14             verzeichnis = new Verzeichnis();
15         }
16
17         private void ZeigeMenu()
18         {
19             Console.Clear();
20             Console.WriteLine("1. Alle Einträge anzeigen.");
21             Console.WriteLine("2. Einträge suchen nach Vornamen.");
22             Console.WriteLine("3. Einträge suchen nach Nachnamen.");
23             Console.WriteLine("4. Einträge suchen nach Nachnamen.");
24             Console.WriteLine("5. Eintrag hinzufügen.");
25             Console.WriteLine("6. Eintrag ändern.");
26             Console.WriteLine("7. Eintrag löschen.");
27             Console.WriteLine("8. Programme beenden.");
28             Console.WriteLine("-----");
29             Console.WriteLine("Wählen Sie einen Menüpunkt (1-8)");
30         }
31
32         private int LeseMenuPunkt()
33         {

```

**Abb. 13.4.6** Festlegen des Suchbereichs

### Aktueller Block:

Durch diese Auswahl werden nur Treffer im aktuellen Block angezeigt. Das umfasst Treffer in dem aktuellen, von geschweiften Klammern umgebenen Block, in dem sich der Cursor befindet.

### Alle geöffneten Dokumente:

Diese Einschränkung begrenzt den Suchbereich auf alle aktuell im Editor geöffneten Programmdateien.

### Aktuelles Projekt

Bei dieser Einstellung ist der Suchbereich das Projekt, zu dem die Programmdatei gehört, die gerade im Editor aktiv ist.

### Gesamte Projektmappe

Mit dieser Auswahl werden alle Projekte der gesamten Projektmappe durchsucht.

Der horizontale Pfeil rechts neben dem Eingabefeld für den Suchtext, mit dem der nächste Treffer angesprungen werden kann, ist eine Auswahl einer Combobox.



Abb. 13.4.7 Alle Treffer auf einmal suchen

Mit der Auswahl „Vorheriges suchen“ beziehungsweise mit der Tastenkombination UMSCHALT-F3 wird zum vorherigen Treffer zurückgesprungen. Mit der Auswahl „Alle suchen“ werden alle Treffer des Suchtextes unter Berücksichtigung der Sucheinstellungen und des Suchbereichs gesucht.

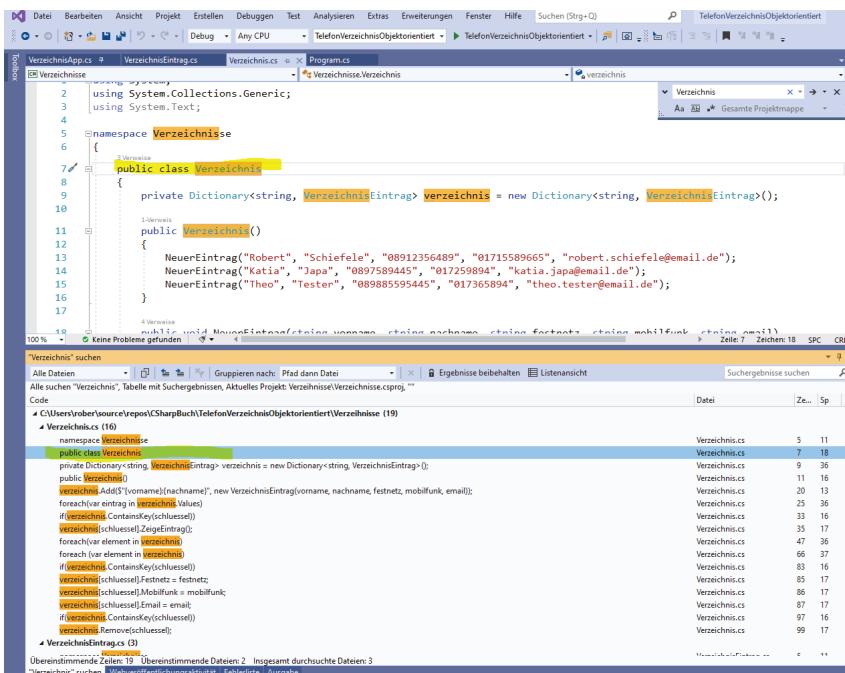


Abb. 13.4.8 Alle Treffer werden in einer Liste angezeigt

Die Funktion „Alle suchen“ zeigt alle Treffer in einer Liste an. Mit einem Klick auf einen Eintrag der Liste können wir direkt zu der betreffenden Codestelle springen.

Neben der rein textbasierten Suchfunktion enthält die Codenavigation von Visual Studio auch Funktionen, die sich an der Syntax der Programmiersprache orientieren.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Wenn wir zum Beispiel mit rechten Maustaste auf einen Klassennamen an einer Codestelle, wo die Klasse instanziiert wird, klicken, finden wir im Kontextmenü die Auswahl „Definition einsehen“.

The screenshot shows a code editor window with C# code. A right-click context menu is open over the word 'Verzeichnis' in the line 'verzeichnis = new Verzeichnis();'. The menu item 'Definition' is highlighted. The code in the editor is:

```

1 verzeichnis = new Verzeichnis();
2
3 public class Verzeichnis
4 {
5     private Dictionary<string, VerzeichnisEintrag> verzeichnis = new Dictionary<string, VerzeichnisEintrag>();
6
7     public Verzeichnis()
8     {
9         NeuerEintrag("Robert", "Schiefele", "08912356489", "0171589665", "robert.schiefele@email.de");
10        NeuerEintrag("Katia", "Japa", "0897589445", "017259894", "katia.japa@email.de");
11        NeuerEintrag("Theo", "Tester", "089885595445", "017365894", "theo.tester@email.de");
12    }
13
14    public void NeuerEintrag(string vorname, string nachname, string festnetz, string mobilfunk, string email)
15    {
16        verzeichnis.Add($"{vorname}:{nachname}", new VerzeichnisEintrag(vorname, nachname, festnetz, mobilfunk));
17    }
18
19
20
21

```

Abb. 13.4.9 Definition einsehen

Der Programmcode der entsprechende Klasse wird in einem gelb hinterlegten Fenster eingeblendet. In diesem Fenster kann durch den Programmcode der Klasse gescrollt werden. Benötigt man einen größeren Überblick über die betreffende Klasse, kann mit dem Kontextmenü „Gehe zu Definition“ direkt in den Programmcode der entsprechenden Klasse gesprungen werden.

Die Umkehrung der Funktionen „Definition einsehen“ und „Gehe zu Definition“ stellt die Funktion „Alle Verweise suchen“ dar. Sie befindet sich im Kontextmenü hinter den Namen von Klassen, Variablen und Methoden. Wird sie aufgerufen, wird eine Liste aller Verwendungen des betreffenden Elements angezeigt.

The screenshot shows a code editor window with C# code. A right-click context menu is open over the word 'Verzeichnis' in the line 'public class Verzeichnis'. The menu item 'Alle Verweise suchen' is highlighted. Below the editor is a 'Verzeichnis'-Verweise' tool window showing a list of references. The code in the editor is identical to the one in Abb. 13.4.9.

Code	Datei	Ze...	Sp	Projekt	Art
VerzeichnisApp.cs	10	17		TelefonVerzeichnisObjekte...	V...
VerzeichnisApp.cs	14	31		TelefonVerzeichnisObjekte... Konstrukt...	V...
VerzeichnisApp.cs	14	31		TelefonVerzeichnisObjekte... Konstrukt...	V...
Verzeichnis.cs	7	18		Verzeichnisse	
Verzeichnis.cs	11	16		Verzeichnisse	V...

Abb. 13.4.10 Alle Verweise suchen

Durch einen Klick auf einen Listeneintrag kann direkt zur entsprechenden Stelle im Programmcode navigiert werden.

Visual Studio merkt sich die Stellen im Programmcode in der Reihenfolge, in der der Programmierer sie bearbeitet hat. Daher kann zwischen diesen Stellen Vorwärts und Rückwärts gesprungen werden. Wenn wir zum Beispiel in einer Programmdatei eine Instanz einer Klasse erzeugen, können wir mit der Funktion „Gehe zu Definition“ zum Programmcode dieser Klasse springen, um uns Ihre Funktionsweise nochmal ins Gedächtnis zu rufen und dann mit der Schaltfläche „Rückwärts Navigieren“ an die Stelle im unserem Programmcode zurückzuspringen, an der wir uns vorher befunden haben.



Abb. 13.4.11 Die Schaltfläche „Rückwärts navigieren“

Nachdem wir zurückgesprungen sind, können wir mit der Schaltfläche „Vorwärts navigieren“ auch wieder innerhalb unseres „Workflows“ vorwärts springen. In unserem obigen Beispiel wäre das wieder der Programmcode der Klasse.

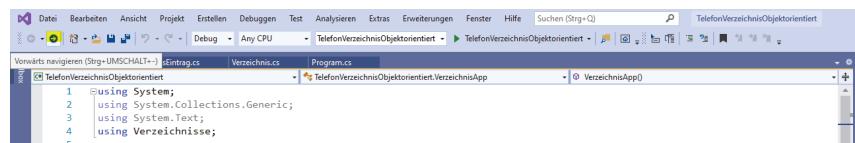


Abb. 13.4.12 Die Schaltfläche „Vorwärts navigieren“

Eine weitere, sehr praktische Funktion der Codenavigation ist die „Gehe zu“-Funktion. Mit dieser Funktion kann spezifisch nach verschiedenen Aspekten eines Programms gesucht werden. Die „Gehe zu“-Funktion wird über das Hauptmenü von Visual Studio mit „Bearbeiten/Gehe zu“ erreicht.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

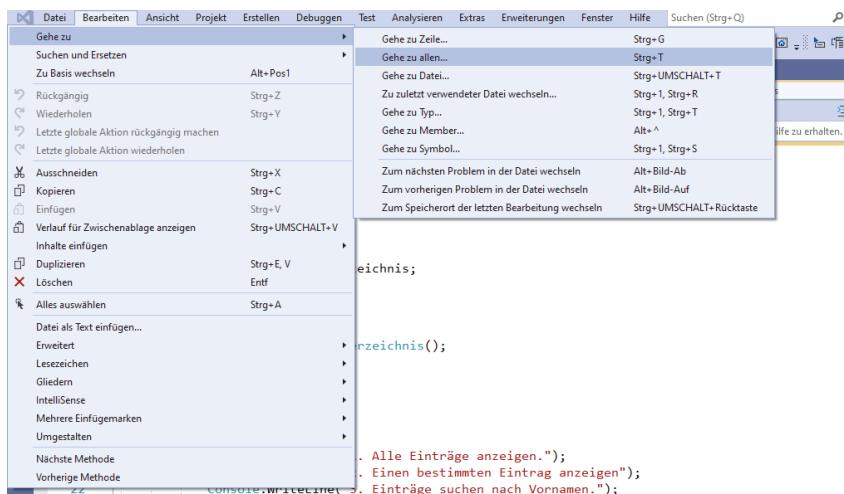


Abb. 13.4.13 Das „Gehe zu“-Menü

Das „Gehe zu“-Menü enthält für jeden Aspekt, nach dem ein Programm durchsucht werden kann, einen Menüpunkt:

*Gehe zu Zeile...*

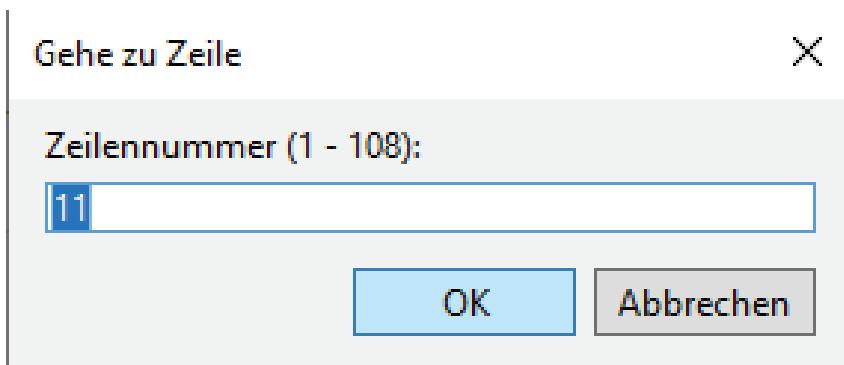


Abb. 13.4.14 Geh zu Zeile ...

Dieser Menüpunkt springt direkt zu einer vom Benutzer angegebenen Zeilennummer in der Programmdatei, die aktuell bearbeitet wird.

*Geh zu allen ...*

Die „Gehe zu allen ...“-Funktion öffnet ein Suchfenster, mit dem wir eine Projektmappe nach Treffern in Dateien, Klassen, Membervariablen und Methoden durchsuchen können. Eine Komponente wird als Treffer erkannt, wenn ihr Name den Suchbegriff enthält.

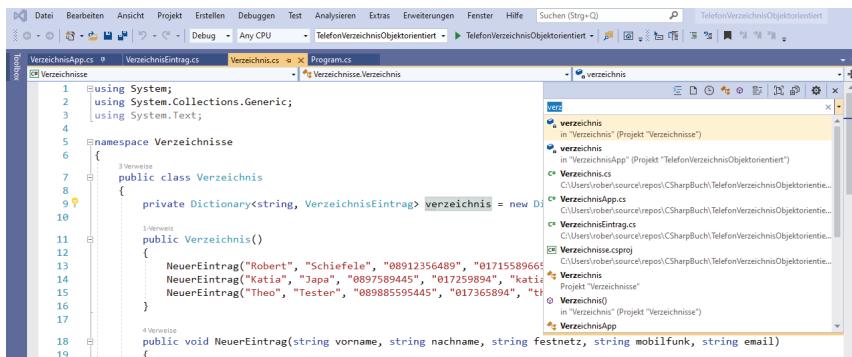


Abb. 13.4.15 Gehe zu allen ...

Sämtliche Treffer werden in Echtzeit in einer Liste angezeigt und können per Mausklick angesprungen werden.

*Gehe zu Datei ...*

Die Funktion „Gehe zu Datei“ ist ein Spezialfall der „Gehe zu allen ...“-Funktion. Sie blendet das gleiche Suchfenster ein.

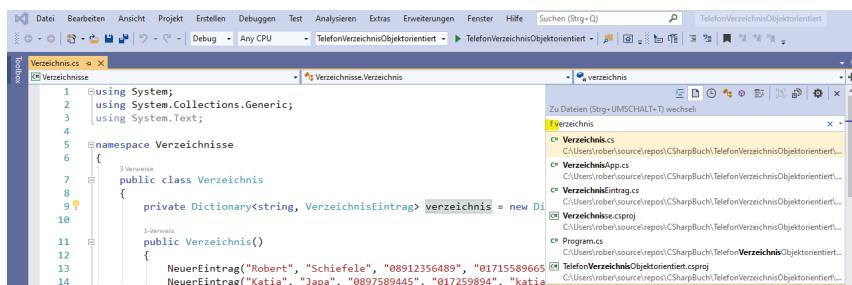


Abb. 13.4.16 Gehe zu Datei ...

Allerdings wird im Eingabefeld dem Suchbegriff der Buchstabe f und ein Leerzeichen vorangestellt und in der Ergebnisliste erscheinen nur Dateien als Treffer.

*Zu zuletzt verwendeter Datei wechseln ...*

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Diese Funktion arbeitet ähnlich wie die Funktion „Gehe zu Datei ...“.

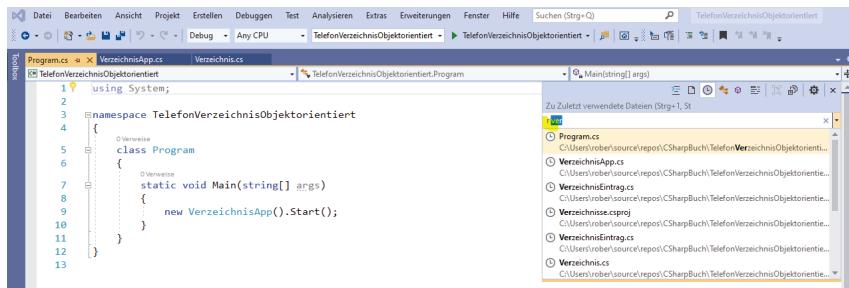


Abb. 13.4.17 Zu zuletzt bearbeiteter Datei wechseln ...

Bei der Funktion „Zu zuletzt bearbeiteter Datei wechseln ...“ wird dem Suchbegriff der Buchstabe r und ein Leerzeichen vorangestellt und in der Ergebnisliste erscheinen die Dateien in der Reihenfolge der letzten Bearbeitung.

*Gehe zu Typ ...*

Hierbei handelt es sich um einen weiteren Spezialfall von „Gehe zu ...“



Abb. 13.4.18 Gehe zu Typ ...

Dem Suchtext wird der Buchstabe t und ein Leerzeichen vorangestellt und die Ergebnisliste enthält nur Klassen.

*Gehe zu Member ...*

Auch diese Funktion verwendet das Suchfenster der „Gehe zu allen ...“-Funktion:



Abb. 13.4.19 Gehe zu Member ...

Hier wird dem Suchtext der Buchstabe m und ein Leerzeichen vorangestellt und die Ergebnisliste enthält nur public-Membervariablen und public-Methoden.

*Gehe zu Symbol ...*

Bei dieser Funktion beschränkt sich die Suche auf Elemente, denen vom Compiler ein sogenanntes Symbol zugeordnet wird. Wir wollen an dieser Stelle aber nicht tiefer in die Compilertheorie einsteigen, daher begnügen wir uns mit der Aufzählung der Elemente, die vom Compiler ein Symbol benötigen. Hierbei handelt es sich um Klassen, public- und private-Membervariablen und public- und private-Methoden. Übergabeparameter und lokale Variablen gehören nicht dazu, da der Compiler ihnen kein Symbol zuweisen muss.

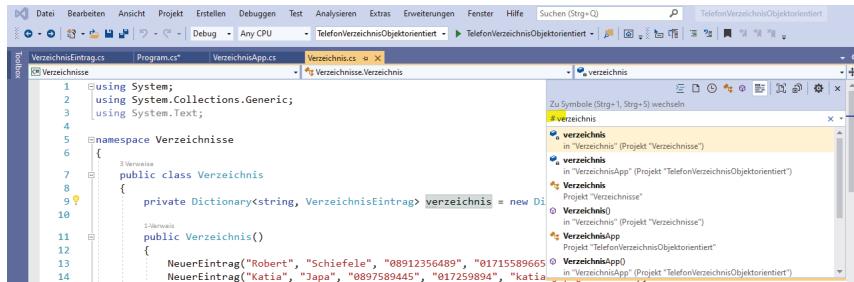


Abb. 13.4.20 Gehe zu Symbol ...

Bei der Suche nach Symbolen wird dem Suchtext das Zeichen # und ein Leerzeichen vorangestellt und die Ergebnisliste enthält nur Elemente, die vom Compiler ein Symbol benötigen.

Die letzte Variante der Codenavigation hat vom Microsoft-Marketing den Namen Code Lens erhalten. Über jeder Klasse, jeder public-Property und jeder public-Methode befindet sich ein kleiner grauer Hinweis, der angibt, wie oft dieses Element im Programcode verwendet wird. Wenn wir auf einen dieser grauen Hinweise klicken, so wird eine Liste der Verwendungen des betreffenden Elements eingeblendet.

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

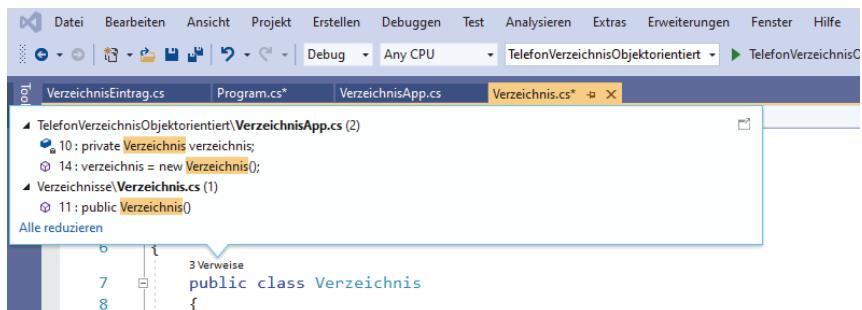


Abb. 13.4.21 Code Lens in Aktion

Durch einen Doppelklick auf eine der Verwendungen des betreffenden Elements kann zur entsprechenden Stelle im Programmcode navigiert werden.

### 13.5 Fehler finden mit dem Debugger

In diesem Unterkapitel beschäftigen wir uns mit der Suche von Fehlern in Programmen und wie uns Visual Studio dabei unterstützen kann. Wenn wir bisher in diesem Buch Programmfehlern begegnet sind, dann waren es Fehler, die vom Compiler gefunden wurden. Die Programmzeile:

```
1 int a = "Hallo";
```

ist ein typisches Beispiel dafür. Ein String-Literal kann einer Variablen vom Typ `int` nicht zugewiesen werden. Visual Studio kompiliert ständig im Hintergrund und zeigt den Fehler sofort im Editor an.

Deutlich komplizierter sind sogenannte Laufzeitfehler zu finden. Das heißt Fehler, die der Compiler nicht finden kann. Betrachten wir folgendes Codefragment:

```
1 int a = 6;
2 int b = 7;
3 int c = a + b;
```

Hier meldet uns der Compiler keinen Fehler. Auch sehen wir auf den ersten Blick nicht, was daran falsch sein könnte. Aber stellen wir uns vor, die Absicht des Programmierers war es, die Zahlen 6 und 7 zu multiplizieren und der Ausdruck `a + b` ist lediglich das Resultat eines Flüchtigkeitsfehlers beim Tippen. In einem kleinen Demoprogramm findet ein Programmierer seinen Fehler sofort bei einer kurzen Durchsicht seines Programmcodes. Aber bei einem umfangreichen und komplexen Programm kann das Aufspüren derartiger Fehler sehr langwierig werden.

Visual Studio enthält einen sogenannten Debugger, der bei der Suche von Laufzeitfehlern sehr hilfreich sein kann. Der Begriff Debugger kommt aus dem Englischen und bedeutet so viel wie „Entwanzen“. Bereits im 19. Jahrhundert benutzen Ingenieure den Begriff „Bug (Käfer, Wanze) in der Maschine“ für seltsame und unerklärliche technische Defekte. 1947 entdeckten Techniker im Team von Grace Hopper, einer Pionierin der Informatik, eine tote Motte in einem Relais als Ursache für eine Fehlfunktion eines Computers. Grace Hopper klebte die Motte in ihr Tagebuch und schrieb dazu den Satz: „First actual case of a bug being found.“ (Deutsch: „Das erste Mal, dass eine Wanze tatsächlich gefunden wurde.“). Daher wird in der Informatik die Suche nach Fehlern in Computerprogrammen als debugging (entwanzen) bezeichnet.

Visual Studio verfügt über einen sehr mächtigen und umfangreichen Debugger, dessen vollständige Beschreibung den Rahmen dieses Buches sprengen würde. Daher werden wir uns auf die wichtigsten Funktionalitäten beschränken, die zum Beseitigen von Laufzeitfehlern und unerwartetem Programmverhalten nötig sind.

Um die Verwendung des Visual Studio Debuggers zu demonstrieren, verwenden wir die Klasse Mathematik aus dem Unterkapitel „13.2 Eine Klassenbibliothek erstellen“. Des Weiteren bauen wir einen kleinen Fehler in die Klasse ein und lernen, wie wir diesen Fehler mit Hilfe des Debuggers finden können.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MeineFirma.Mathematik
6  {
7      public class Rechner
8      {
9          public double Addiere(double a, double b)
10         {
11             return a * b;
12         }
13
14         public double Subtrahiere(double a, double b)
15         {
16             return a - b;
17         }
18
19         public double Multipliziere(double a, double b)
20         {
21             return a * b;
22         }
23
24         public double Dividiere(double a, double b)
25         {
26             return a / b;
27         }
28     }
29 }
```

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

30 }

Die Methode `Addiere()` multipliziert die übergebenen Parameter `a` und `b`, anstatt sie zu addieren. Stellen wir uns vor, wir hätten diesen Fehler nicht bemerkt und wollten die Funktion der Klasse `Mathematik` mit einem Testprogramm überprüfen.

```
1 using System;
2 using MeineFirma.Mathematik;
3
4 namespace TestMathematik
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var rechner = new Rechner();
11             var a = 5.0d;
12             var b = 4.0d;
13             var c = rechner.Addiere(a, b);
14             Console.WriteLine($"{a} + {b} = {c}");
15         }
16     }
17 }
```

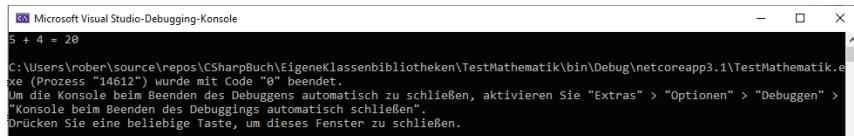


Abb. 13.5.1 Bildschirmausgabe eines fehlerhaften Programms

Jetzt sehen wir, dass mit unserem Programm etwas nicht stimmen kann, da  $5 + 4$  nicht 20 ist.

Um den Fehler mit Hilfe des Debuggers zu finden, klicken wir am rechten Rand des Editorfensters auf Höhe der Zeile:

```
1 var rechner = new Rechner();
```

## 13.5 Fehler finden mit dem Debugger

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Datei, Bearbeiten, Ansicht, Projekt, Erstellen, Debuggen, Test, Analysieren, Extras, Erweiterungen, Fenster, Hilfe, and Suchen (Strg+Q). The toolbar has icons for file operations like Open, Save, and Print. The status bar at the bottom right shows EigeneKlassenbibliotheken. The code editor window displays a C# file named Program.cs with the following code:

```

1  using System;
2  using MeineFirma.Mathematik;
3
4  namespace TestMathematik
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var rechner = new Rechner();
11             var a = 5.0d;
12             var b = 4.0d;
13             var c = rechner.Addiere(a, b);
14             Console.WriteLine($"{a} + {b} = {c}");
15         }
16     }
17 }

```

A red dot, indicating a break point, is placed on the line before the variable declaration "var rechner = new Rechner();". The line number 10 is highlighted in yellow.

Abb. 13.5.2 Ein Haltepunkt

Die Programmzeile wird dunkelrot hinterlegt. Damit haben wir einen sogenannten Haltepunkt definiert. Wenn wir jetzt unser Programm starten, wird die Ausführung des Programms bei dieser Programmzeile angehalten.

The screenshot shows the Microsoft Visual Studio IDE interface with the same code as in Abb. 13.5.2. The line "var rechner = new Rechner();" is highlighted in yellow, indicating it is the current line of execution. A red dot at the start of this line indicates it is a break point. The status bar at the bottom right shows 13.

Abb. 13.5.3 An einem Haltepunkt wird die Programmausführung angehalten

Wenn die Programmausführung anhält, wird die betroffene Programmzeile gelb hinterlegt. Jetzt haben wir verschiedene Möglichkeiten:

Mit einem Klick auf das grüne Dreieck im Menü oder mit der Taste F5 können wir das Programm weiterlaufen lassen. Das Programm läuft dann bis zum nächsten

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

Haltepunkt weiter beziehungsweise bis zum Ende des Programms, falls kein weiterer Haltepunkt definiert ist.

Mit dem Menüpunkt „Debuggen\Prozedurschritt“ oder der Taste F10 können wir das Programm Schritt für Schritt durchlaufen. Jedes Mal, wenn wir F10 drücken, springen wir zur nächsten Anweisung.

Wir drücken solange F10, bis die Programmausführung bei der Zeile `Console.WriteLine($"a + b = {c}");` anhält. Die Anweisung `var c = rechner.Addiere(a, b);` wurde dabei vollständig ausgeführt und dabei wurde die Methode `Addiere()` nicht schrittweise abgearbeitet. Jetzt können wir untersuchen, was im „Inneren“ unseres Programms passiert ist. Dazu bewegen wir die Maus über die Variable `c`.

```

8     static void Main(string[] args)
9
10    {
11        var rechner = new Rechner();
12        var a = 5.0d;
13        var b = 4.0d;
14        var c = rechner.Addiere(a, b);
15    }

```

Abb. 13.5.4 Der Debugger zeigt den aktuellen Wert einer Variablen

Visual Studio blendet uns ein kleines Fenster ein, das uns anzeigen, dass die Variable `c` den Wert 20 hat. Dadurch können wir leicht erkennen, dass die Methode `Addiere()` ein falsches Ergebnis zurückgeliefert hat. Der Fehler muss also in der Methode `Addiere()` stecken. Wir starten einen neuen, schrittweisen Durchgang durch unser Programm. Wenn die Programmausführung auf der Anweisung `var c = rechner.Addiere(a, b);` steht, rufen wir den Menüpunkt „Debuggen/EinzelSchritt“ auf oder drücken die Taste F11.

```

5  namespace MeineFirma.Mathematik
6  {
7      public class Rechner
8      {
9          public double Addiere(double a, double b)
10         {
11             return a * b;
12         }
13     }

```

Abb. 13.5.5 Mit „EinzelSchritt“ wird eine Methode durchlaufen.

Die Programmausführung hält in der Methode `Addiere()` und wir können leicht erkennen, dass die Methode `Addiere()` zwei Zahlen multipliziert, anstatt sie zu addieren.

Um eine weitere, sehr nützliche Funktionalität des Debuggers zu demonstrieren, betrachten wir das folgende kleine Programm:

```

1  using System;
2
3  namespace DebuggerDemo
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var a = 3;
10             for(var i = 1; i < 5; i++)
11             {
12                 var b = a / (a-i);
13             }
14         }
15     }
16 }
```

Dieses Programm hat keinen praktischen Sinn, es dient nur dazu, Funktionalitäten des Debuggers vorzustellen. Beim vierten Schleifendurchgang es kommt zu einer DivideByZeroException. Die Programmausführung stoppt und wir können den Zustand aller Variablen im Fenster „Lokal“ sehen.

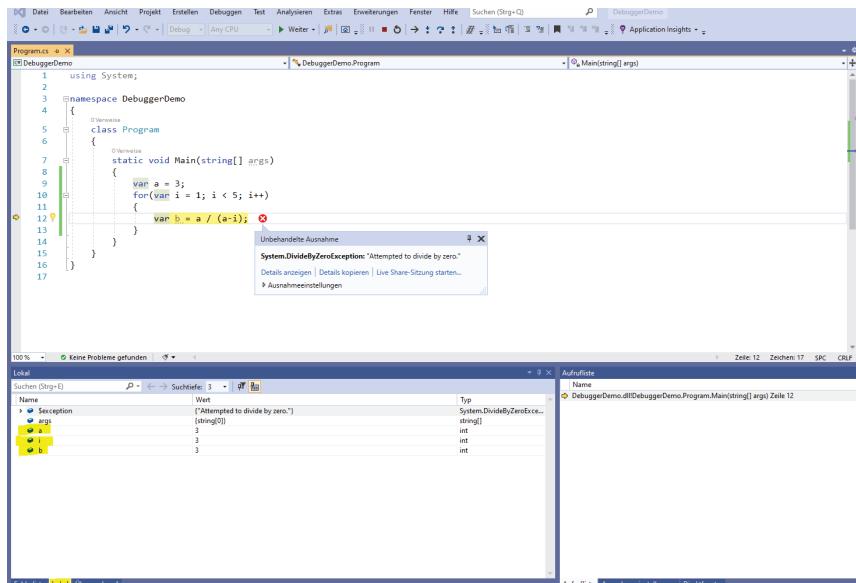


Abb. 13.5.6 Das Fenster „Lokal“

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

In diesem Beispiel ist es leicht zu erkennen, dass der Ausdruck  $a - i$  Null ergibt, wenn  $i$  den Wert 3 erhält und dass dadurch eine DivideByZeroException ausgelöst wird.

Wenn es sich um kompliziertere Ausdrücke handelt, kann das Fenster „Überwachen 1“ hilfreich sein. Dieses Fenster kann durch einen Klick auf den Reiter „Überwachen 1“ rechts neben dem Reiter „Lokal“ aktiviert werden.

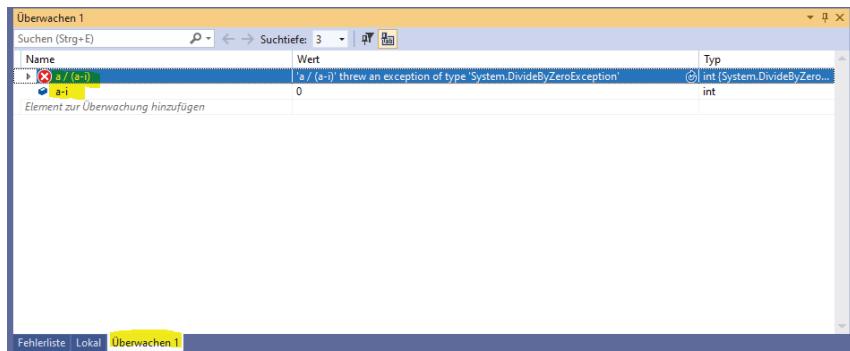
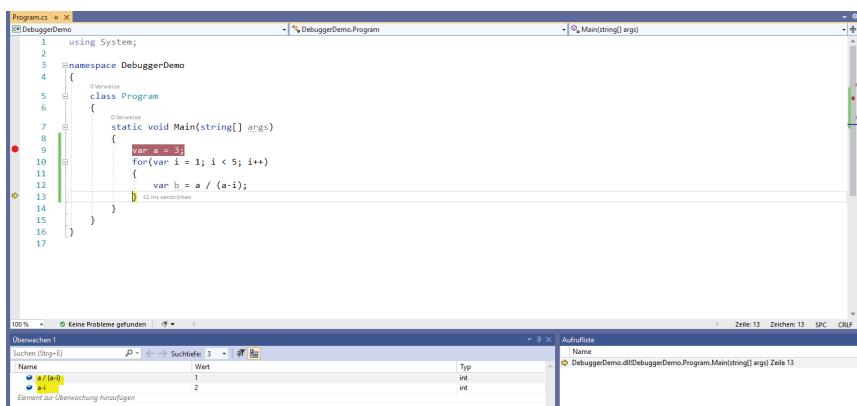


Abb. 13.5.7 Das Fenster „Überwachen 1“

In diesem Fenster können wir verschiedene Ausdrücke erfassen und deren Werte betrachten. Für unser Beispiel sehen wir, dass der Ausdruck  $a / (a-i)$  mit den aktuellen Werten eine DivideByZeroException auslöst und der Ausdruck  $a-i$  den Wert 0 ergibt.

Das Fenster „Überwachen 1“ zeigt die Ergebnisse der erfassten Ausdrücke immer bezüglich der Werte der in ihnen verwendeten Variablen an. Wenn wir unser Demoprogramm schrittweise ausführen, ändern sich bei jedem Schleifendurchgang die Werte für unsere erfassten Ausdrücke.

## 13.6 Übungsaufgabe: Eine eigene Klassenbibliothek erstellen



**Abb. 13.5.8** Das Fenster „Überwachen 1“ nach dem ersten Schleifendurchgang

## 13.6 Übungsaufgabe: Eine eigene Klassenbibliothek erstellen

In dieser Übung erstellen wir eine Klassenbibliothek und verwenden diese in einem Testprogramm so wie man es während der Entwicklung einer Klassenbibliothek tun würde. Danach verwenden wir die Klassenbibliothek in kompilierter Form, so wie es ein Benutzer oder Kunde tun würde.

### Teilaufgabe 1:

Erstellen Sie eine Klassenbibliothek, die die folgende Klasse bereitstellt.

```

1  using System;
2
3  namespace BegruesserBibliothek
4  {
5      public class Begruesser
6      {
7          public void HalloWelt()
8          {
9              Console.WriteLine("Hallo Welt");
10         }
11
12         public void HelloWorld()
13         {
14             Console.WriteLine("Hello World");
15         }
16
17         public void HolaMundo()
18         {
19             Console.WriteLine("Hola Mundo");
    
```

**13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene**

```
20     }
21 }
22 }
```

**Teilaufgabe 2:**

Erstellen Sie eine Konsolen-App als Testprogramm für die Klassenbibliothek `BegruesserBibliothek` in derselben Projektmappe.

**Teilaufgabe 3:**

Erstellen Sie eine Konsolen-App als Demoprogramm für die Klassenbibliothek `BegruesserBibliothek` in einer eigenen Projektmappe unter Verwendung der kompilierten `BegruesserBibliothek`.

## 13.6 Übungsaufgabe: Eine eigene Klassenbibliothek erstellen

## Musterlösung für Teilaufgabe 1:

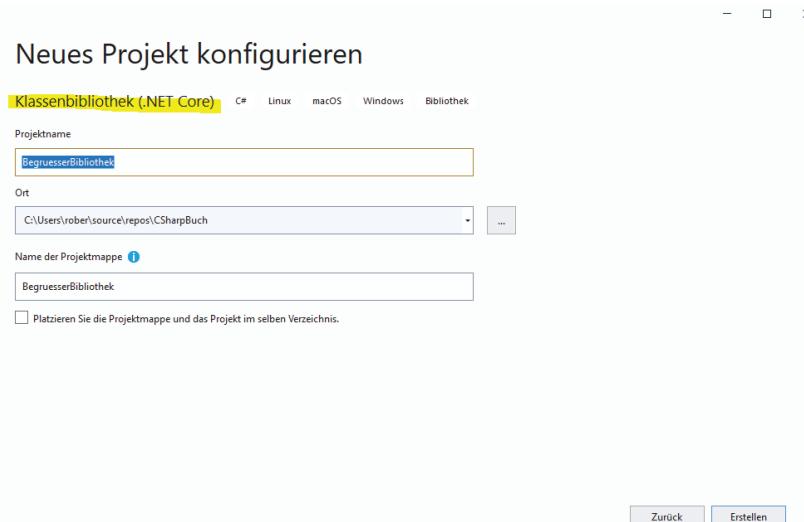


Abb. 13.6.1 Anlegen der Klassenbibliothek BegruesserBibliothek

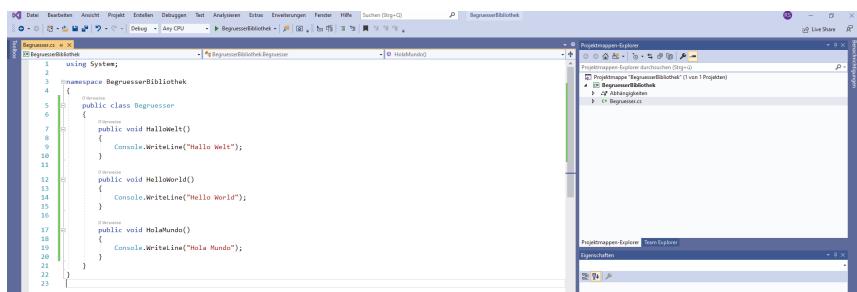


Abb. 13.6.2 Die Klasse BegruesserBibliothek als Klassenbibliothek

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

## Musterlösung für Teilaufgabe 2:

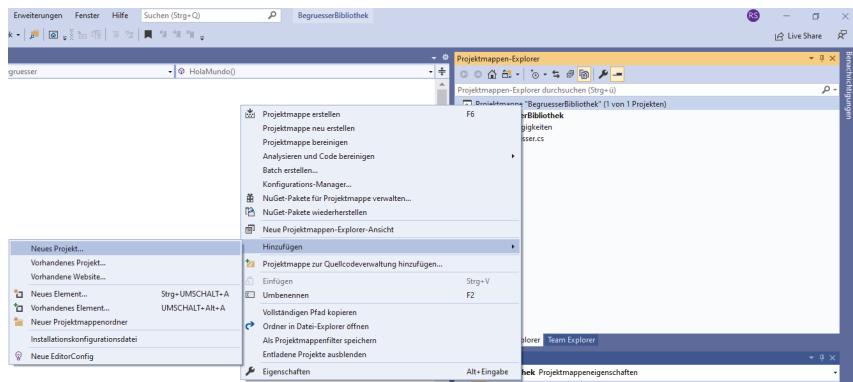


Abb. 13.6.3 Ein Projekt zur Projektmappe BegruesserBibliothek hinzufügen

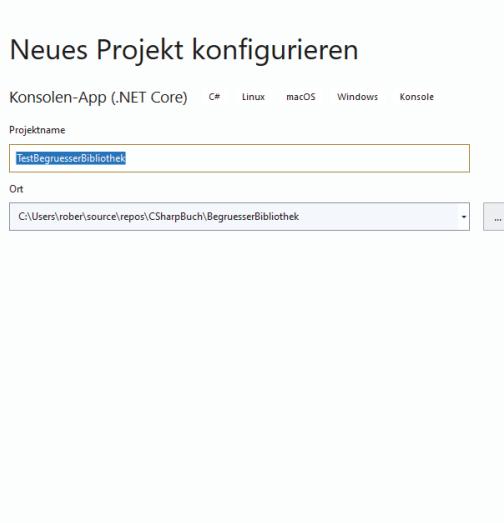


Abb. 13.6.4 Die Konsolen-App TestBegruesserBibliothek erstellen

## 13.6 Übungsaufgabe: Eine eigene Klassenbibliothek erstellen

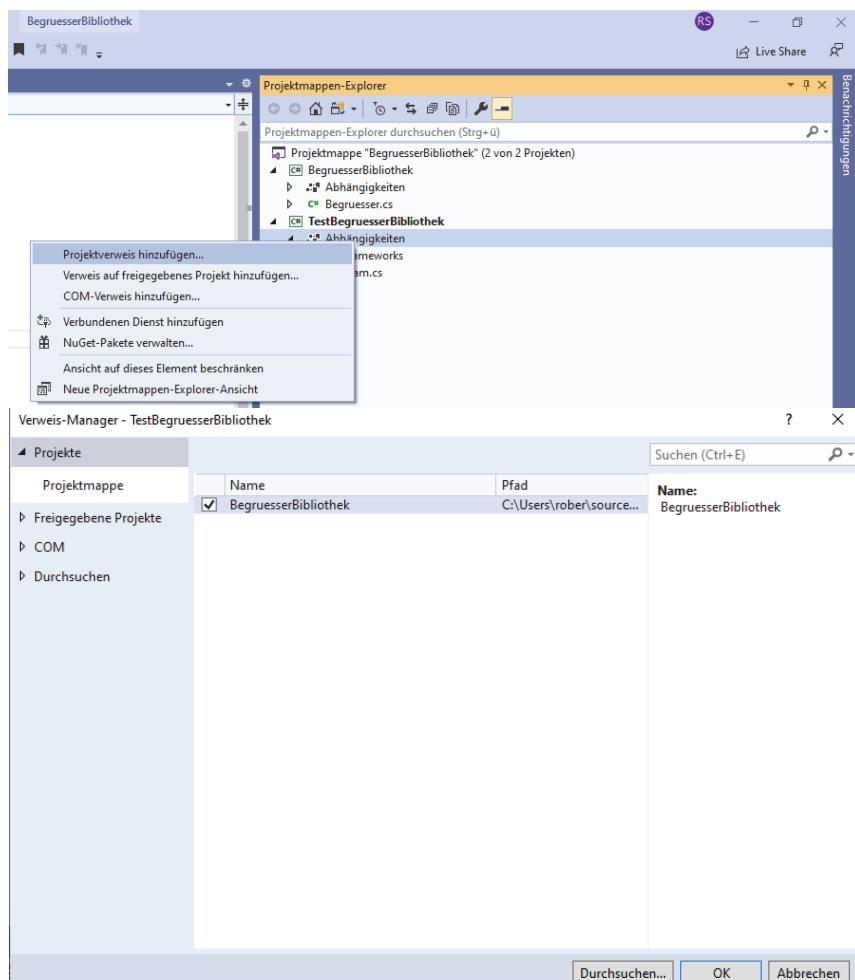
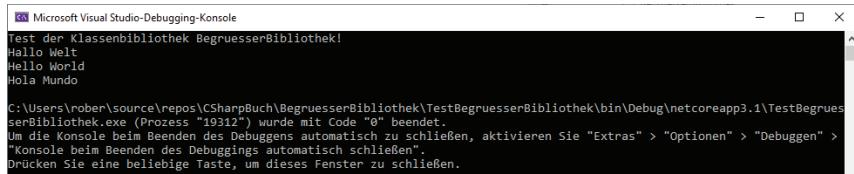


Abb. 13.6.6 Die Klassenbibliothek BegruesserBibliothek auswählen

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

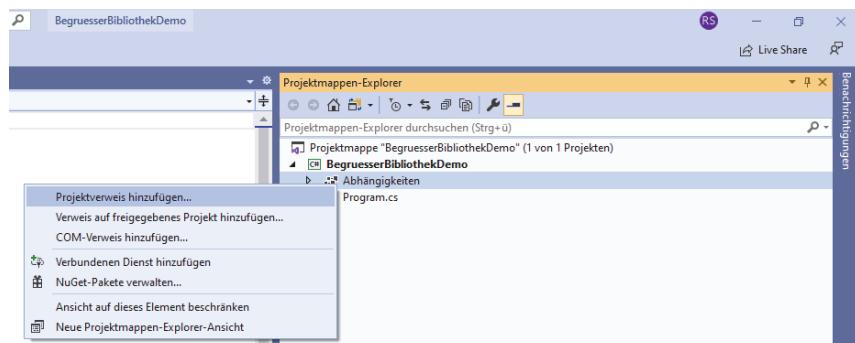
Testprogramm für die Klassenbibliothek BegruesserBibliothek:

```
1  using BegruesserBibliothek;
2  using System;
3
4  namespace TestBegruesserBibliothek
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              Console.WriteLine("Test der Klassenbibliothek
11                  BegruesserBibliothek!");
12              var begruesser = new Begruesser();
13              begruesser.HalloWelt();
14              begruesser.HelloWorld();
15              begruesser.HolaMundo();
16          }
17      }
18 }
```



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar reads "Microsoft Visual Studio-Debugging-Konsole". The console output displays the following text:  
Test der Klassenbibliothek BegruesserBibliothek!  
Hallo Welt  
Hello World  
Hola Mundo  
  
C:\Users\rober\source\repos\CSharpBuch\BegruesserBibliothek\TestBegruesserBibliothek\bin\Debug\netcoreapp3.1\Tes...  
serBibliothek.exe (Prozess "19312") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >  
"Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

Abb. 13.6.7 Bildschirmausgabe des Testprogramms

**Musterlösung für Teilaufgabe 3:****Abb. 13.6.8** Erstellen der Konsolen-App BegruesserBibliothekDemo**Abb. 13.6.9** Einen Projektverweis zur Projektmappe BegruesserBibliothekDemo hinzufügen

## 13 Visual Studio reloaded: Funktionalitäten für Fortgeschrittene

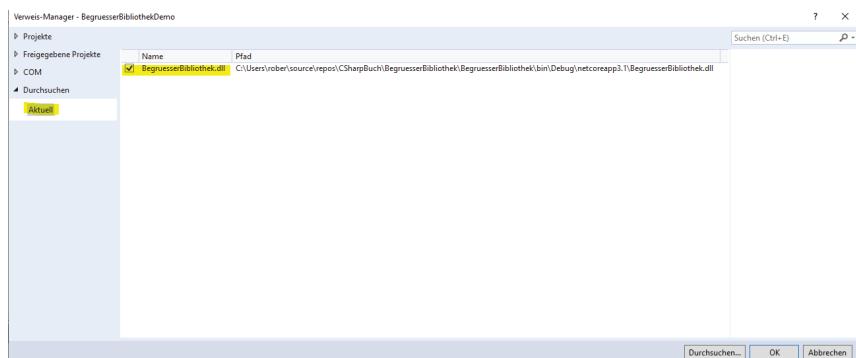


Abb. 13.6.10 Die kompilierte BegrueßerBibliothek.dll auswählen

Demoprogramm für die Klassenbibliothek BegrueßerBibliothek:

```

1  using BegrueßerBibliothek;
2  using System;
3  namespace BegrueßerBibliothekDemo
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Demo der Klassenbibliothek
10             BegrueßerBibliothek!");
11             var begrueßer = new Begrueßer();
12             begrueßer.HalloWelt();
13             begrueßer.HelloWorld();
14             begrueßer.HolaMundo();
15         }
16     }
17 }
```

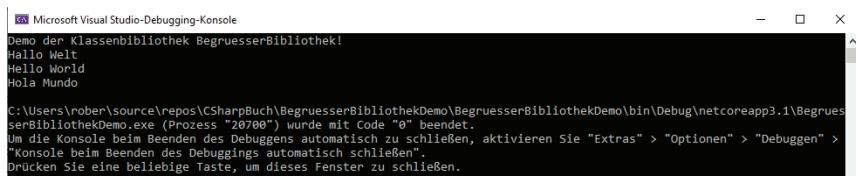


Abb. 13.6.11 Bildschirmausgabe des Demoprogramms

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 14

# Dateizugriff mit C#

Bisher haben all unsere Programme ihre Eingangsdaten von der Tastatur erhalten und ihre Ausgangsdaten am Bildschirm ausgegeben. Mit diesem Kapitel werden wir eine weitere Möglichkeit kennenlernen, wie C#-Programme Daten einlesen beziehungsweise ausgeben können. Wir werden uns ansehen, wie in C# Dateien gelesen und geschrieben werden können. Aus Platzgründen wollen wir uns auf Textdateien beschränken. Des Weiteren werden wir lernen, wie man XML-Dateien verarbeitet. Da man mit XML auch Objekte beschreiben kann, betrachten wir im Unterkapitel „Objekte serialisieren und deserialisieren“, wie Objekte in XML und wieder zurück konvertiert werden können.

### 14.1 Textdateien lesen und schreiben

Wir starten mit dem Lesen von Textdateien. Textdateien sind im Klartext lesbare Dateien, die mit jedem beliebigen Editor bearbeitet werden können. Auch bei unseren Programmdateien handelt es sich um Textdateien. Typischerweise sind Textdateien zeilenweise aufgebaut. Eine Textdatei unter Windows enthält am Ende jeder Zeile die nicht druckbaren Zeichen „Carriage Return“ (Wagenrücklauf) und „Line Feed“ (Zeilenvorschub). Diese Bezeichnungen stammen noch aus Zeiten von elektronischen Schreibmaschinen und Nadeldruckern. Unter dem Betriebssystem Unix enden Zeilen in Textdateien nur dem Zeichen „Line Feed“. Wir werden im Weiteren die Windows-Variante verwenden und die Zeilen in unseren Textdateien mit „Carriage Return“ und „Line Feed“ enden lassen. Das String-Literal für die beiden Zeichen ist „\r\n“.

Um eine Datei lesen zu können, benötigen wir eine Datei. In unserem ersten Praxisbeispiel verwenden wir eine Datei, die dazu dient, ein paar Konfigurationseinstellungen für ein Programm zu speichern. In dieser Datei speichern wir jede Konfigurationseinstellung in einer eigenen Zeile. Eine Konfigurationseinstellung hat die Form: Variable=Wert.

Wir erstellen eine neue Konsolen-App und fügen ihr eine Textdatei hinzu. Indem wir mit der rechten Maustaste auf unser Projekt klicken und „Hinzufügen/Neues Element ...“ aus dem Kontextmenü auswählen.

## 14.1 Textdateien lesen und schreiben

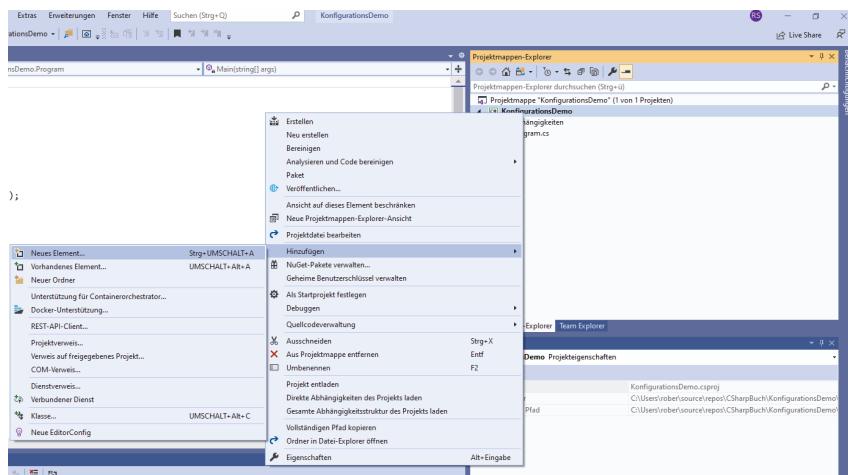


Abb. 14.1.1 Das Kontextmenü „Hinzufügen/Neues Element ...“

Im Dialog „Neues Element hinzufügen“ wählen wir als Vorlagen-Kategorie „installiert/Visual C#-Elemente/Allgemein“ aus und als Vorlage wählen wir „Textdatei“. Für den Namen der Textdatei vergeben wir „Konfiguration.txt“.

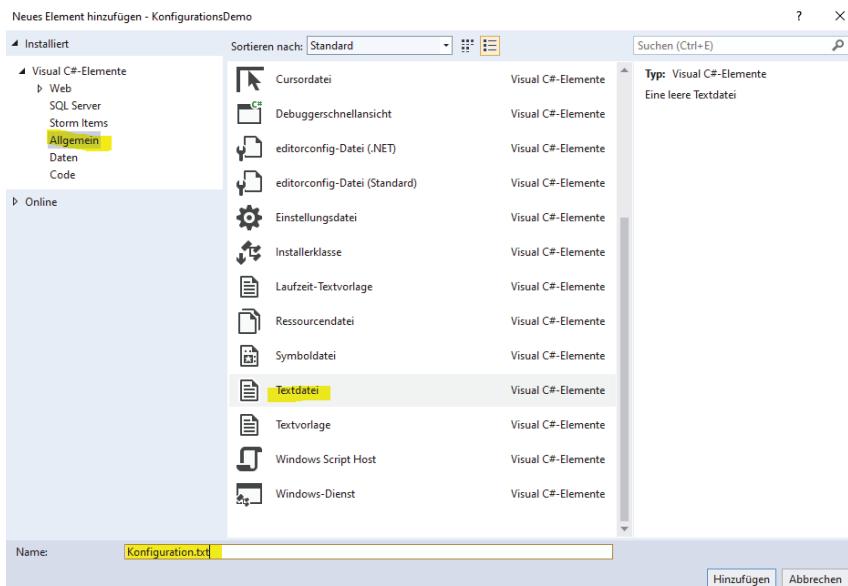


Abb. 14.1.2 Hinzufügen einer Textdatei

## 14 Dateizugriff mit C#

Dann klicken wir auf die Schaltfläche „Hinzufügen“.

Als nächstes klicken wir auf die hinzugefügte Datei und stellen im Panel Eigenschaften das Feld „In Ausgabeverzeichnis kopieren“ auf den Wert „Kopieren, wenn neuer“. Falls bei Ihnen das Panel Eigenschaften nicht sichtbar ist, klicken Sie mit der rechten Maustaste auf die hinzugefügte Datei und wählen das Kontextmenü Eigenschaften aus.

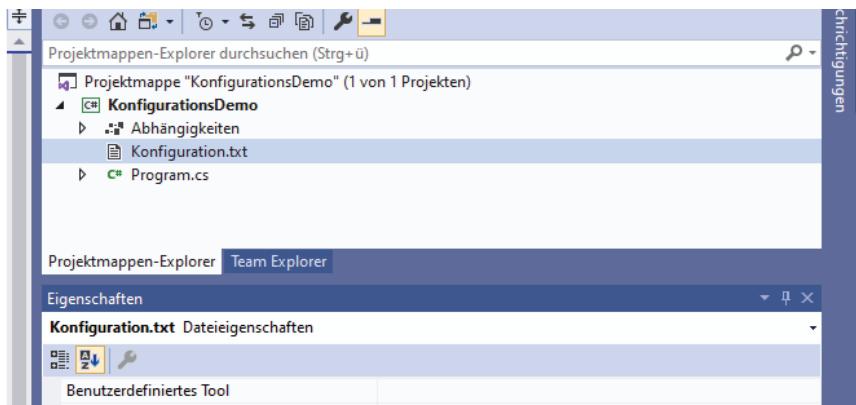


Abb. 14.1.3 Die Dateieigenschaft „In Ausgabeverzeichnis kopieren“

Wenn wir ein Programm in Visual Studio starten, wird es kompiliert und im Ausgabeverzeichnis des Projekts abgelegt und von dort gestartet. Mit obiger Einstellung sorgen wir dafür, dass unsere Textdatei beim Kompilieren in das Ausgabeverzeichnis mit kopiert wird, aber nur wenn sich die Textdatei geändert hat. Wir benötigen diese Einstellung, weil wir unsere Textdatei zum Lesen im gleichen Verzeichnis erwarten werden, aus dem unser Programm gestartet wurde. Dadurch wird unser Programm verschiebbar. Das heißt, wenn wir das Programm ausliefern, kann man es in jedes beliebige Verzeichnis kopieren und es wird immer funktionieren.

In unserer Konfigurationsdatei erfassen wir folgen Inhalt.

```
1 Benutzer = Robert Schiefele
2 SchuhGröße = 42
```

Mit dem folgenden kleinen Demoprogramm können wir die Datei lesen.

```
1 using System;
2 using System.IO;
3
4 namespace KonfigurationsDemo
5 {
6     class Program
```

```

7  {
8      static void Main(string[] args)
9      {
10         string text = File.ReadAllText("Konfiguration.txt");
11         Console.WriteLine(text);
12     }
13 }
14 }
```

Wir binden die Klassenbibliothek `System.IO` ein, welche die benötigten Klassen für den Dateizugriff bereitstellt. Das Einlesen der Datei erledigen wir mit der statischen Methode `ReadAllText()` der Klasse `File`. Wir übergeben der Methode den Namen der zu lesenden Datei ohne Pfadangabe. Das heißt, die Datei wird in dem Verzeichnis erwartet, aus dem das Programm gestartet wurde. Dass unsere Datei auch wirklich dort existiert, haben wir mit einer Einstellung in den Eigenschaften der Datei sichergestellt. Wenn wir der Methode `ReadAllText()` zum Beispiel „C:\MeinVerzeichnis\Konfiguration.txt“ übergeben würden, dann müssten wir die Datei `Konfiguration.txt` immer im Verzeichnis `MeinVerzeichnis` im Laufwerk C: bereitstellen. In diesem Fall spricht man von einer absoluten Pfadangabe. Wir könnten aber auch die Methode `ReadAllText()` mit „`MeinVerzeichnis\Konfiguration.txt`“ aufrufen. Dann würde die Datei `Konfiguration.txt` im Unterverzeichnis `MeinVerzeichnis` unterhalb des Verzeichnisses erwartet werden, in dem sich die ausführbare Datei befindet. Diese Form wird relative Pfadangabe genannt.

Die Methode `ReadAllText()` gibt den vollständigen Dateiinhalt als String zurück. In unserem Beispielprogramm geben wir diesen Inhalt zunächst nur am Bildschirm aus.



```

Benutzer = Robert Schiefele
SchuhGröße = 42
C:\Users\rober\source\repos\cSharpBuch\KonfigurationsDemo\KonfigurationsDemo\bin\Debug\netcoreapp3.1\KonfigurationsDemo.exe (Prozess "16600") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
```

Abb. 14.1.4 Bildschirmausgabe der Datei `Konfiguration.txt`

In realistischen Programmierszenarios möchte man aber so eine Datei nicht einfach nur am Bildschirm ausgeben, sondern die beiden Informationen „`Benutzer = Robert Schiefele`“ und „`Schuhgröße = 42`“ auch einzeln verarbeiten können. Natürlich könnten wir den eingelesenen String mit der Methode `String.Split()` zeilenweise aufspalten. Aber als Programmierer wissen wir nicht, ob wir bei unseren Benutzern mit einer Windows-Datei oder einer Unix-Datei rechnen müssen. Daher müssten wir berücksichtigen, dass die eingelesene Datei sowohl die Zeichenfolge „`\r\n`“ als auch nur das Zeichen „`\n`“ als Zeilenende Kennzeichen verwenden könnte. Diese Arbeit können wir aber auch dem .NET-Framework überlassen und die Methode `ReadAllLines()` zum Einlesen der Datei verwenden. Die Funktionsweise dieser Methode wollen wir mit dem folgenden Beispielprogramm untersuchen:

## 14 Dateizugriff mit C#

```

1  using System;
2  using System.IO;
3
4  namespace KonfigurationsDemo
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             string[] zeilen = File.ReadAllLines("Konfiguration.
11             txt");
12
13             foreach( var zeile in zeilen)
14             {
15                 var zeilenElemente = zeile.Split("=");
16
17                 if (zeilenElemente.Length != 2)
18                 {
19                     continue;
20                 }
21                 var variable = zeilenElemente[0].Trim();
22                 var wert = zeilenElemente[1].Trim();
23                 Console.WriteLine($"Die Variable '{variable}' hat
24                 den Inhalt '{wert}'");
25             }
26         }
27     }
28 }
```

Die Methode `ReadAllLines()` liefert ein String-Array zurück. Das Array enthält jede Zeile der Datei in einem Element. In unserem Beispielprogramm durchlaufen wir das Array mit einer Schleife und spalten jede Zeile am Zeichen „=“ auf. Dadurch Zerlegen wir die Zeile in die Teile Variable und Wert. Wenn sich eine Zeile nicht in genau zwei Elementen aufspalten lässt, ist sie ungültig und wird ignoriert. Zusätzlich sorgen wir mit der Methode `string.Trim()` dafür, dass bei der Variablen und dem Wert die führenden und abschließenden Leerzeichen weggeschnitten werden. Zum Schluss geben wir die Variable und den Wert für jede Zeile der Datei `Konfiguration.txt` am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
Die Variable 'Benutzer' hat den Wert 'Robert Schiefele'
Die Variable 'SchuhGröße' hat den Wert '42'

C:\Users\rober\source\repos\CSharpBuch\KonfigurationsDemo\KonfigurationsDemo\bin\Debug\netcoreapp3.1\KonfigurationsDemo.exe (Prozess "12304") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 14.1.5 Ausgabe der Variablen Benutzer und Schuhgröße

Bei den Methoden `ReadAllText()` und `ReadAllLines()` gibt es eine Besonderheit zu beachten: Beide Methoden lesen die gesamte Datei auf einmal in den Hauptspeicher des Computers ein. Das heißt, wenn wir eine 10 Gigabyte große Datei mit

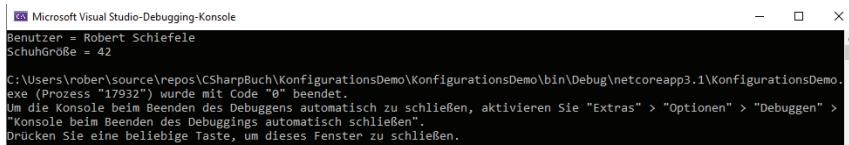
einer dieser Methode einlesen, kann das nicht nur ein bisschen dauern, sondern wir belegen auch 10 Gigabyte im Hauptspeicher, was - je nach Leistungsfähigkeit des betreffenden Computers - das ganze System schlagartig verlangsamen kann. Daher gibt es im .NET-Framework auch eine Technik, mit der wir eine Textdatei Zeile für Zeile von der Festplatte lesen können. Diese Technik wollen wir im nächsten Beispielprogramm näher betrachten.

```
1  using System;
2  using System.IO;
3
4  namespace KonfigurationsDemo
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var dateiInfo = new FileInfo("Konfiguration.txt");
11             using(StreamReader leser = dateiInfo.OpenText())
12             {
13                 while(!leser.EndOfStream)
14                 {
15                     var zeile = leser.ReadLine();
16                     Console.WriteLine(zeile);
17                 }
18             }
19         }
20     }
21 }
```

Wir benötigen wieder die Klassenbibliothek `System.IO`, die alle für diese Technik benötigten Klassen enthält. Zuerst erzeugen wir eine Instanz der Klasse `FileInfo`. Dazu übergeben wir dem Konstruktor den Namen und Pfad für die zu lesende Datei. Hierbei gelten die gleichen Regeln, die wir schon bei den Methoden `ReadAllText()` und `ReadAllLines()` kennengelernt haben. Dann rufen wir die Methode `OpenText()` des zuvor erzeugten `FileInfo`-Objekts auf, um eine Instanz der Klasse `StreamReader` zu erzeugen. Die Anweisung `StreamReader leser = dateiInfo.OpenText()` ist in runde Klammern eingeschlossen und hat das Schlüsselwort `using` vorangestellt. Danach folgt ein Anweisungsblock in geschweiften Klammern. Die Verwendung von `using` ist zwar für unser kleines Beispielprogramm nicht zwingend notwendig, aber wir sollten ein `StreamReader`-Objekt immer über das Schlüsselwort `using` erstellen. Beim Thema Garbage Collector haben wir schon gelernt, dass der Garbage Collector nur Ressourcen wieder freigegeben kann, die vom .NET-Framework verwaltet werden. Eine Datei ist eine Ressource, die vom Betriebssystem verwaltet wird. Wenn wir eine Instanz eines Objektes, das vom .NET-Framework nicht verwaltete Ressourcen verwendet, mit Hilfe des Schlüsselworts `using` erstellen, sorgen wir dafür, dass diese Ressourcen wieder freigegeben werden, sobald der Anweisungsblock, der auf die `using`-Anweisung folgt, wieder verlassen wird. Jetzt haben wir ein Objekt, mit dem wir eine Datei sequenziell mit einer Schleife durchlaufen können. Die Methode

## 14 Dateizugriff mit C#

`ReadLine()` der Klasse `StreamReader` liest eine Zeile einer Datei und gibt die Zeile als String zurück. Zudem sorgt die Methode `ReadLine()` dafür, dass beim nächsten Aufruf der Methode die nächste Zeile gelesen wird. Die `bool`-Property `EndOfStream` der Klasse `StreamReader` gibt an, ob noch mindestens eine zu lesende Zeile existiert. In unserem Beispielprogramm lesen wir eine Zeile und geben Sie danach gleich wieder am Bildschirm aus.



```
Microsoft Visual Studio-Debugging-Konsole
Benutzer = Robert Schiefele
SchuhGröße = 42

C:\Users\rober\source\repos\CSharpBuch\KonfigurationsDemo\KonfigurationsDemo\bin\Debug\netcoreapp3.1\KonfigurationsDemo.exe (Prozess "17932") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

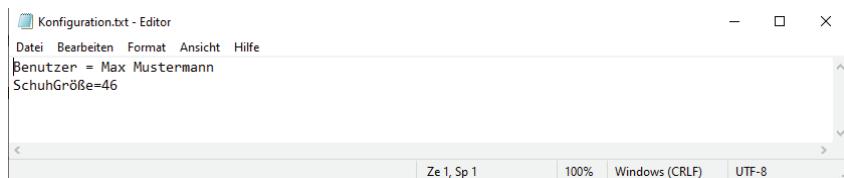
Abb. 14.1.6 Bildschirmausgabe bei zeilenweisem Lesen

Zu den drei hier vorgestellten Methoden, eine Textdatei zu lesen, gibt es analog drei Methoden eine Textdatei zu schreiben. Die erste Methode heißt `File.WriteAllText()`:

```
1 using System;
2 using System.IO;
3
4 namespace KonfigurationsDemo
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var text= "Benutzer = Max Mustermann\r\n"
11             + "SchuhGröße=46\r\n";
12             File.WriteAllText("Konfiguration.txt", text);
13         }
14     }
15 }
```

Wir erzeugen die String-Variablen `text` und weisen ihr ein String Literal zu. Beachten Sie dabei, dass wir „\r\n“ als Zeilenende Kennzeichen verwenden. Das heißt, wir werden eine Windows-Datei schreiben. Der statischen Methode `WriteAllText()` der Klasse `File` übergeben wir als ersten Parameter den Namen und Pfad der zu schreibenden Datei. Auch hier gelten für den Pfad der Datei wieder die gleichen Regeln wie bei der Methode `ReadAllText()`. Als zweiten Parameter übergeben wir die string-Variablen `text`, die den vollständigen Text der zu schreibenden Datei enthält. Falls die zu schreibende Datei schon existiert, wird sie ohne Vorwarnung überschrieben.

Wenn wir das Programm ausführen und dann den Windows Datei-Explorer öffnen und zum Ausgabeverzeichnis unseres Visual Studio Projekts wechseln, sehen wir die Datei „Konfiguration.txt“. Mit einem Doppelklick können wir den Inhalt der Datei überprüfen.



**Abb. 14.1.7** Die Datei Konfiguration.txt im Editor

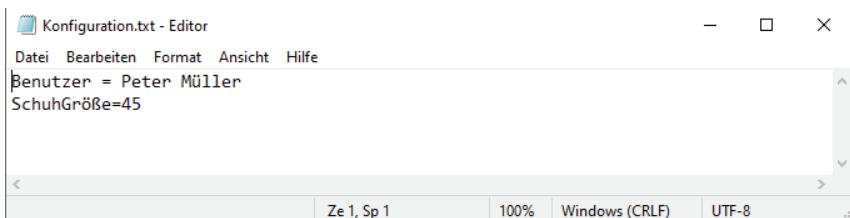
Wenn wir in unserem Programm die zu schreibenden Daten nicht in einem einzigen String, sondern zeilenweise in einem String-Array vorliegen haben, können wir sie mit der Methode `File.WriteAllText()` in eine Textdatei schreiben.

```

1  using System;
2  using System.IO;
3
4  namespace KonfigurationsDemo
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              string[] text =
11              {
12                  "Benutzer = Peter Müller",
13                  "SchuhGröße=45"
14              };
15
16              File.WriteAllText("Konfiguration.txt", text);
17          }
18      }
19 }
```

Wir erzeugen ein String-Array, das für jede Zeile, die wir schreiben wollen, ein Element enthält. Beachten Sie dabei, dass unsere Zeilen kein Zeilenende-Kennzeichen enthalten. Das Zeilenende-Kennzeichen wird von der Methode `File.WriteAllText()` gesetzt. Das heißt, wenn unser Programm unter Windows läuft, wird unsere Datei „\r\n“ als Zeilenende-Kennzeichen erhalten und wenn unser Programm unter Unix läuft, wird die Datei „\n“ als Zeilenende-Kennzeichen erhalten. Der statischen Methode `WriteAllLines()` der Klasse `File` übergeben wir als ersten Parameter den Namen und Pfad der Datei. In unserem Beispiel ist das nur der Name der Datei relativ zum Ausgabeverzeichnis unseres Visual Studio Projekts. Als zweiten Parameter erhält die Methode das String-Array `text`, das die zu schreibenden Zeilen als Elemente enthält. Wenn wir das Programm starten, können wir im Ausgabeverzeichnis des Projekts den Inhalt der Datei Konfiguration.txt überprüfen.

## 14 Dateizugriff mit C#



**Abb. 14.1.8** Die Datei Konfiguration.txt im Editor

Die beiden Methoden `WriteAllText()` und `WriteAllLines()` schreiben eine ganze Datei auf einmal. Daher gibt es im .NET-Framework analog zum Lesen von Dateien auch eine Technik, mit der eine Datei zeilenweise direkt auf die Festplatte geschrieben werden kann. Betrachten wir dazu das folgende Beispielprogramm:

```

1  using System;
2  using System.IO;
3
4  namespace KonfigurationsDemo
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              string[] text =
11              {
12                  "Benutzer = Marta Musterfrau",
13                  "SchuhGröße=37"
14              };
15
16              var dateiInfo = new FileInfo("Konfiguration.txt");
17              using (FileStream dateiStrom = dateiInfo.OpenWrite())
18              {
19                  using (StreamWriter schreiber = new
20                      StreamWriter(dateiStrom))
21                  {
22                      foreach (var zeile in text)
23                      {
24                          schreiber.WriteLine(zeile);
25                      }
26                      schreiber.Flush();
27                      schreiber.Close();
28                  }
29              }
30          }
31      }
32  }
```

Auch hier binden wir zunächst die Klassenbibliothek `System.IO` ein. Die zu schreibenden Zeilen befinden sich im String-Array `text`. Wir erzeugen eine Instanz der Klasse `FileInfo` und übergeben dem Konstruktor wieder den Namen und den Pfad

der zu schreibenden Datei. In unserem Beispiel übergeben wir nur den Namen der Datei, da wir die Datei in das Verzeichnis schreiben wollen, in dem sich die ausführbare Programmdatei befindet. Dann erzeugen wir mit Hilfe der using-Anweisung eine Instanz der Klasse `FileStream`, indem wir die Methode `OpenWrite()` des Objekts `dateiInfo` aufrufen. Dadurch sorgen wir dafür, dass die nicht verwalteten Ressourcen der Klasse `FileStream` wieder freigegeben werden, wenn sie nicht mehr gebraucht werden. Im Anweisungsblock der ersten using-Anweisung erzeugen wir mit einer zweiten using-Anweisung eine neue Instanz der Klasse `StreamWriter` mit dem Schlüsselwort `new`. Dem Konstruktor der Klasse `StreamWriter` übergeben wir die zuvor erzeugte Instanz der Klasse `FileStream`. Dieses zweistufige Erzeugen eines `StreamWriter`-Objekts ist notwendig, da uns die Klasse `FileInfo` nur eine Methode zum Erzeugen eines `StreamReader`-Objekts, aber leider keine Methode zum Erzeugen eines `StreamWriter`-Objekts zur Verfügung stellt. Im Anweisungsblock der zweiten using-Anweisung durchlaufen wir das String-Array `text` und schreiben mit der Methode `WriteLine()` des `StreamWriter`-Objekts die jeweilige Textzeile in unsere Ausgabedatei. Beachten Sie dabei, dass wir kein Zeilenende-Kennzeichen festlegen. Das wird von der Methode `WriteLine()` erledigt. Damit erzeugen wir das korrekte Zeilenende-Kennzeichen für Windows oder Unix - je nachdem, unter welchem Betriebssystem unser Programm läuft. Nach unserer Schleife rufen wir die Methoden `Flush()` und `Close()` des `StreamWriter`-Objekts auf. Die Methode `Flush()` erzwingt das sofortige Schreiben des internen Pufferspeicher des `StreamWriter`-Objekts. Die Methode `Close()` gibt die Datei wieder frei, so dass sie von unserem Programm und anderen eventuell gleichzeitig auf dem Computer laufenden Programmen wieder geöffnet werden kann. Durch den Aufruf dieser Methoden stellen wir sicher, dass unsere Datei direkt nach dem Schreiben sofort wieder geöffnet werden kann und dass sie vollständig ist, auch wenn unser Programm direkt nach dem Schreiben beendet wird. Wenn wir unser Beispielprogramm starten, können wir im Ausgabeverzeichnis des Projekts die Datei `Konfiguration.txt` überprüfen.

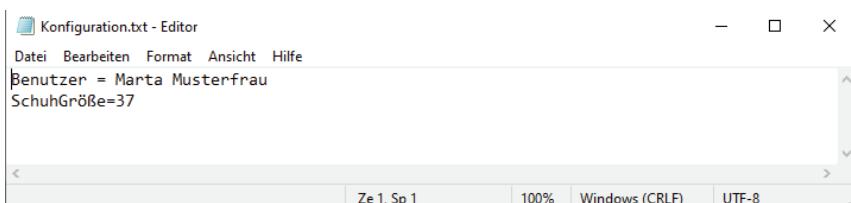


Abb. 14.1.9 Die zeilenweise geschriebene Datei Konfiguration.txt

Zum Abschluss lernen wir noch eine spezielle Variante des Schreibens einer Datei kennen. Diese Methode wird zum Beispiel benötigt, wenn Sie in einem Programm bei bestimmten Vorkommnissen einen Logeintrag in eine Datei schreiben wollen. Dabei wollen wir den Logeintrag immer an die gleiche Datei anhängen. Zum Anhängen

## 14 Dateizugriff mit C#

eines Textes an eine Datei verwenden wir die statische Methode AppendAllText () der Klasse File aus der Klassenbibliothek System.IO.

Das folgende Beispielprogramm zeigt die Verwendung dieser Methode:

```
1  using System;
2  using System.IO;
3
4  namespace LogDemo
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             string logDatei = "Log.txt";
11             File.AppendAllText(logDatei, "1. Logeintrag\r\n");
12             File.AppendAllText(logDatei, "2. Logeintrag\r\n");
13             File.AppendAllText(logDatei, "3. Logeintrag\r\n");
14         }
15     }
16 }
```

Wir speichern den Namen und Pfad der Datei, an die wir den Text anhängen wollen in der String-Variablen logDatei. Wir begnügen uns auch hier wieder mit dem Namen der Datei. Dadurch wird eine Datei im Ausgabeverzeichnis des Projekts verwendet. Dann rufen wir drei Mal die statische Methode AppendAllText () der Klasse File auf, um eine Zeile an die Datei Log.txt anzuhängen. Der Methode AppendAllText () übergeben wir als ersten Parameter die Variable logDatei und als zweiten Parameter die anzuhängende Zeile als String-Literal. Beachten Sie dabei, dass wir hier wieder das Zeilenende-Kennzeichen selbst festlegen müssen. Dass die Datei beim ersten Aufruf der Methode AppendAllText () noch gar nicht existiert, ist kein Problem, da diese Methode selbstständig eine Datei erzeugt, wenn die Datei noch nicht existiert. Wenn wir unser Programm starten, können wir im Ausgabeverzeichnis des Projekts die Datei Log.txt überprüfen.

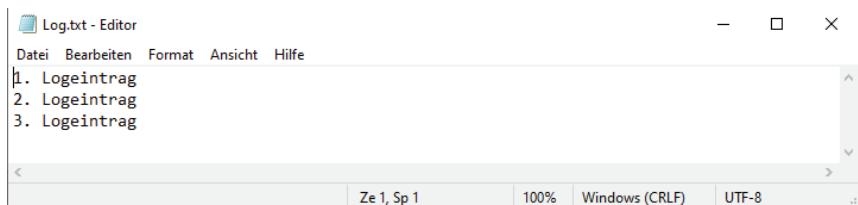


Abb. 14.1.10 Die Datei Log.txt

## 14.2 XML-Dateien verarbeiten

Im vorherigen Unterkapitel haben wir gelernt, wie wir mit Hilfe von C# Textdateien im Allgemeinen lesen und schreiben können. Dabei haben wir auf spezielle Formate von Textdateien keinen Wert gelegt. In diesem Unterkapitel gehen wir auf Textdateien ein, die im sogenannten XML-Format vorliegen, da es im .NET-Framework eine spezielle Unterstützung für dieses Format gibt. Da das Thema XML sehr umfangreich ist, wenn man es vollständig behandeln möchte, werden wir im Folgenden nur einen kleinen Crashkurs für XML absolvieren, um denjenigen Lesern, die bisher noch nicht mit XML zu tun hatten, wenigstens ein paar Basiskenntnisse zu vermitteln. XML ist ein textbasiertes Format, um strukturierte Daten zu speichern. Eine XML-Datei enthält dabei sowohl die Beschreibung der Struktur der Daten als auch die Daten selbst. Am einfachsten wird einem das Prinzip von XML an einem Beispiel klar: Dazu erstellen wir uns eine neue Konsolen-Applikation in Visual Studio. Wir klicken mit der rechten Maustaste auf das Projekt und wählen das Kontextmenü „Hinzufügen/Neues Element...“ aus. Der Dialog „Neues Element hinzufügen“ erscheint.

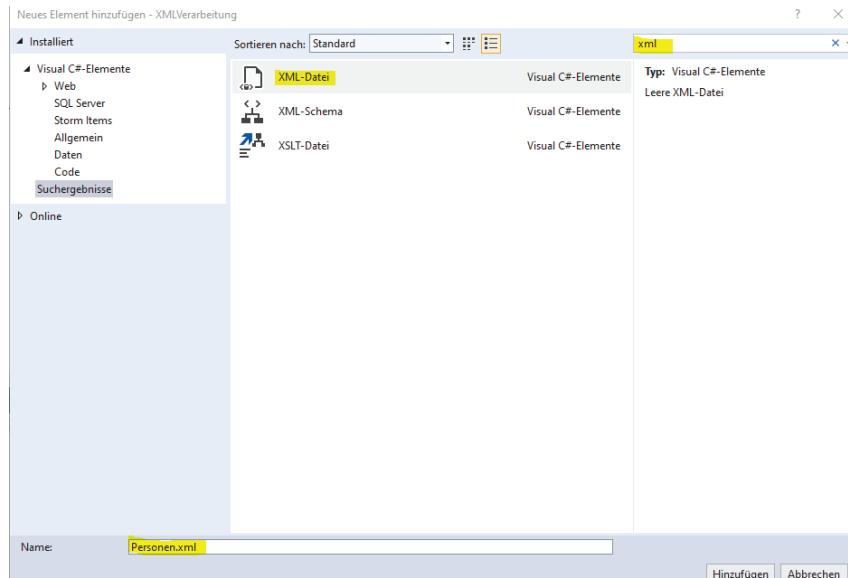


Abb. 14.2.1 Eine XML-Datei hinzufügen

Im Suchfeld rechts oben geben wir den Suchbegriff „xml“ ein und dann wählen wir XML-Datei aus den Suchergebnissen aus. Als Namen für die XML-Datei vergeben wir Personen.xml und drücken die Schaltfläche Hinzufügen. Bei den Datei-Eigenschaften stellen wir das Feld „In Ausgabeverzeichnis kopieren“ auf den Wert „Kopieren, wenn neuer“. Als Inhalt für die Datei Personen.xml geben wir den folgenden Text ein:

## 14 Dateizugriff mit C#

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <Personen>
3   <Person Id="1">
4     <Vorname>Robert</Vorname>
5     <Nachname>Schiefele</Nachname>
6   </Person>
7   <Person Id="2">
8     <Vorname>Max</Vorname>
9     <Nachname>Mustermann</Nachname>
10    </Person>
11   <Person Id="3">
12     <Vorname>Marta</Vorname>
13     <Nachname>Musterfrau</Nachname>
14   </Person>
15 </Personen>
```

Die erste Zeile enthält den sogenannten Prolog, er darf in keiner XML-Datei fehlen. Der Prolog zeigt an, dass es sich um eine XML-Datei handelt. Zusätzlich enthält unser Prolog die Information, dass wir eine XML-Syntax in der Version 1.0 verwenden und dass der Text utf-8 kodiert ist. Nach dem Prolog kommt der erste XML-Knoten, der Wurzelknoten oder auch Root-Knoten genannt wird. Eine XML-Datei kann nur einen Wurzelknoten enthalten. Ein XML-Knoten beginnt mit seinem Namen, der in spitze Klammern eingeschlossen wird. In unserem Fall `<Personen>`. Ein XML-Knoten endet mit seinem Namen, dem ein Schrägstrich vorangestellt wird und der in spitze Klammern eingeschlossen wird. In unserem Fall `</Personen>`. Zwischen Anfang und Ende eines Knotens befindet sich der Inhalt des Knotens. Der Inhalt kann aus einfachem Text oder aus einem oder mehreren Unterknoten bestehen, wobei jeder Unterknoten wieder weitere Unterknoten enthalten kann und so weiter. Wenn sich eine XML-Datei an all diese Regeln hält, nennt man sie well-formed. In unserem Beispiel heißt der Wurzelknoten Personen und enthält drei Unterknoten, die Person heißen. Ein Person-Unterknoten enthält zusätzlich noch ein sogenanntes Attribut. Ein Attribut hat einen Namen, in unserem Beispiel `Id`. Jedes Attribut hat auch einen Wert, der dem Attribut mit dem `=`-Operator zugewiesen wird. Der Wert eines Attributs ist immer von doppelten Anführungszeichen eingeschlossen. Ein XML-Knoten kann beliebig viele Attribute haben. Er muss aber kein Attribut haben. Jeder Person-Unterknoten enthält zwei attributlose Unterknoten, die Vorname und Nachname heißen. Die Vorname- und Nachname-Unterknoten enthalten Daten in Textform. Nach diesem Schnelleinstieg in das XML-Format wollen wir uns ansehen, wie man eine XML-Datei mit C# einlesen kann. Hier lernen wir den im Kapitel „Was ist Linq?“ erwähnten Linq-Provider „Linq to XML“ näher kennen.

```
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Xml.Linq;
6
7 namespace XMLVerarbeitung
```

```
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             XElement wurzel = XElement.Load("Personen.xml");
14             IEnumerable< XElement> xmlPersonen = wurzel.
15             Descendants("Person");
16
17             foreach(XElement xmlPerson in xmlPersonen)
18             {
19                 var id = xmlPerson.Attribute("Id").Value;
20                 var vorname = xmlPerson.Descendants("Vorname").
21                 First().Value;
22                 var nachname = xmlPerson.Descendants("Nachname").
23                 First().Value;
24                 Console.WriteLine($"Id {id}; Vorname: {vorname};".
25                               Nachname: {nachname}");
26             }
27         }
28     }
29 }
```

Zuerst rufen wir die statische Methode `Load()` der Klasse `XElement` auf. Der Methode übergeben wir den Namen der XML-Datei, die wir lesen wollen. Unsere zu lesende Datei befindet sich im Ausgabeverzeichnis des Projekts. Die Methode `Load()` gibt ein Objekt vom Typ `XElement` zurück, welches einen XML-Knoten repräsentiert. Der von `Load()` zurückgegebene Knoten entspricht dem Wurzelknoten der XML-Datei, in unserem Beispiel ist das der Knoten Personen. Den zurückgegebenen Wurzelknoten speichern wir in der Variablen `wurzel`. Als Nächstes erfolgt ein Aufruf der Methode `Descendants()`. Dieser Methode übergeben wir das String-Literal „Person“. Die Methode `Descendants()` hat den Rückgabetyp `IEnumerable< XElement>`, das heißt, sie liefert eine Liste von XML-Knoten zurück. In unserem Fall enthält sie die Liste alle XML-Knoten, die sich unterhalb des Knotens `wurzel` befinden und deren Name `Person` ist. Beachten Sie dabei das `Descendants()` wirklich alle gesuchten Knoten zurückliefert, auch wenn sie sich in einer größeren Verschachtelungstiefe befinden. Wir durchlaufen mit einer `foreach`-Schleife alle Person-Knoten. Den Wert des Attributs `Id` eines Person-Knotens erhalten wir mit einem verketteten Aufruf der Methode `Attribute()` und der Property `Value`. Der Methode `Attribute()` übergeben wir den Namen des gewünschten Attributs. Die Methode `Attribute()` gibt ein Objekt vom Typ `Attribute` zurück und die Property `Value` liefert den Wert des Attributes. Die Werte der Unterknoten `Vorname` und `Nachname` erhalten wir durch einen verketteten Aufruf der Methoden `Descendants()`, `First()` und der Property `Value`. Die Methode `Descendants()` mit dem Übergabeparameter „`Vorname`“ beziehungsweise „`Nachname`“ liefert uns jeweils eine Liste die einen `Vorname`-Knoten beziehungsweise einen `Nachname`-Knoten enthält. Mit der Methode `First()` holen wir den jeweiligen Knoten aus der Liste und die Property `Value` liefert uns den Wert des Knotens, also den Vornamen beziehungsweise den Nachnamen der Person.

## 14 Dateizugriff mit C#

Am Ende der Schleife geben wir Id, Vorname und Nachname einer Person am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
Id 1; Vorname: Robert; Nachname: Schiefele
Id 2; Vorname: Max; Nachname: Mustermann
Id 3; Vorname: Marta; Nachname: Musterfrau

C:\Users\rober\source\repos\CSharpBuch\XMLVerarbeitung\XMLVerarbeitung\bin\Debug\netcoreapp3.1\XMLVerarbeitung.exe (Prozess "25032") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 14.2.2** Ausgabe der im XML-Format gespeicherten Personen

Die Klasse XElement kann auch zum Erstellen von XML verwendet werden. Die Anweisung:

```
1 var vorname = new XElement("Vorname", "Robert");
```

erstellt ein XML-Knotenobjekt. Der erste Parameter des Konstruktors von XElement ist der Name des Knotens. Der zweite Parameter ist der Inhalt des Knotens. Der zweite Parameter ist vom Typ Object, deshalb können wir für den Inhalt einen Text, wie im obigen Beispiel, übergeben, aber auch ein XML-Attribut oder einen Unterknoten. Die folgende Anweisung erstellt einen Knoten mit einem Attribut:

```
1 var person = new XElement("Person", new XAttribute("Id", 1));
```

Der erstellte Knoten hat den Namen Person und enthält das Attribut Id, welches den Wert 1 hat. Allerdings hat der Knoten keinen Inhalt, es handelt sich um einen sogenannten leeren Knoten. Die Klasse XElement besitzt einen weiteren Konstruktor, bei dem der zweite Parameter vom Typ params object [] ist. Das heißt, wir können mehrere Objekte für den Inhalt übergeben. Die folgenden Anweisungen erstellen einen vollständigen Person-Knoten.

```
1 var vorname = new XElement("Vorname", "Robert");
2 var nachname = new XElement("Nachname", "Schiefele");
3 var person = new XElement("Person", new XAttribute("Id",
4 1), vorname, nachname);
```

Wenn wir die Konstruktoren der Klasse XElement geschickt verschachteln, können wir auch komplexe XML-Knoten mit einer einzigen Anweisung erstellen.

Nach dem Erstellen eines XML-Knotens kann dieser mit der Methode Save () der Klasse XElement als Datei geschrieben werden, wie wir im folgenden Beispielprogramm sehen.

```
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
```

```
5  using System.Xml.Linq;
6
7  namespace XMLVerarbeitung
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             XElement wurzel = new XElement("Personen",
14                 new XElement("Person",
15                     new XAttribute("Id", "1"),
16                     new XElement("Vorname", "Robert"),
17                     new XElement("Nachname", "Schiefele")),
18                 new XElement("Person",
19                     new XAttribute("Id", "2"),
20                     new XElement("Vorname", "Max"),
21                     new XElement("Nachname", "Mustermann")),
22                 new XElement("Person",
23                     new XAttribute("Id", "3"),
24                     new XElement("Vorname", "Marta"),
25                     new XElement("Nachname", "Musterfrau")));
26
27             wurzel.Save("Personen.xml");
28         }
29     }
30 }
```

Wir erzeugen das Wurzelknoten-Objekt für unsere XML-Datei durch verschachtelte Aufrufe von Konstruktoren der Klasse `XElement` und legen es in der Variablen `wurzel` ab. Danach speichern wir die XML-Datei mit Hilfe der Methode `Save()` des Objekts `wurzel`. `Save()` bekommt den Namen der Datei, die wir schreiben wollen, übergeben. Da wir nur den Namen ohne Pfadangabe übergeben, wird die XML-Datei `Personen.xml` im Ausgabeverzeichnis des Projekts gespeichert. Dort können wir auch den Inhalt der Datei überprüfen.

## 14 Dateizugriff mit C#

```

<?xml version="1.0" encoding="utf-8"?>
<Personen>
    <Person Id="1">
        <Vorname>Robert</Vorname>
        <Nachname>Schiefele</Nachname>
    </Person>
    <Person Id="2">
        <Vorname>Max</Vorname>
        <Nachname>Mustermann</Nachname>
    </Person>
    <Person Id="3">
        <Vorname>Marta</Vorname>
        <Nachname>Musterfrau</Nachname>
    </Person>
</Personen>

```

Abb. 14.2.3 Die erstellte Datei Personen.xml

### 14.3 Objekte serialisieren und deserialisieren

In den beiden vorherigen Unterkapiteln haben wir gesehen, wie wir XML-Dateien allgemein verarbeiten können. Dabei haben wir Daten von Personen im XML-Format gelesen und geschrieben. In einem komplizierteren und umfangreicheren Programm zur Verarbeitung von Personendaten hätten wir eine Klasse `Person` angelegt, die die Daten einer Person aufnehmen kann und mehrere Person-Objekte hätten wir in einer Struktur, wie zum Beispiel `List<Person>`, abgelegt. Da wir in einem derartigen Programm des Öfteren Person-Objekte in XML und XML in Person-Objekte konvertieren müssten, hätten wir uns dafür ein paar Konvertierungsmethoden geschrieben. Das Konvertieren von Objekten in Datenformate und zurück wird in der Informatik als Serialisieren und Deserialisieren bezeichnet. Damit wir als Programmierer nicht ständig Methoden für das Serialisieren und Deserialisieren von unseren Objekten schreiben müssen, stellt uns der .NET-Framework eine allgemeine Lösung für dieses Problem zur Verfügung. Um diese Technik näher zu untersuchen, schreiben wir uns eine Klasse `Person`.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace XMLVerarbeitung
6  {
7      public class Person
8      {
9          public int Id { get; set; }
10         public string Vorname { get; set; }

```

## 14.3 Objekte serialisieren und deserialisieren

```

11     public string Nachname { get; set; }

12     public static List<Person> Personen
13     {
14         get
15         {
16             return new List<Person>
17             {
18                 new Person
19                 {
20                     Id =1,
21                     Vorname = "Robert",
22                     Nachname = "Schiefele",
23                 },
24                 new Person
25                 {
26                     Id =2,
27                     Vorname = "Max",
28                     Nachname = "Mustermann",
29                 },
30                 new Person
31                 {
32                     Id =3,
33                     Vorname = "Marta",
34                     Nachname = "Musterfrau",
35                 }
36             };
37         }
38     }
39 }
40 }
41 }
```

Die Klasse Person besitzt die public-Properties Id, Vorname und Nachname, um die Daten einer Person zu speichern. Zusätzlich besitzt sie die statische public-Property Personen, die uns eine Liste von bereits mit Daten vorbelegten Person-Objekten liefert. Mit dem folgenden Beispielprogramm schreiben wir die Person-Objekte, die von der statischen Property Personen geliefert werden, in eine Datei im XML-Format.

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Linq;
6  using System.Xml.Linq;
7  using System.Xml.Serialization;
8
9  namespace XMLVerarbeitung
10 {
11     class Program
12     {
13         static void Main(string[] args)
14         {
15             var seralisierer = new
16             XmlSerializer(typeof(List<Person>));

```

## 14 Dateizugriff mit C#

```

17         using (var fileStream = new FileStream("Personen.
18             xml", FileMode.Create))
19         {
20             serialisierer.Serialize(fileStream, Person.
21             Personen);
22         }
23     }
24 }
25 }
```

Zu den bereits bekannten Klassenbibliotheken `System.IO`, `System.Linq` und `System.Xml.Linq` binden wir zusätzlich noch die Klassenbibliothek `System.Xml.Serialization` ein. Als erstes erstellen wir uns ein Objekt vom Typ `XmlSerializer`. Der Konstruktor von `XmlSerializer` erwartet als Übergabeparameter ein Objekt vom Typ `Type`. Der Typ `Type` beschreibt einen c#-Typ, der entweder in Form einer selbst erstellten Klasse oder durch eine eingebundene Klassenbibliothek zur Verfügung steht. Wir übergeben dem Konstruktor von `XmSerializer` den Ausdruck `typeof(List<Person>)`. Dieser Ausdruck gibt ein Objekt vom Typ `Type` zurück, welches den Typ `List<Person>` beschreibt. Damit haben wir dem erzeugten `XmlSerializer`-Objekt mitgeteilt, dass es Objekte vom Typ `List<Person>` serialisieren beziehungsweise deserialisieren soll. Danach erzeugen wir uns ein Objekt vom Typ `FileStream`, welches unsere Zielfile zum Schreiben repräsentieren soll. Dem Konstruktor übergeben wir den Namen der Datei, die wir schreiben wollen, und den enum-Wert `FileMode.Create`. Damit legen wir fest, dass wir eine neue Datei erstellen und falls die Datei schon existiert, diese überschreiben wollen. Da es sich bei einer Datei um eine nicht verwaltete Ressource handelt, erzeugen wir das `FileStream`-Objekt mit Hilfe des Schlüsselworts `using`. Zum Schluss rufen wir die Methode `Serialize()` des `XmlSerializer`-Objekts auf. Der Methode `Serialize()` übergeben wir das zuvor erzeugte `FileStream`-Objekt und unsere Personenliste. Nach dem Ausführen des Beispielprogramms können wir das Resultat im Ausgabeverzeichnis des Projekts überprüfen.

```

<?xml version="1.0"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Person>
        <Id>1</Id>
        <Vorname>Robert</Vorname>
        <Nachname>Schiefele</Nachname>
    </Person>
    <Person>
        <Id>2</Id>
        <Vorname>Max</Vorname>
        <Nachname>Mustermann</Nachname>
    </Person>
    <Person>
        <Id>3</Id>
        <Vorname>Marta</Vorname>
        <Nachname>Muster-frau</Nachname>
    </Person>
</ArrayOfPerson>
```

Abb. 14.3.1 Die serialisierte Datei Personen.xml

Für den Wurzelknoten vergibt die Methode `Serialize()` automatisch den Name „`ArrayOfPerson`“, welchen sie aus dem Typ `List<Person>` ableitet. Außerdem sehen wir im Wurzelknoten noch die Deklaration der beiden Standard-XML-Namensräume `xsi` und `xsd`, welche wir aber vollständig ignorieren werden, da die Behandlung des Themas XML-Namensräume den Umfang dieses Buches sprengen würde. Auch die Namen der Unterknoten `Person`, `Id`, `Vorname` und `Nachname` werden automatisch aus den Namen der jeweiligen Properties generiert. Für die Programmierpraxis ist dieses Verfahren aber nicht immer optimal. Normalerweise bekommt man als Programmierer Vorgaben für die Namen der Knoten in XML-Dateien, welche aber nicht den üblichen Namenskonventionen in C#-Programmen entsprechen. Außerdem bestimmen die Vorgaben, ob eine Property eines Objekts als XML-Unterknoten oder als XML-Attribute serialisiert werden soll. Für unser Beispiel gehen wir jetzt von folgenden Vorgaben für die XML-Datei aus:

- Die Namen der XML-Knoten sollen `personen`, `person`, `vorname`, und `nachname` heißen.
- Ein Person-Knoten soll die Property `Id` nicht als Unterknoten, sondern als Attribut mit dem Namen `id` enthalten.

Damit wir diese Vorgaben erfüllen können und trotzdem die C#-Konvention, nämlich public-Properties mit Großbuchstaben zu beginnen, einhalten können, ändern wir die Klasse `Person` wie folgt ab:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.Serialization;
4  using System.Text;
5  using System.Xml.Serialization;
6
7  namespace XMLVerarbeitung
8  {
9      [XmlType("person")]
10     public class Person
11     {
12         [XmlAttribute("id")]
13         public int Id { get; set; }
14
15         [XmlElement("vorname")]
16         public string Vorname { get; set; }
17
18         [XmlElement("nachname")]
19         public string Nachname { get; set; }
20
21         public static List<Person> Personen
22         {
23             get
24             {
25                 return new List<Person>
```

## 14 Dateizugriff mit C#

```

26         {
27             new Person
28             {
29                 Id =1,
30                 Vorname = "Robert",
31                 Nachname = "Schiefele",
32             },
33             new Person
34             {
35                 Id =2,
36                 Vorname = "Max",
37                 Nachname = "Mustermann",
38             },
39             new Person
40             {
41                 Id =3,
42                 Vorname = "Marta",
43                 Nachname = "Musterfrau",
44             }
45         );
46     }
47 }
48 }
49 }
```

Zuerst binden wir die Klassenbibliothek `System.Xml.Serialization` ein. Über die Deklaration der Klasse `Person` schreiben wir `[XmlType("person")]`. Bei dieser Syntax handelt es sich um ein sogenanntes Klassenattribut. Klassenattribute ändern die Funktionsweise einer Klasse nicht, aber man kann mit Hilfe von Klassenattributen Zusatzinformationen über eine Klasse angeben. Mit dem Klassenattribut `XmlAttribute` geben wir an, wie die Klasse `Person` nach XML serialisiert werden soll. In unserem Beispiel legen wir fest, dass XML-Knoten, die die Klasse `Person` repräsentieren, den Knotennamen `person` erhalten sollen. Über die `public`-Property `Id` schreiben wir `[XmlAttribute("id")]`. Das ist ein sogenanntes Eigenschaftsattribut, das, analog zu einem Klassenattribut, zusätzliche Informationen zu einer Property einer Klasse angibt. Mit diesem Eigenschaftsattribut legen wir fest, dass die zugehörige Property `Id` nicht in einen XML-Unterknoten, sondern in ein XML-Attribut mit dem Namen `id` serialisiert werden soll. Die `public`-Properties `Vorname` und `Nachname` erhalten die Eigenschaftsattribute `[XmlElement("vorname")]` und `[XmlElement("nachname")]`, damit bestimmen wir, dass die `public`-Properties `Vorname` und `Nachname` als XML-Unterknoten mit den Namen `vorname` und `nachname` serialisiert werden.

Damit wir bei der Serialisierung für den Wurzelknoten den Namen `personen` vergeben können, benötigen wir noch eine kleine Anpassung des Hauptprogramms.

```

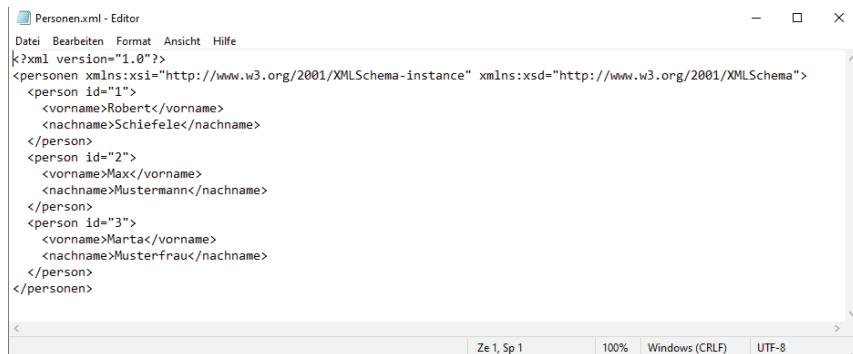
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.IO;
```

## 14.3 Objekte serialisieren und deserialisieren

```

5  using System.Linq;
6  using System.Xml.Linq;
7  using System.Xml.Serialization;
8
9  namespace XMLVerarbeitung
10 {
11     class Program
12     {
13         static void Main(string[] args)
14         {
15             var rootAttribute = new XmlRootAttribute {
16                 ElementName = "personen" };
17             var serialisierer = new XmlSerializer
18             (typeof(List<Person>), rootAttribute);
19             using (var fileStream = new FileStream("Personen.
20                 xml", FileMode.Create))
21             {
22                 serialisierer.Serialize(fileStream, Person.
23                 Personen);
24             }
25         }
26     }
27 }
```

Wir erzeugen zuerst eine neue Instanz der Klasse `XmlRootAttribute` und setzen deren Property `ElementName` auf den Wert „`personen`“, das ist der Name, den der Wurzelknoten erhalten soll. Zum Erzeugen des `XmlSerializer` verwenden wir einen weiteren Konstruktor der Klasse `XmlSerializer`, der als zweiten Parameter das zuvor erzeugte `XmlRootAttribute`-Objekt entgegennimmt. Damit weisen wir den `XmlSerializer` an, für den Wurzelknoten den Namen `personen` zu verwenden. Wenn wir das Programm starten, können wir das Ergebnis wieder im Ausgabeverzeichnis des Projekts überprüfen.



14

Abb. 14.3.2 Die Datei `Personen.xml` mit geänderten Knotennamen

Als nächstes betrachten wir die Deserialisierung von XML-Dateien. Sie ist die Umkehrung der Serialisierung. Das heißt, wir lesen Daten von bekannten Objekten, die wir

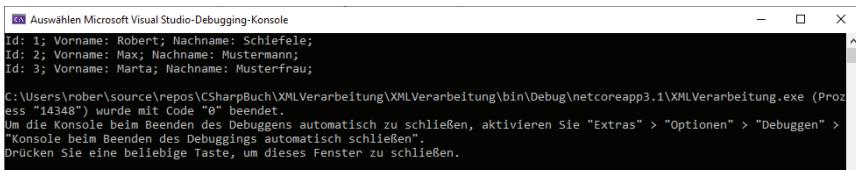
## 14 Dateizugriff mit C#

zuerst in einer XML-Datei gespeichert haben, wieder in Objekte ein. Das folgende Beispielprogramm zeigt das Vorgehen bei der Deserialisierung.

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Linq;
6  using System.Xml.Linq;
7  using System.Xml.Serialization;
8
9  namespace XMLVerarbeitung
10 {
11     class Program
12     {
13         static void Main(string[] args)
14         {
15             List<Person> personen = null;
16             var rootAttribute = new XmlRootAttribute {
17                 ElementName = "personen" };
18             var serialisierer = new XmlSerializer
19                 (typeof(List<Person>), rootAttribute);
20             using (var fileStream = new FileStream("Personen.
21                 xml", FileMode.Open))
22             {
23                 personen = (List<Person>)serialisierer.
24                 Deserialize(fileStream);
25             }
26
27             foreach(var person in personen)
28             {
29                 Console.WriteLine($"Id: {person.Id}; Vorname:
30                     {person.Vorname}; Nachname: {person.Nachname};");
31             }
32         }
33     }
34 }
```

Wir deklarieren die Variable `personen` vom Typ `List<Person>`. Diese Variable soll die deserialisierten Person-Objekte aufnehmen. Dann erzeugen wir ein Objekt vom Typ `XmlRootAttribute` und setzen dessen Property `ElementName` auf den Wert „`personen`“. Jetzt können wir den Serialisierer, ein Objekt vom Typ `XmSerializer` erzeugen. Dem Konstruktor übergeben wir als ersten Parameter den Ausdruck `typeof(List<Person>)`. Damit teilen wir dem Serialisierer mit, dass er eine Struktur vom Typ `List<Person>` verarbeiten soll. Als zweiten Parameter übergeben wir dem Konstruktor das vorher erzeugte `XmlAttribute`-Objekt. Dadurch legen wir fest, dass der Serialisierer einen Wurzelknoten mit dem Namen `personen` verarbeiten soll. Um eine Datei einzulesen, erzeugen wir ein `FileStream`-Objekt für einen `using`-Block. Dem Konstruktor des `FileStream`-Objekts übergeben wir den Namen der zu lesenden XML-Datei. In unserem Fall ist es die XML-Datei, die wir mit unserem Beispielprogramm zum Serialisieren von Objekten erstellt haben.

Da wir den Dateinamen ohne Pfadangabe übergeben, wird die Datei im Ausgabeverzeichnis des Projekts erwartet. Der zweite Parameter, den wir übergeben, bekommt den Enum-Wert  `FileMode.Open`. Damit legen wir fest, dass die Datei bereits existieren muss. Falls die Datei unerwarteterweise doch nicht existieren sollte, würden wir eine `FileNotFoundException` erhalten. Im `using`-Block rufen wir die Methode  `Deserialize()` des `Serialisiers` auf und übergeben ihr unser `FileStream`-Objekt. Die Methode  `Deserialize()` gibt die serialisierten Daten in Form einer Struktur vom Typ `List<Person>` zurück. Allerdings ist der offizielle Rückgabetyp von  `Deserialize()` der Typ `object`. Da alle Typen von `object` entweder direkt oder indirekt abgeleitet sind, kann die Methode  `Deserialize()` somit alle möglichen Typen zurückgeben. Je nachdem, was für eine XML-Struktur wir deserialisieren wollen, damit wir in unserem Beispiel das Ergebnis von  `Deserialize()` an eine Variable vom Typ `List<Person>` zuweisen können, benötigen wir einen Type Cast auf `List<Person>`. Nach dem Deserialisieren durchlaufen wir die Struktur `personen` mit einer `foreach`-Schleife und geben die darin enthaltenen Daten am Bildschirm aus.



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
Id: 1; Vorname: Robert; Nachname: Schiefele;
Id: 2; Vorname: Max; Nachname: Mustermann;
Id: 3; Vorname: Marta; Nachname: Musterfrau;

C:\Users\rober\source\repos\CSharpBuch\xmlVerarbeitung\xmlVerarbeitung\bin\Debug\netcoreapp3.1\xmlVerarbeitung.exe (Proz
ess "14348") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 14.3.3 Bildschirmausgabe der deserialisierten Daten

## 14.4 Verzeichnisse erzeugen und durchsuchen

Im letzten Unterkapitel über den Dateizugriff mit C# beschäftigen wir uns mit Verzeichnissen und wie wir sie erzeugen und durchsuchen können. Die zentrale Klasse, die ein Verzeichnis im Dateisystem des Betriebssystems repräsentiert, ist die Klasse  `DirectoryInfo` aus der Klassenbibliothek `System.IO`. Die Klasse hat nur einen Konstruktor, der den Namen, inklusive Pfad, des gewünschten Verzeichnisses als String entgegennimmt. Beginnt der Verzeichnisname nicht mit einem Laufwerksbuchstaben oder mit der Zeichenfolge „\\“, so wird die Pfadangabe relativ zum Verzeichnis, in dem sich die ausführbare Datei befindet, interpretiert. Wenn der übergebene Verzeichnisname nicht existiert, wirft der Konstruktor keine `Exception`, sondern gibt ein gültiges  `DirectoryInfo`-Objekt zurück. Mit der Boole'schen Property `Exists` des  `DirectoryInfo`-Objekts kann überprüft werden, ob das Verzeichnis schon existiert oder nicht. Mit der Methode `Create()` des Objekts kann das Verzeichnis dann erstellt werden, falls es nicht existiert. Mit der Methode  `Refresh()` kann das  `DirectoryInfo`-Objekt aktualisiert werden, sodass es nach dem Erstellen des Verzeichnisses auf ein existierendes Verzeichnis verweist. Das folgende Beispiel verdeutlicht diese Grundfunktionalitäten der Klasse  `DirectoryInfo`:

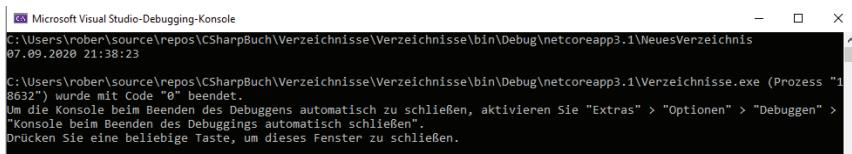
## 14 Dateizugriff mit C#

```

1  using System;
2  using System.IO;
3
4  namespace Verzeichnisse
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              var verzeichnisInfo = new
11                  DirectoryInfo("NeuesVerzeichnis");
12              if (!verzeichnisInfo.Exists)
13              {
14                  verzeichnisInfo.Create();
15                  verzeichnisInfo.Refresh();
16              }
17
18              Console.WriteLine(verzeichnisInfo.FullName);
19              Console.WriteLine(verzeichnisInfo.CreationTime);
20          }
21      }
22  }

```

Zuerst erstellen wir ein neues DirectoryInfo-Objekt, das das Verzeichnis „NeuesVerzeichnis“ relativ zum Ausgabeverzeichnis des Projekts repräsentiert, und speichern es in der Variablen `verzeichnisInfo`. Danach prüfen wir mit Hilfe der Property `Exists` des Objekts `verzeichnisInfo`, ob das Verzeichnis bereits existiert. Wenn es noch nicht existiert, legen wir es mit der Methode `Create()` an und aktualisieren das Objekt `verzeichnisInfo` mit der Methode `Refresh()`. Zum Schluss verwenden wir die Properties `FullName` und `CreationTime`, um den vollständigen Pfad des Verzeichnisses und sein Erstellungsdatum am Bildschirm auszugeben.



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output text is:

```

Microsoft Visual Studio-Debugging-Konsole
C:\Users\rober\source\repos\CSharpBuch\Verzeichnisse\Verzeichnisse\bin\Debug\netcoreapp3.1\NeuesVerzeichnis
07.09.2020 21:38:23
C:\Users\rober\source\repos\CSharpBuch\Verzeichnisse\Verzeichnisse\bin\Debug\netcoreapp3.1\Verzeichnisse.exe (Prozess "1
8632") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 14.4.1** Pfad und Erstellungsdatum des Verzeichnisses

Als nächstes betrachten wir die Methode `GetDirectories()` der Klasse `DirectoryInfo`. Diese Methode liefert uns ein Array zurück, das für alle Unterverzeichnisse des betreffenden `DirectoryInfo`-Objekts ein Element vom Typ `DirectoryInfo` enthält. Das folgende Beispiel zeigt die Verwendung dieser Methode:

```

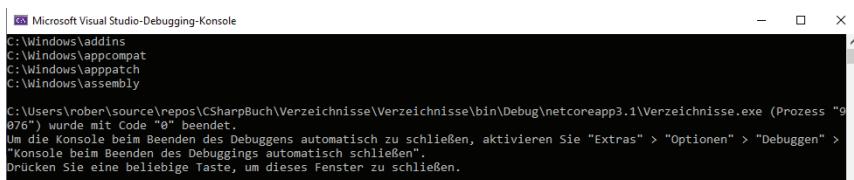
1  using System;
2  using System.IO;
3  using System.Linq;
4

```

```

5  namespace Verzeichnisse
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             var verzeichnisInfo = new DirectoryInfo("C:\\\\
12             Windows");
13             var verzeichnisse = verzeichnisInfo.GetDirectories();
14
15             foreach (var verzeichnis in verzeichnisse.Where(v =>
16                 v.Name.StartsWith("a")))
17             {
18                 Console.WriteLine(verzeichnis.FullName);
19             }
20         }
21     }
22 }
```

Zum Beginn des Programms erstellen wir uns ein neues DirectoryInfo-Objekt. Dem Konstruktor übergeben wir die absolute Pfadangabe „C:\\Windows“. Damit repräsentiert das DirectoryInfo-Objekt das Verzeichnis „Windows“ im Laufwerk „C:“. Mit der Methode GetDirectories() lesen wir alle Unterverzeichnisse des Verzeichnisses C:\\Windows. Da die Methode GetDirectories() ein Array zurück gibt und jedes Array das Interface IEnumerable<T> implementiert, können wir mit Linq unser DirectoryInfo-Array filtern. Durch den Aufruf der Linq-Methode Where(v => v.Name.StartsWith(„a“)) erhalten wir nur Unterverzeichnisse, deren Name mit dem Buchstaben a beginnen. Im Lambda-Ausdruck v => v.Name.StartsWith(„a“) verwenden wir die Property Name der Klasse DirectoryInfo, die den Namen ohne vollständige Pfadangabe des Verzeichnisses liefert. Das gefilterte DirectoryInfo-Array durchlaufen wir mit einer foreach-Schleife und geben den vollständigen Pfad der jeweiligen Verzeichnisse am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays a list of directory paths starting with 'a':

- C:\\Windows\\adlins
- C:\\Windows\\appcompat
- C:\\Windows\\apppatch
- C:\\Windows\\assembly

Below the list, there is some additional text from the console output:

C:\\Users\\rober\\source\\repos\\CSharpBuch\\Verzeichnisse\\Verzeichnisse\\bin\\Debug\\netcoreapp3.1\\Verzeichnisse.exe (Prozess "9076") wurde mit Code "0" beendet.  
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".  
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

**Abb. 14.4.2** Alle Unterverzeichnisse, die mit a beginnen

Eine weitere wichtige Methode beim Arbeiten mit Verzeichnissen ist die Methode `GetFiles()`. Sie liefert die Dateien, die ein Verzeichnis in Form eines Arrays mit Elementen vom Typ `FileInfo` enthält. Das folgende Beispiel zeigt die Verwendung dieser Methode:

```

1  using System;
2  using System.IO;
```

## 14 Dateizugriff mit C#

```

3  using System.Linq;
4
5  namespace Verzeichnisse
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             var verzeichnisInfo = new DirectoryInfo("C:\\\\
12             Windows");
13             var dateien = verzeichnisInfo.GetFiles();
14
15             foreach (var datei in dateien.Where(v => v.Extension
16             == ".exe"))
17             {
18                 Console.WriteLine(datei.FullName);
19             }
20         }
21     }
22 }
```

Wir erzeugen ein Objekt vom Typ `DirectoryInfo` für das Windows-Verzeichnis mit der absoluten Pfadangabe „`C:\\Windows`“. Dann rufen wir die Methode `GetFiles()` des erzeugten `DirectoryInfo`-Objekts und erhalten ein Array, das ein Element vom Typ `FileInfo` für jede Datei im Windows-Verzeichnis enthält. Danach filtern wir mit Hilfe der Linq-Methode `Where()` nur die `FileInfo`-Objekte für ausführbare Dateien aus dem Array `dateien`. Zum Filtern verwenden wir den Lambda-Ausdruck `v => v.Extension == ".exe"`. Dabei hilft uns die Property `Extension` der Klasse `FileInfo`, die die Endung des Dateinamens enthält. Für ausführbare Dateien heißt die Endung „`.exe`“. In der Schleife geben wir Pfad und Namen der jeweiligen Datei am Bildschirm aus.



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays a list of executable files located in the `C:\Windows` directory. The output is as follows:

```

Microsoft Visual Studio-Debugging-Konsole
C:\Windows\bfsvc.exe
C:\Windows\explorer.exe
C:\Windows\HelpPane.exe
C:\Windows\hh.exe
C:\Windows\notepad.exe
C:\Windows\regedit.exe
C:\Windows\RtcAmU64.exe
C:\Windows\RtcRvU64.exe
C:\Windows\sptLwW64.exe
C:\Windows\winhlp32.exe
C:\Windows\write.exe

C:\Users\rober\source\repos\CSharpBuch\Verzeichnisse\Verzeichnisse\bin\Debug\netcoreapp3.1\Verzeichnisse.exe (Prozess "1
1349") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 14.4.3** Die Liste der ausführbaren Dateien im Windows-Verzeichnis

## 14.5 Übungsaufgaben: Arbeiten mit Dateien

In dieser Übung schreiben wir schrittweise eine Konsolen-App, die die Struktur eines übergebenen Verzeichnisses in eine Objektstruktur einliest und die diese Objekt-

struktur mit Einrückungen am Bildschirm ausgibt beziehungsweise im XML-Format in eine Datei schreibt. Da es sich hier nur um ein Übungsprogramm handelt, werden wir bei den Musterlösungen auf die Fehlerbehandlung verzichten.

**Teilaufgabe 1:**

Erstellen Sie die Daten-Klassen `Datei` und `Verzeichnis`. Die Klasse `Datei` hat die public-Properties `Name` und `ErstellungsDatum` vom Typ `string` beziehungsweise vom Typ `DateTime`. Die Klasse `Verzeichnis` wird von der Klasse `Datei` abgeleitet und hat zusätzlich die public-Properties `Dateien` und `Verzeichnisse` vom Typ `List<Datei>` beziehungsweise `List<Verzeichnis>`. Der Konstruktor der Klasse `Verzeichnis` initialisiert die Properties `Dateien` und `Verzeichnisse`, jeweils mit einer leeren Liste.

**Teilaufgabe 2:**

Erweitern Sie die Klasse `Verzeichnis` um die statische Methode `Lade()`. Die Methode `Lade()` erwartet im Übergabeparameter `verzeichnisName` vom Typ `string` den Namen und Pfad eines Verzeichnisses und liest dann rekursiv die vollständige Verzeichnisstruktur des übergebenen Verzeichnisnamens ein und gibt ein Objekt vom Typ `Verzeichnis` zurück.

**Teilaufgabe 3:**

Erweitern Sie die Klasse `Verzeichnis` um die Methode `Ausgabe()`. Die Methode `Ausgabe()` gibt rekursiv den Namen und das Erstellungsdatum aller Unterverzeichnisse und Dateien des Verzeichnis-Objekts aus. Verdeutlichen Sie die hierarchische Struktur durch Einrückungen. Unterscheiden Sie Dateien und Verzeichnisse bei der Ausgabe, indem Sie Verzeichnissen den Text „<Verzeichnis>“ voranstellen.

**Teilaufgabe 4:**

Erweitern Sie die Klasse `Verzeichnis` um die Methode `AusgabeInDatei()`. Diese Methode erwartet als Übergabeparameter den Pfad und Namen einer Datei - entweder mit absoluter oder relativer Pfadangabe. Die Methode erstellt die auszugebende Datei und schreibt das Verzeichnis-Objekt im XML-Format in die Datei. Sollte die auszugebende Datei bereits existieren, wird sie ohne Rückfrage überschrieben. Die ausgegebene XML-Datei soll die folgende Struktur haben:

Ein Verzeichnis wird als XML-Knoten mit dem Namen `verzeichnis` und eine Datei als XML-Knoten mit dem Namen `datei` serialisiert. Die Properties `Name` und `ErstellungsDatum` werden als XML-Attribute mit den Namen `name` und `erstellungsdatum` serialisiert. Die Listen `Dateien` und `Verzeichnisse` werden als XML-Knoten mit den Namen `dateien` und `verzeichnisse` serialisiert.

**Teilaufgabe 5:**

Erstellen Sie das Hauptprogramm. Das Hauptprogramm nimmt zwei Anwendungsargumente entgegen. Das erste Argument ist die Pfadangabe für das Verzeichnis, das eingelesen werden soll. Das zweite Argument ist optional und enthält, wenn angegeben, einen Dateinamen für eine XML-Ausgabedatei. Das Hauptprogramm liest die Struktur des im ersten Argument angegebenen Verzeichnisses ein und gibt sie am Bildschirm aus. Wenn das zweite Argument angegeben ist, schreibt das Hauptprogramm die Verzeichnisstruktur in die angegebene Ausgabedatei.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace DateienÜbung
6  {
7      public class Datei
8      {
9          public string Name { get; set; }
10         public DateTime ErstellungsDatum { get; set; }
11     }
12 }
13
14 using System;
15 using System.Collections.Generic;
16 using System.Text;
17
18 namespace DateienÜbung
19 {
20     public class Verzeichnis : Datei
21     {
22         public Verzeichnis()
23         {
24             Dateien = new List<Datei>();
25             Verzeichnisse = new List<Verzeichnis>();
26         }
27
28         public List<Datei> Dateien { get; set; }
29
30         public List<Verzeichnis> Verzeichnisse { get; set; }
31     }
32 }
```

**Musterlösung für Teilaufgabe 2:**

```
1 public static Verzeichnis Lade(string verzeichnisName)
2 {
3     var verzeichnisInfo = new DirectoryInfo(verzeichnisName);
4     var verzeichnis = new Verzeichnis();
5
6     if(verzeichnisInfo.Exists)
7     {
8         verzeichnis = LeseVerzeichnisRekursiv(verzeichnisInfo);
9     }
10
11     return verzeichnis;
12 }
13
14 private static Verzeichnis LeseVerzeichnisRekursiv(DirectoryInfo
15 verzeichnisInfo)
16 {
17     var verzeichnis = new Verzeichnis();
18     verzeichnis.Name = verzeichnisInfo.Name;
19     verzeichnis.ErstellungsDatum = verzeichnisInfo.CreationTime;
20
21     foreach (var dateiInfo in verzeichnisInfo.GetFiles())
22     {
23         var datei = new Datei();
24         datei.Name = dateiInfo.Name;
25         datei.ErstellungsDatum = dateiInfo.CreationTime;
26
27         verzeichnis.dateien.Add(datei);
28     }
29
30     foreach (var unterverzeichnisInfo in verzeichnisInfo.
31 GetDirectories())
32     {
33         var unterVerzeichnis = LeseVerzeichnisRekursiv
34         (unterverzeichnisInfo);
35         verzeichnis.verzeichnisse.Add(unterVerzeichnis);
36     }
37
38     return verzeichnis;
39 }
```

### Musterlösung für Teilaufgabe 3

```
1 public void Ausgabe()
2 {
3     AusgabeRekursiv(this, "");
4 }
5
6 private void AusgabeRekursiv(Verzeichnis verzeichnis, string
7 einrueckung)
8 {
9     Console.WriteLine(${einrueckung}<Verzeichnis> {verzeichnis.
10 Name} ({verzeichnis.ErstellungsDatum})");
11
12     einrueckung += "    ";
13
14     foreach (var datei in verzeichnis.Dateien)
15     {
16         Console.WriteLine(${einrueckung}{datei.Name} ({datei.
17 ErstellungsDatum})");
18     }
19
20     foreach (var unterverzeichnis in verzeichnis.Verzeichnisse)
21     {
22         AusgabeRekursiv(unterverzeichnis, einrueckung);
23     }
24 }
```

## 14 Dateizugriff mit C#

**Musterlösung Aufgabe 4:**

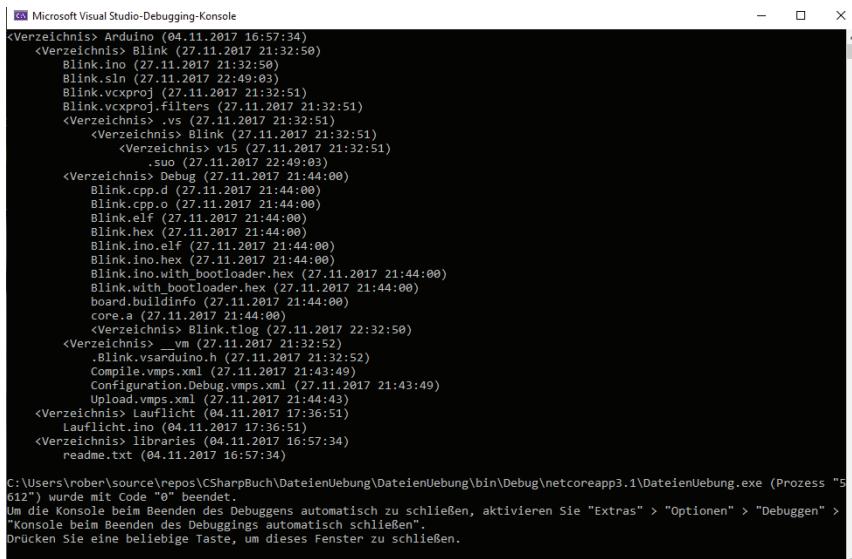
```
1 public void AusgabeInDatei(string dateiName)
2 {
3     var Serialisierer = new XmlSerializer(typeof(Verzeichnis));
4     using (var fileStream = new FileStream(dateiName, FileMode.
5         Create))
6     {
7         Serialisierer.Serialize(fileStream, this);
8     }
9 }
10
11 using System;
12 using System.Collections.Generic;
13 using System.Text;
14 using System.Xml.Serialization;
15
16 namespace DateienUebung
17 {
18     [XmlType("datei")]
19     public class Datei
20     {
21         [XmlAttribute("name")]
22         public string Name { get; set; }
23         [XmlAttribute("erstellungsdatum")]
24         public DateTime ErstellungsDatum { get; set; }
25     }
26 }
27
28 using System;
29 using System.Collections.Generic;
30 using System.IO;
31 using System.Runtime.Serialization;
32 using System.Text;
33 using System.Xml.Serialization;
34
35 namespace DateienUebung
36 {
37     [XmlType("verzeichnis")]
38     public class Verzeichnis : Datei
39     {
40
41         public Verzeichnis()
42         {
43             Dateien = new List<Datei>();
44             Verzeichnisse = new List<Verzeichnis>();
45         }
46
47         [XmlElement("dateien")]
48         public List<Datei> Dateien { get; set; }
49
50         [XmlElement("verzeichnisse")]
51         public List<Verzeichnis> Verzeichnisse { get; set; }
52     }
53 }
```

**Musterlösung für Teilaufgabe 5:**

```

1  using System;
2
3  namespace DateienUebung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              if(args.Length < 1 || args.Length > 2)
10              {
11                  return;
12              }
13
14              var verzeichnis = Verzeichnis.Lade(args[0]);
15              verzeichnis.Ausgabe();
16
17              if (args.Length == 2)
18              {
19                  verzeichnis.AusgabeInDatei(args[1]);
20              }
21          }
22      }
23  }

```



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output displays a directory listing for the folder 'Blink' under 'Verzeichnis'. The listing includes files like 'Blink.ino', 'Blink.sln', 'Blink.vcxproj', 'Blink.vcxproj.filters', 'Blink.vs', 'Blink.vl5', 'Blink.v15', 'Blink.cpp.d', 'Blink.cpp.o', 'Blink.elf', 'Blink.hex', 'Blink.ino.elf', 'Blink.ino.hex', 'Blink.ino.with bootloader.hex', 'Blink.with bootloader.hex', 'board.buildinfo', 'core.a', 'Blink.tlog', 'Blink.vm', 'Compile.vmps.xml', 'Configuration.Debug.vmps.xml', 'Upload vmps.xml', 'Lauflight.ino', and 'librarian'. It also shows 'readme.txt'. The console window has a dark theme.

**Abb. 14.5.1** Bildschirmausgabe eines Verzeichnisses

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 15

# Datenbankzugriff mit dem Microsoft SQL-Server

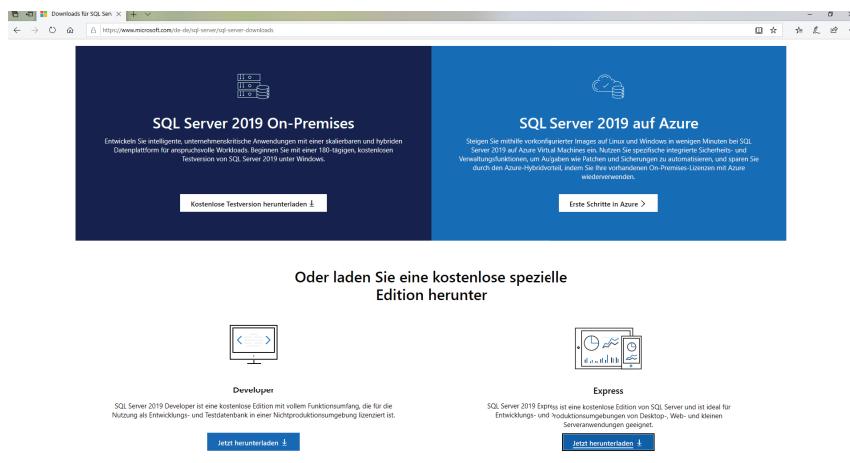
Nach dem Schreiben und Lesen von Daten in Dateien werden wir jetzt das Schreiben und Lesen von Daten auf einen Datenbankserver behandeln. Datenbankserver sind aus der modernen Applikationsentwicklung nicht mehr wegzudenken. Ein Datenbankserver ist ein Softwareprodukt, das normalerweise auf einem eigenen Computer läuft. Programme, die den Datenbankserver verwenden, werden Datenbank-Clients genannt und können auf verschiedenen anderen Computern laufen. Damit diese Programme auf den Datenbankserver zugreifen können, müssen die Computer, auf denen Datenbank-Clients laufen, mit dem Computer, auf dem der Datenbankserver läuft, über das gleiche Netzwerk verbunden sein. Ein Datenbank-Client kann auch auf den Datenbankserver zugreifen, wenn er auf dem gleichen Computer wie der Datenbankserver läuft. Diese Eigenschaft macht man sich in der Software-Entwicklung häufig zunutze und installiert einen Datenbankserver auf dem Computer eines Softwareentwicklers. Für dieses Buch verwenden wir den Microsoft SQL-Server. Wenn Sie die Beispiele in diesem Kapitel nachvollziehen wollen, müssen Sie auf Ihrem Computer den Microsoft SQL-Server installieren. Im nächsten Unterkapitel beschäftigen wir uns mit der Installation des SQL-Servers. Falls Sie den Microsoft SQL-Server bereits auf Ihrem Computer installiert haben, können Sie das folgende Unterkapitel überspringen.

### 15.1 Microsoft SQL-Server installieren

Die Installation für den Microsoft SQL-Server express finden Sie unter folgender URL:

<https://www.microsoft.com/de-de/sql-server/sql-server-downloads>

## 15 Datenbankzugriff mit dem Microsoft SQL-Server



**Abb. 15.1.1** Download Seite für Microsoft SQL Server

Klicken Sie bei der Express Version auf „Jetzt herunterladen“ und dann auf „Ausführen“. Die Sicherheitsabfrage von Windows, ob Sie das Programm auch wirklich ausführen wollen, bestätigen Sie mit „Ja“.



**Abb. 15.1.2** Auswahl des Installationstyps

Klicken Sie auf „Benutzerdefiniert“.

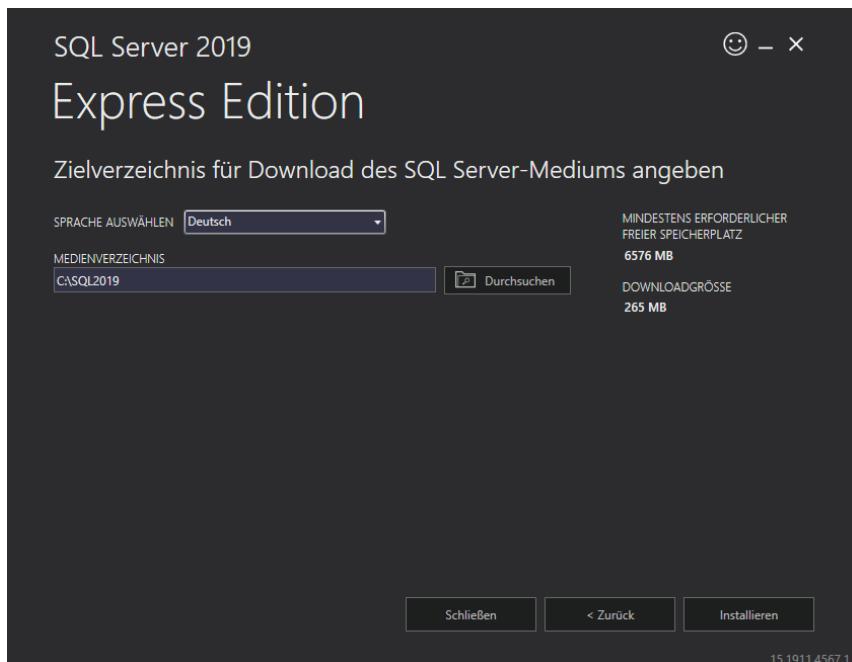


Abb. 15.1.3 Zielverzeichnis für Download des SQL Server-Mediums

Im Feld „SPRACHE AUSWÄHLEN“ wählen Sie „Deutsch“ und im Feld „MEDIENVERZEICHNIS“ können Sie Voreinstellung „C:\SQL2019“ lassen. Klicken Sie auf die Schaltfläche „Installieren“.

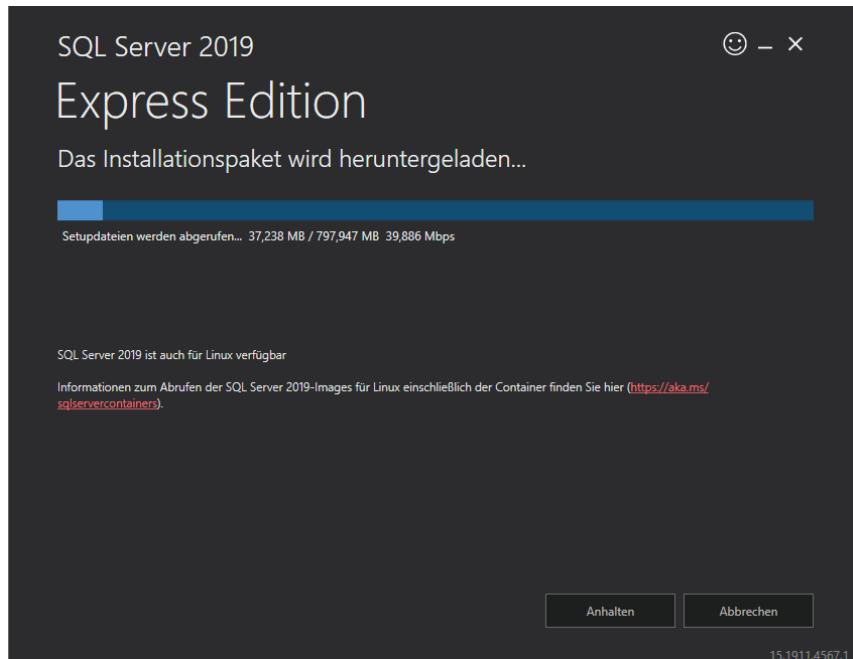


Abb. 15.1.4 Download des Installationspaketes

Nach dem Herunterladen des Installationspaketes erscheint das „SQL Server-Installationscenter“.

## 15.1 Microsoft SQL-Server installieren

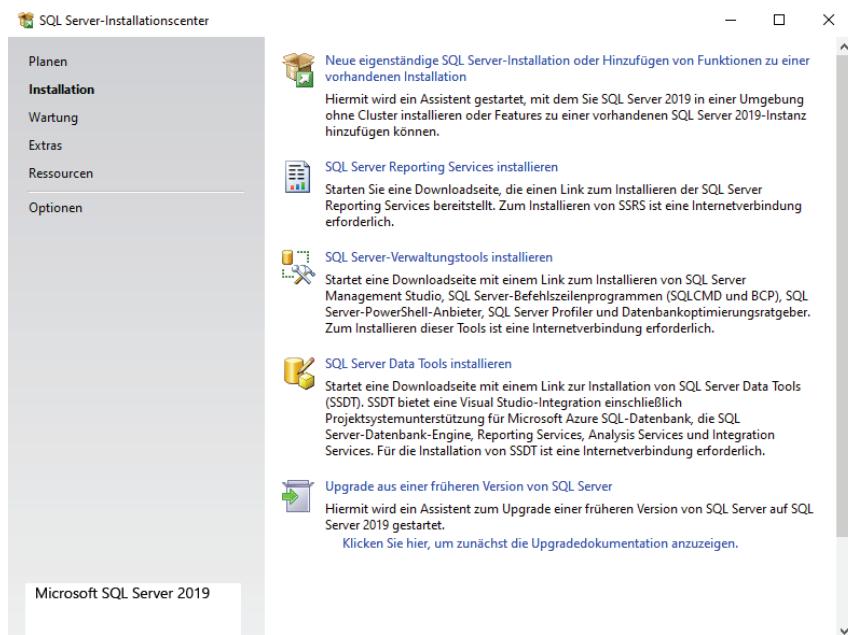


Abb. 15.1.5 SQL Server-Installationscenter

Klicken Sie auf „Neue eigenständige SQL Server-Installation oder Hinzufügen von Funktionen zu einer vorhanden Installation“. Der Dialog „Installationsregeln erscheint“:

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

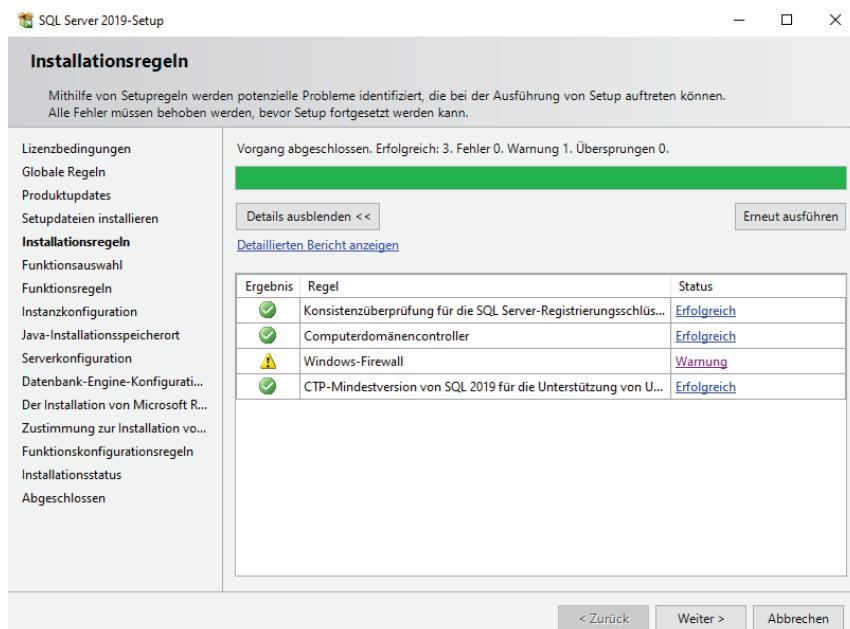


Abb. 15.1.6 Überprüfung der Installationsregeln

Hier überprüft das SQL Server-Installationscenter, ob alle technischen Voraussetzungen gegeben sind, um den SQL Server auf diesem Computer zu installieren. Im Normalfall sollten keine Probleme auftreten. Die Warnung, die Sie auf dem Bild sehen, erscheint deswegen, weil auf dem Computer des Autors die Windows-Firewall aktiviert ist. Da die Windows-Firewall den Zugriff von anderen Computern auf den zu installierenden SQL Server blockieren könnte, gibt das SQL Server-Installationscenter eine Warnung aus. Da wir aber mit unseren Beispielprogrammen nur von dem Computer, auf dem der SQL Server installiert ist, auf ihn zugreifen werden, können wir diese Warnung ignorieren.

Klicken Sie auf die Schaltfläche „Weiter“. Der Dialog Funktionsauswahl erscheint.

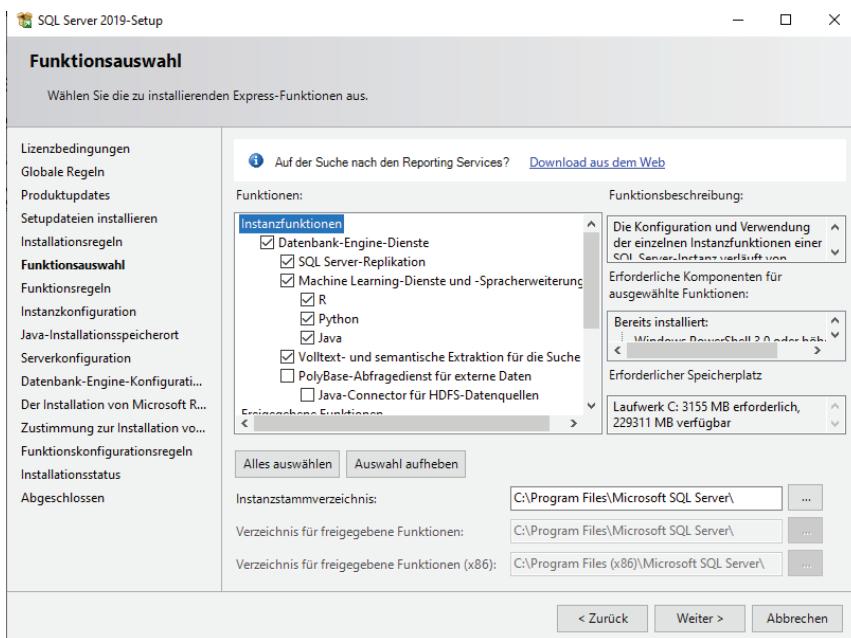


Abb. 15.1.7 Funktionsauswahl

Hier behalten Sie bitte die Voreinstellungen des SQL Server-Installationscenters. Diese Funktionsauswahl genügt für unsere Zwecke. Klicken Sie auf die Schaltfläche „Weiter“. Das Fenster „Instanzkonfiguration“ erscheint.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

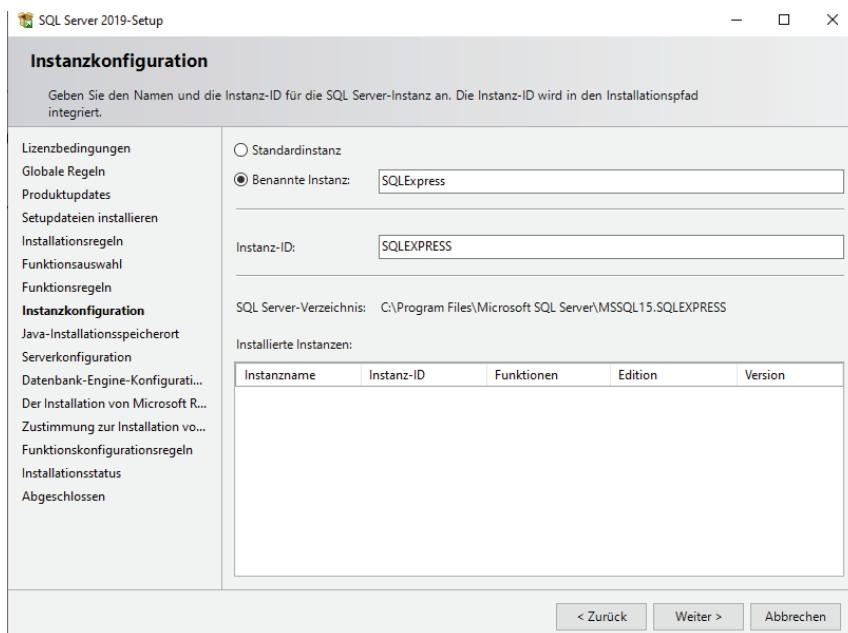


Abb. 15.1.8 Konfiguration der Instanz

Hier wählen Sie „Benannte Instanz“ aus und geben im Feld rechts daneben „SQLEXPRESS“ an. Im Feld „Instanz-ID“ geben Sie „SQLEXPRESS“ an. Das sollten auch die Voreinstellungen des SQL Server-Installationscenters sein. Klicken Sie auf „Weiter“. Das Fenster „Java-Installationsspeicherort“ erscheint.

## 15.1 Microsoft SQL-Server installieren

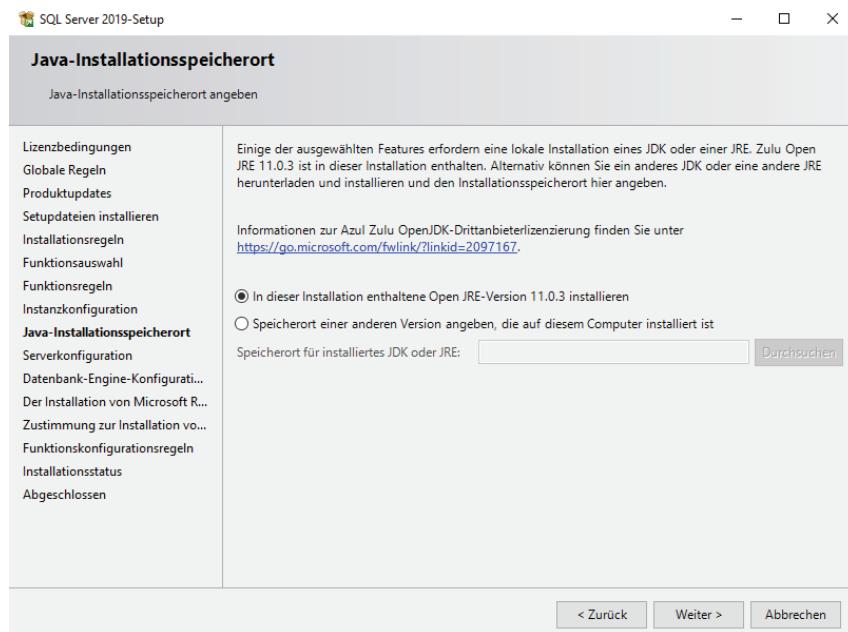


Abb. 15.1.9 Java-Installationsspeicherort

Wenn Sie kein eigenes JDK verwenden wollen, belassen Sie die Voreinstellung „In dieser Installation enthaltene Open JRE-Version 11.0.3 installieren“, dann klicken Sie auf die Schaltfläche „Weiter“. Als nächstes folgt der Dialog „Serverkonfiguration“.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

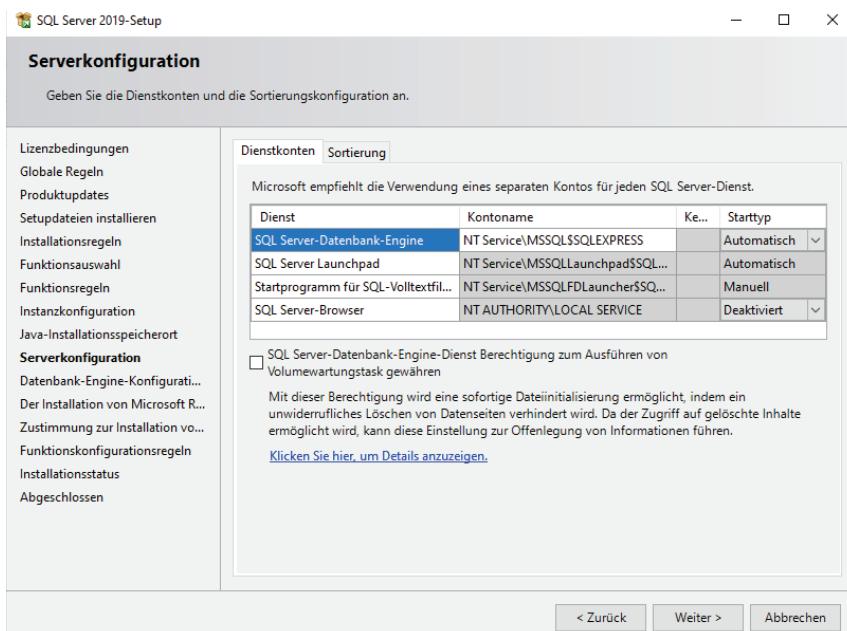
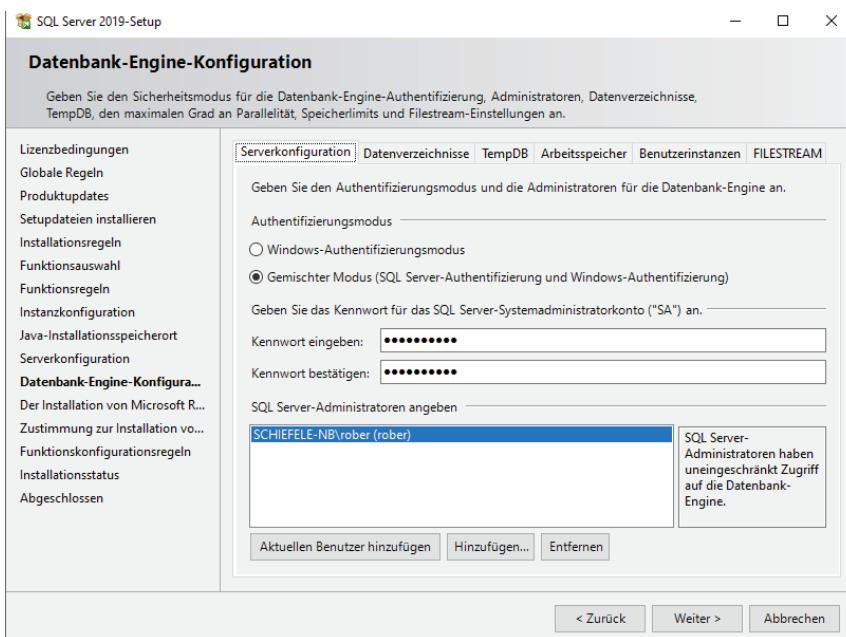


Abb. 15.1.10 Serverkonfiguration

Hier können Dienstkonten für die verschiedenen Dienste des SQL Servers konfiguriert werden. Für unsere Zwecke genügen hier wieder die Voreinstellungen. Klicken Sie auf die Schaltfläche „Weiter“, um das Fenster „Datenbank-Engine-Konfiguration“ anzuzeigen.



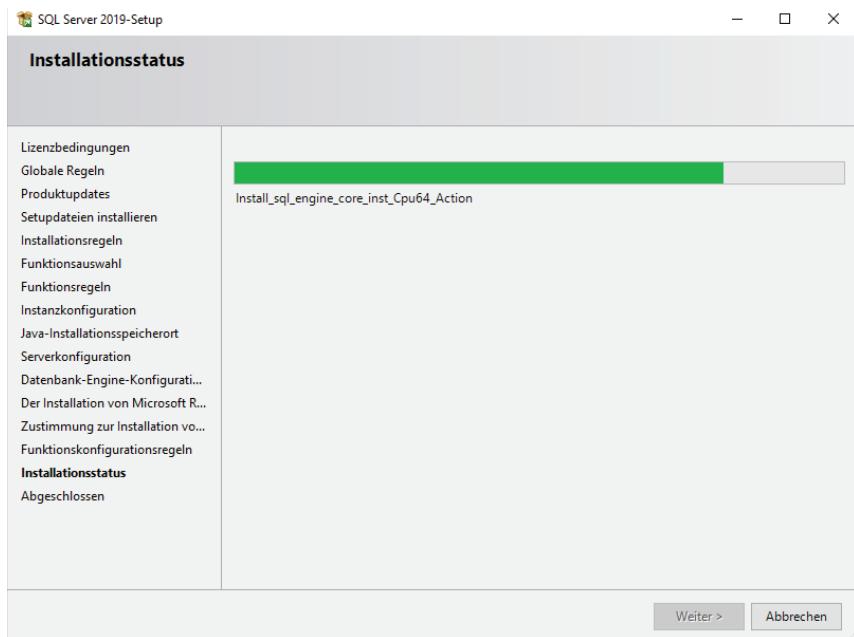
**Abb. 15.1.11** Datenbank-Engine-Konfiguration

Als Authentifizierungsmodus wählen Sie „Gemischter Modus (SQL Server-Authentifizierung und Windows-Authentifizierung)“ aus und vergeben ein Passwort für das SQL Server-Systemadministratorkonto. Dadurch teilen Sie dem SQL-Server mit, dass Sie sowohl SQL-Server-eigene Konten als auch Konten des Betriebssystems für die Authentifizierung am SQL Server verwenden wollen. Stellen Sie sicher, dass im Feld „SQL Server-Administratoren angeben“ ihr aktuelles Benutzerkonto angegeben ist. Wenn nicht, fügen Sie es hinzu, durch einen Klick auf die Schaltfläche „Aktuellen Benutzer hinzufügen“. Klicken Sie auf die Schaltfläche „Weiter“.

Das Fenster „Der Installation von Microsoft R Open zustimmen“ erscheint. Klicken Sie zuerst auf die Schaltfläche „Zustimmen“ und dann auf „Weiter“.

Der nächste Dialog „Zustimmung zur Installation von Python“ erscheint. Klicken Sie hier ebenfalls auf „Zustimmen“ und dann auf „Weiter“.

Jetzt öffnet sich das Fenster „Installationsstatus“ und die Installation beginnt.

**15 Datenbankzugriff mit dem Microsoft SQL-Server****Abb. 15.1.12** Installationsstatus

Je nach der verfügbaren Bandbreite Ihres Internetzugangs kann dies von wenigen Minuten bis zu ein bis zwei Stunden dauern.

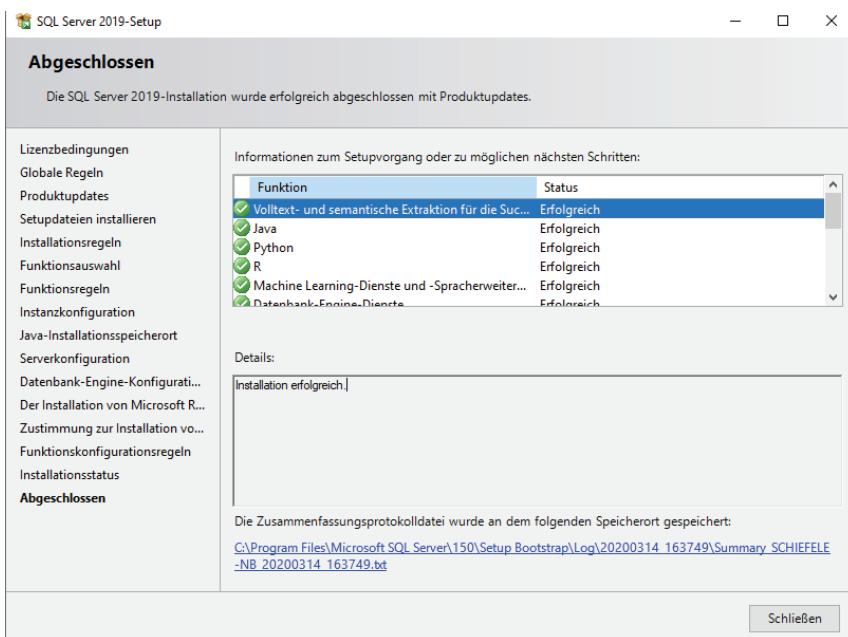


Abb. 15.1.13 Abschluss der Installation des Microsoft SQL Servers

Zum Abschluss der Installation des SQL Servers sehen Sie den Dialog „Abgeschlossen“, der Ihnen anzeigen, dass die Installation korrekt verlaufen ist. Klicken Sie auf die Schaltfläche „Schließen“.

Im SQL Server-Installationscenter klicken Sie auf „SQL Server-Verwaltungstools installieren“, um das SQL Server Management Studio zu installieren. Das benötigen wir zum Anlegen und Verwalten von Datenbanken.

Die Webseite „Herunterladen von SQL Server Management Studio (SSMS)“ öffnet sich. Klicken Sie auf den blauen Link: „Herunterladen von SQL Server Management Studio (SSMS)“ und danach auf „Ausführen“ und bestätigen Sie die Sicherheitsabfrage von Windows mit „Ja“.

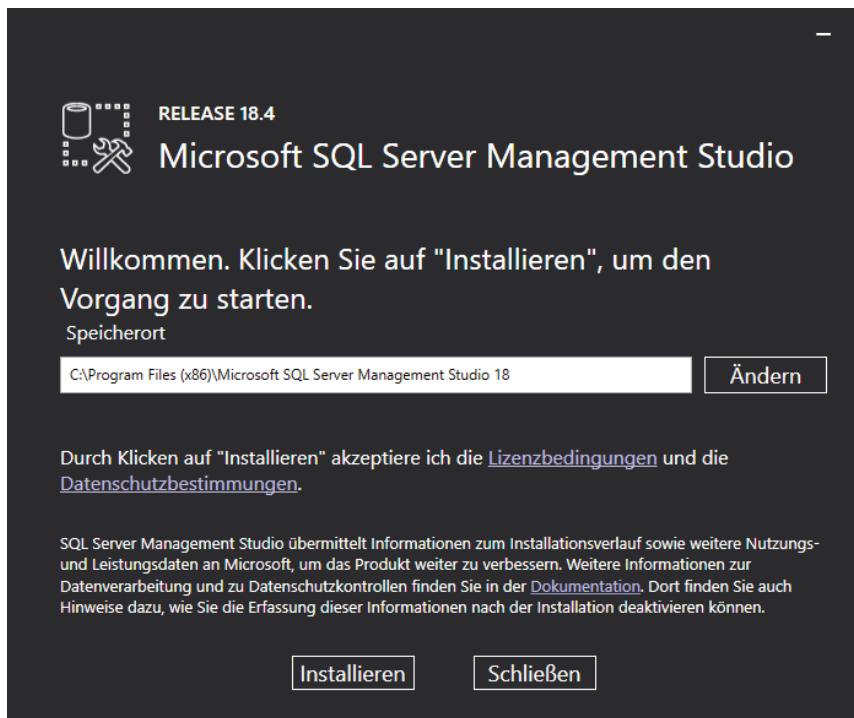


Abb. 15.1.14 Installationsprogramm des Microsoft SQL Server Management Studios

Klicken Sie auf die Schaltfläche „Installieren“.

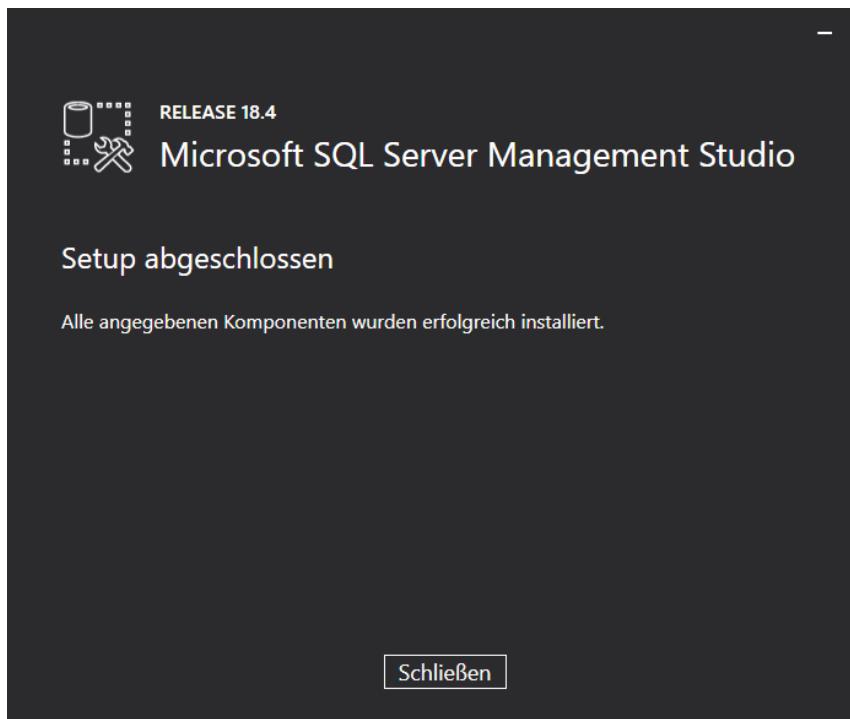


Abb. 15.1.15 Abschluss der Microsoft SQL Server Management Studio Installation

Nach dem Ende der Installation klicken Sie auf die Schaltfläche schließen. Auch das SQL Server-Installationscenter können Sie jetzt schließen.

Das „SQL Server Management Studio“ finden Sie, wenn Sie in der Windows-Suche „sql“ tippen.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

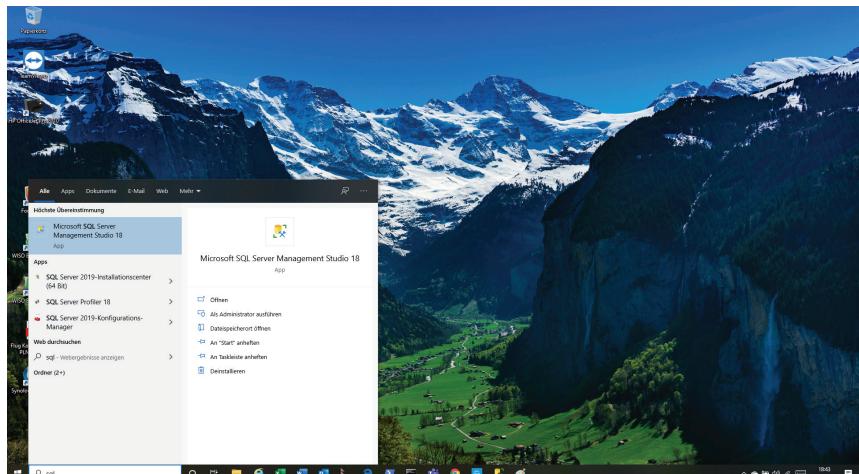


Abb. 15.1.16 SQL Server Management Studio starten

Zur Überprüfung der Installation starten Sie jetzt das „SQL Server Management Studio“.

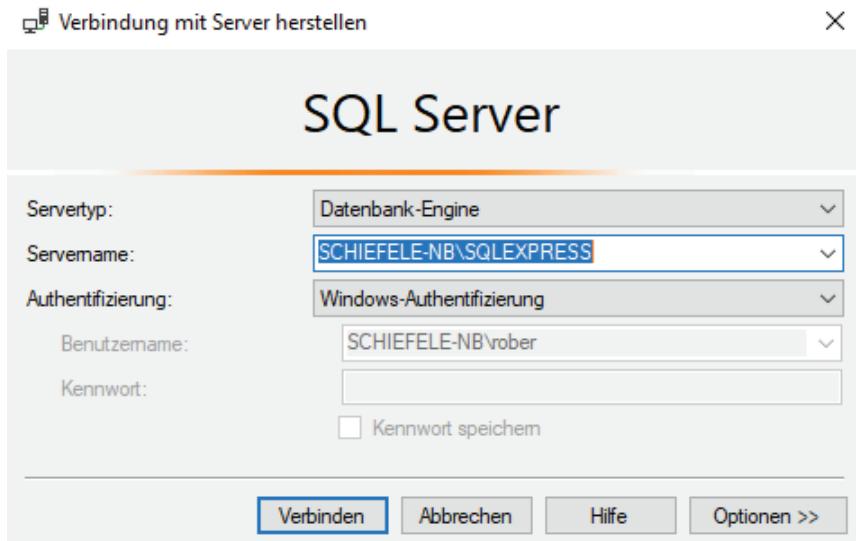


Abb. 15.1.17 Der Anmeldedialog des Microsoft SQL Servers

Im Anmeldedialog des SQL Servers sollte das Feld „Servertyp“ mit „Datenbank-Engine“ vorbelegt sein. Das Feld Servername enthält den Namen Ihres Computers gefolgt von einem „\“-Zeichen und der Instanz-ID des SQL-Servers - in unserem Fall „SQLEXPRESS“. Klicken Sie auf „Verbinden“ und wenn Sie jetzt alles richtig gemacht haben, zeigt das SQL Management Studio Ihren SQL-Server an.

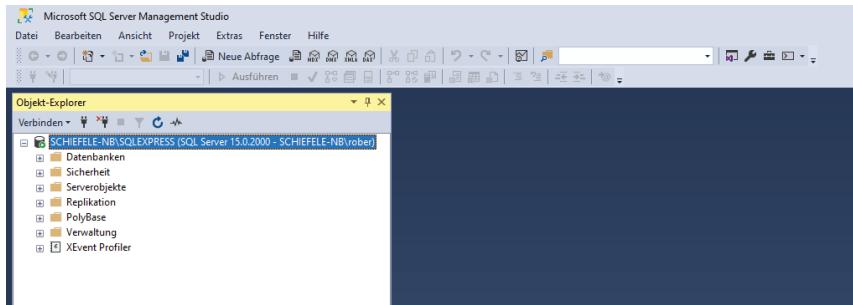


Abb. 15.1.18 Das Microsoft SQL Server Management Studio nach dem Anmelden

## 15.2 Eine Datenbank erstellen

Um einen Datenbankserver als Datenablage für ein Programm zu verwenden, benötigen wir zunächst eine Datenbank. Um eine Datenbank zu erstellen, öffnen wir das Microsoft SQL Management Studio und klicken mit der rechten Maustaste auf den Ordner Datenbanken. Aus dem Kontextmenü wählen wir „Neue Datenbank ...“ aus.

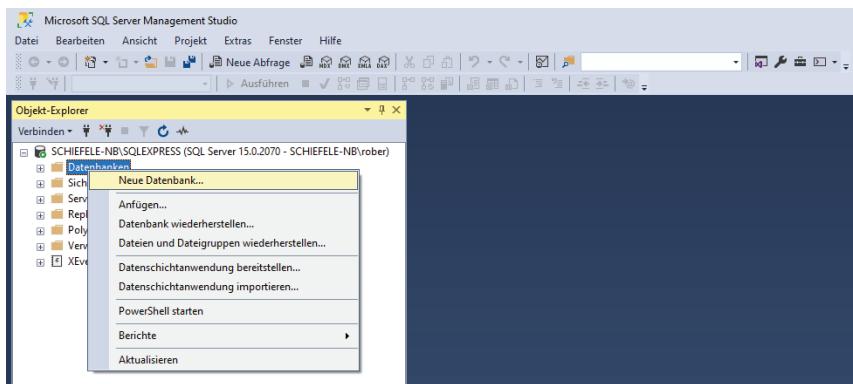


Abb. 15.2.1 Eine neue Datenbank anlegen

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

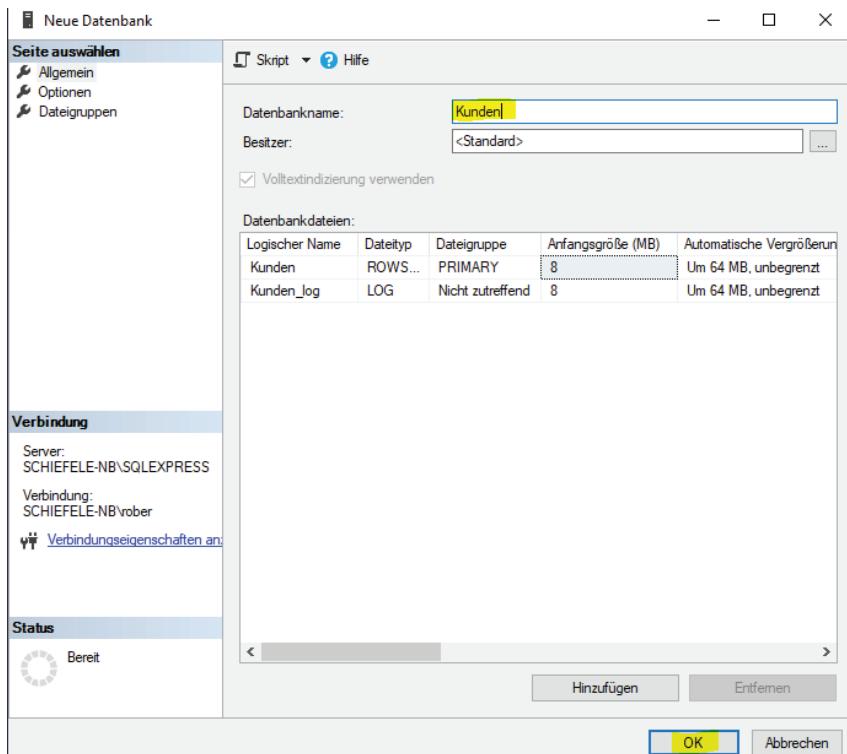


Abb. 15.2.2 Der Dialog „Neue Datenbank“

Im Dialog „Neue Datenbank“ vergeben wir für unsere neue Datenbank den Namen Kunden und klicken auf die Schaltfläche OK.

Das SQL-Management Studio legt jetzt eine neue Datenbank mit dem Namen Kunden an.

### 15.3 Den Entity Framework zum Projekt hinzufügen

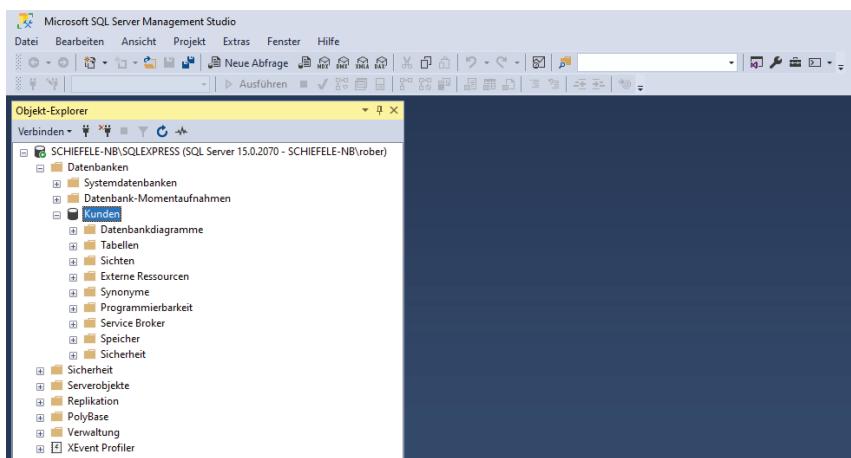


Abb. 15.2.3 Die neu angelegte Datenbank „Kunden“

Wir finden unsere neue Datenbank SQL-Management Studio im Ordner Datenbanken. Die neue Datenbank ist im Moment noch leer. In den folgenden Unterkapiteln werden wir die Datenbank mit Leben füllen.

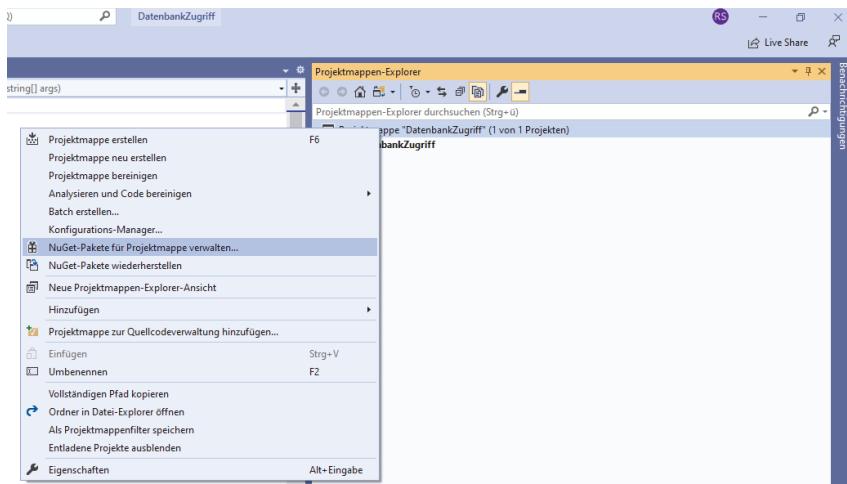
### 15.3 Den Entity Framework zum Projekt hinzufügen

Es gibt viele Möglichkeiten, mit C# auf eine Datenbank zuzugreifen. Aus Platzgründen werden wir hier nur den sogenannten Entity Framework betrachten. Der Entity Framework ist die modernste und inzwischen in der Praxis geläufigste Methode, auf den SQL-Server zuzugreifen. Zudem ist der Entity Framework die Methode, die die wenigsten Kenntnisse über den Datenbankserver selbst erfordert. Um den Entity Framework in einem Projekt verwenden zu können, müssen wir ihn zuerst dem Projekt hinzufügen. Die aktuelle Version des Entity Frameworks finden wir in NuGet. NuGet ist eine Internetplattform, auf der Entwickler Software in Form von Paketen veröffentlichen können. Auch Microsoft nutzt NuGet zur Bereitstellung von Paketen. NuGet ist vollständig in Visual Studio integriert.

15

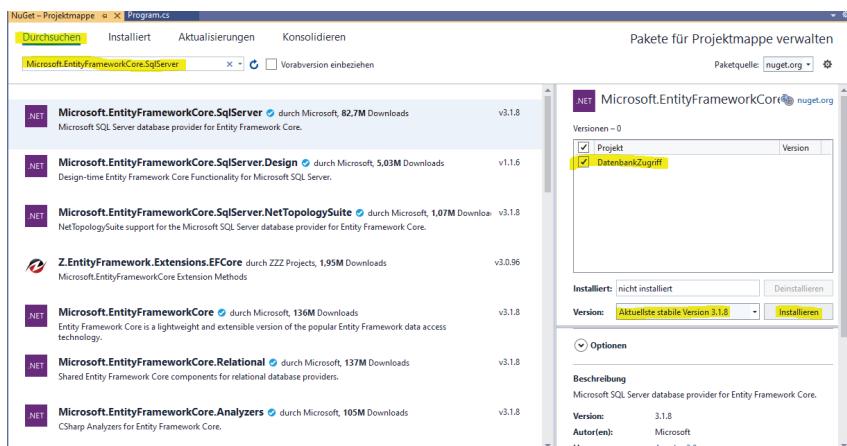
Wir erstellen ein neues Projekt vom Typ Konsolen-App mit dem Namen „Datenbank-Zugriff“ und klicken mit der rechten Maustaste auf die Projektmappe und wählen „NuGet-Pakete für Projektmappe verwalten ...“ aus dem Kontextmenü.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server



**Abb. 15.3.1** NuGet-Pakete für eine Projektmappe verwalten

Damit öffnen wir die Benutzeroberfläche der NuGet-Paketverwaltung.



**Abb. 15.3.2** Die Oberfläche des NuGet Paketmanagers

Wir klicken auf „Durchsuchen“ und geben im Suchfeld den Namen des gewünschten Pakets ein. Das Paket, das wir für den Entity-Framework benötigen, heißt **Microsoft.EntityFrameworkCore.SqlServer**. Als nächstes markieren wir das Paket in der Trefferliste und setzen neben dem Namen unseres Projekts einen Haken in der Checkbox. Im Feld Version wählen wir „Aktuellste stabile Version“ aus. Bei der Erstellung dieses Buchs war die Version 3.1.8 die aktuelle stabile Version. Zum Schluss klicken wir auf die

Schaltfläche „Installieren“. Den Dialog „Vorschau der Änderungen“ bestätigen wir mit „OK“ und den Dialog „Zustimmung zur Lizenz“ bestätigen wir mit „Ich stimme zu“.

Im Projektmappen Explorer sehen wir in unserem Projekt unter dem Knoten Abhängigkeiten/Pakete das mit NuGet installierte Paket Microsoft.EntityFrameworkCore.SqlServer.

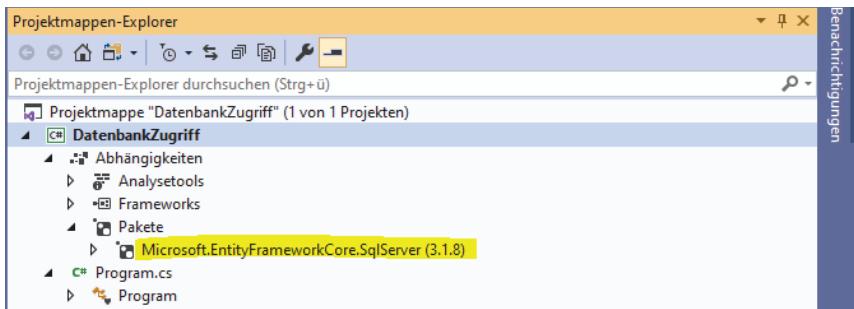


Abb. 15.3.3 Das installierte Paket

Als nächstes installieren wir noch mit Hilfe der NuGet-Paketverwaltung die beiden Pakete:

Microsoft.EntityFrameworkCore.Tools

Microsoft.EntityFrameworkCore.Design

## 15.4 Ein Entity-Modell erstellen

Nachdem wir den Entity-Framework in unser Projekt eingebunden haben, können wir ihn jetzt verwenden, um auf eine Datenbank zuzugreifen. Der Entity-Framework ist ein sogenannter Objekt-Relational-Mapper oder kurz OR-Mapper. Das heißt, er bildet Objekte der Datenbank auf Objekte der Programmiersprache, in unserem Fall C#, ab. In einer Datenbank werden Daten in Tabellen organisiert. Einige Leser, die noch keine Erfahrungen mit Datenbanken haben, werden bei dem Wort Tabellen sofort an Excel denken. Allerdings gibt es einen gravierenden Unterschied zwischen Datenbanktabellen und Exceltabellen. In einer Exceltabelle kann jede Zelle einen eigenen Datentyp haben. Das heißt, in einer Spalte einer Exceltabelle können eine Zahl, ein Text und ein Datum untereinander stehen. In einer Datenbanktabelle ist das nicht erlaubt. In einer Datenbanktabelle haben alle Zellen einer Spalte den gleichen Datentyp. Das heißt, eine Spalte einer Datenbanktabelle enthält entweder nur Zahlen oder nur Texte oder nur Datumswerte. Damit kann eine Klasse in C# eine Datenbanktabelle repräsentieren. Wenn wir in einer Datenbanktabelle alle gewerblichen Kunden eines

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Unternehmens speichern möchten, so können wir in der Datenbank eine Tabelle mit dem Namen Kunde anlegen. Für die Tabelle Kunde können wir dann zum Beispiel die Spalten Firma, Strasse, PLZ und Ort anlegen. In einem C# Programm können wir dann eine Klasse Kunde mit den Properties Firma, Strasse, PLZ und Ort anlegen. In einer Struktur wie zum Beispiel List<Kunde> können wir dann den vollständigen Inhalt der Tabelle Kunde speichern. Der Entity-Framework übersetzt dann zwischen der Datenbanktabelle und der C#-Klasse. Bevor wir von der Theorie in die Praxis wechseln, sollten wir noch den Begriff Entity (zu Deutsch: Entität) klären. Eine Entität ist ein Objekt in einer Datenbank, das - ähnlich wie ein Objekt - in der Objektorientierten Programmierung eine Instanz eines bestimmten Objekttyps repräsentiert. Entitäten werden in Tabellen gespeichert. Die Struktur der Tabelle definiert den Entitätstyp. Unter Struktur der Tabelle versteht man die Ausgestaltung der Tabelle mit Spalten, wobei jeder Spalte ein bestimmter Datentyp zugeordnet wird. Somit ist eine Entität eine Zeile in einer Tabelle.

In unserem Projekt, in das wir den Entity-Framework eingebunden haben, wollen wir ein sogenanntes Entity-Modell erstellen. Ein Entity-Modell ist eine Sammlung von Klassen, mit denen wir mit der Datenbank kommunizieren können. Zuerst erstellen wir einen Ordner in unserem Projekt. Das ist zwar nicht unbedingt notwendig, erhöht aber die Übersicht innerhalb des Projekts. Wir klicken mit der rechten Maustaste auf das Projekt und wählen „Hinzufügen/Neuer Ordner“ aus dem Kontextmenü.

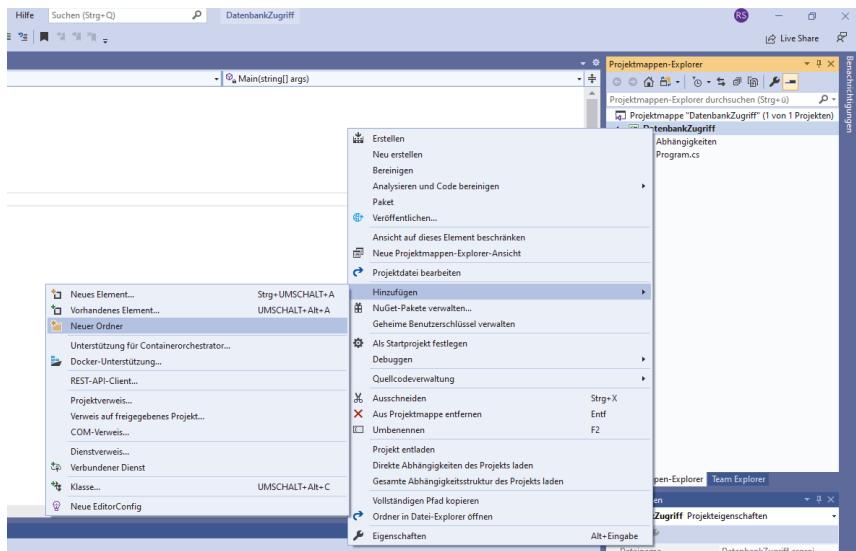


Abb. 15.4.1 Hinzufügen eines Ordners zum Projekt

Als Namen für den neuen Ordner vergeben wir „Modell“.

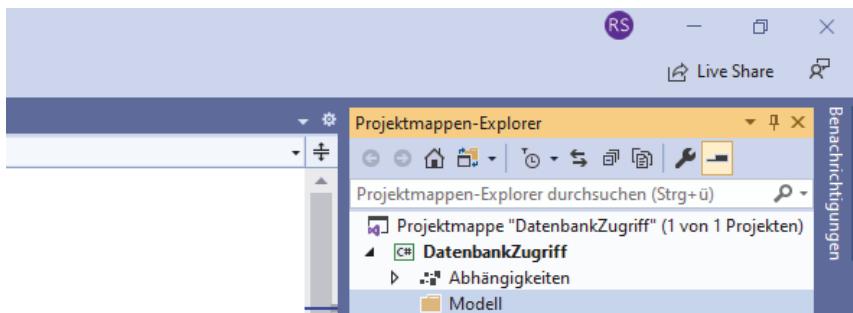


Abb. 15.4.2 Der neue Ordner „Modell“

Als nächstes legen wir zwei Datenklassen im Ordner „Modell“ an. Damit die Klassen im Ordner Modell landen, klicken wir mit der rechten Maustaste auf den Ordner und wählen „Hinzufügen/Klasse ...“ aus dem Kontextmenü.

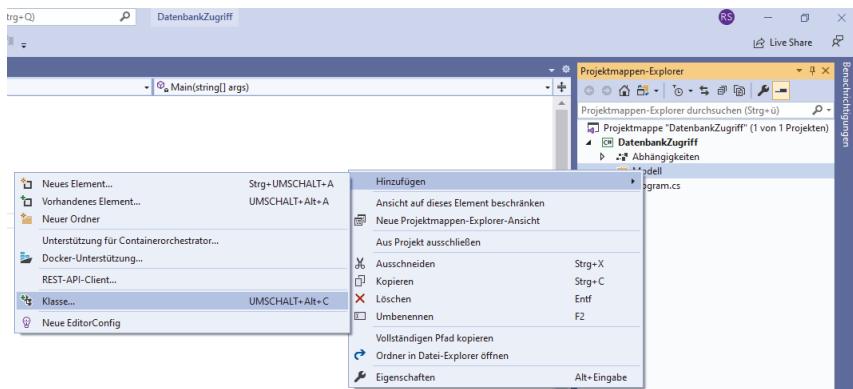


Abb. 15.4.3 Hinzufügen einer Klasse zu einem Ordner

Im folgenden Dialog vergeben wir den Namen `Kunde.cs` für die neue Klasse.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace DatenbankZugriff.Modell
6  {
7      class Kunde
8      {
9      }
10 }
```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Beachten Sie, dass die neue Klasse durch das Erstellen in einem eigenen Ordner auch einen eigenen Namensraum erhalten hat. Der Namensraum der Klasse erhält den Namen des Projekts und den Namen des Ordners durch einen Punkt getrennt. In unserem Beispiel also `DatenbankZugriff.Modell`. Wenn wir diese Klasse verwenden wollen, müssen wir sie mit `using DatenbankZugriff.Modell` einbinden. Als nächstes ändern wir die Klasse `Kunde` wie folgt ab.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace DatenbankZugriff.Modell
6 {
7     public class Kunde
8     {
9         public int KundenId { get; set; }
10        public string Firma { get; set; }
11        public string Strasse { get; set; }
12        public string PLZ { get; set; }
13        public string Ort { get; set; }
14    }
15 }
```

Die Klasse `Kunde` ist eine Entity-Klasse, deren Properties wir später mit Spalten einer Datenbanktabelle assoziieren wollen. Die Property `KundenId` soll eine Zahl enthalten, die den Kunden eindeutig identifiziert. Eine derartige Zahl wird in einer Datenbank auch als Primärschlüssel bezeichnet. Im Laufe dieses Unterkapitels werden wir diesen Begriff noch näher betrachten. Die zweite Entity-Klasse, die wir ebenfalls im Ordner `Modell` anlegen wollen, heißt `Ansprechpartner` und sieht wie folgt aus:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace DatenbankZugriff.Modell
6 {
7     public class Ansprechpartner
8     {
9         public int AnsprechpartnerId { get; set; }
10        public string Vorname { get; set; }
11        public string Nachname { get; set; }
12        public string Telefonnummer { get; set; }
13    }
14 }
```

Auch hier handelt es sich um eine Entity-Klasse, deren Properties später mit Tabellen Spalten assoziiert werden. Die Property `AnsprechpartnerId` repräsentiert den Primärschlüssel für `Ansprechpartner`.

Bisher haben wir nur zwei Datenklassen angelegt, die noch keine Verbindung zu einer Datenbank haben. Für diese Verbindung benötigen wir noch einen Datenbank-Kontext. Deshalb legen wir im Ordner Modell die Klasse KundenKontext mit folgendem Inhalt an:

```
1  using Microsoft.EntityFrameworkCore;
2  using System;
3  using System.Collections.Generic;
4  using System.Text;
5
6  namespace DatenbankZugriff.Modell
7  {
8      public class KundenKontext : DbContext
9      {
10         public DbSet<Kunde> Kunden { get; set; }
11         public DbSet<Ansprechpartner> Ansprechpartner { get;
12             set; }
13
14         protected override void OnConfiguring
15             (DbContextOptionsBuilder optionsBuilder)
16         {
17             optionsBuilder.UseSqlServer(@"Server=.\\
18             SQLEXPRESS;Database=Kunden;Trusted_Connection=True;");
19         }
20     }
21 }
```

Die Klasse KundenKontext wird von der Klasse DbContext aus der Klassenbibliothek Microsoft.EntityFrameworkCore abgeleitet und enthält die beiden public-Properties Kunden und Ansprechpartner die, die Tabellen Kunden und Ansprechpartner repräsentieren. Die Properties Kunden und Ansprechpartner haben den Typ DbSet<Kunde> beziehungsweise den Typ DbSet<Ansprechpartner>. Der Typ DbSet<TEntity> implementiert unter anderem das Interface IEnumerable<TEntity> und kann daher als Listenstruktur aufgefasst werden. Damit kann die Property Kunden einen oder mehrere Kunden enthalten. Analoges gilt für die Property Ansprechpartner. Außerdem überschreibt die Klasse KundenKontext die Methode OnConfiguring() der Klasse DbContext. Der Entity-Framework ruft diese Methode auf und übergibt ihr ein Objekt vom Typ DbContextOptionsBuilder. Wir rufen die Methode UseSqlServer() des DbContextOptionsBuilder-Objekts auf und übergeben ihm einen String. Dieser String ist der sogenannte Connectionstring. Er enthält alle Informationen, die nötig sind, um sich mit einem SQL-Server zu verbinden. In unserem Beispiel verbinden wir uns mit dem Server \SQLEXPRESS. Wobei der Punkt bedeutet, dass wir uns mit einer SQL-Server-Instanz verbinden wollen, die auf dem gleichen Computer läuft wie unser Programm. SQLEXPRESS ist der Instanz-Name, den wir bei der Installation des SQL-Servers im Kapitel „15.1 Microsoft SQL-Server installieren“ vergeben haben. Wenn Sie bei der Installation Ihres SQL-Servers einen anderen Instanz-Namen vergeben haben, verwenden Sie hier bitte diesen Namen. Wenn Sie Ihren SQL-Server als Default Instance installiert haben, verwenden Sie im Connectionstring für den Server

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

nur einen Punkt. Der nächste Teil des Connectionstrings legt die Datenbank fest, die wir verwenden wollen. In unserem Fall ist das die Datenbank Kunden, die wir im Kapitel „15.2 Eine Datenbank erstellen“ angelegt haben. Der letzte Parameter des Connectionstrings heißt `Trusted_Connections=True`. Damit legen wir fest, dass wir uns mit dem Windows-Benutzer, mit dem wir am Betriebssystem angemeldet sind, auch beim SQL-Server anmelden wollen. Das ist die einfachste Methode einen Connectionstring für eine Datenbankverbindung aufzubauen. Viele weitere Varianten von Connectionstrings für verschiedene Datenbanken finden Sie auf der Webseite <https://www.connectionstrings.com>.

Für unser Modell haben wir jetzt zwei Entity-Klassen und eine Datenbankkontext-Klasse erstellt. Aber unsere Datenbank enthält immer noch keine Tabellen. Um unser Modell auf die Datenbank zu übertragen, benutzen wir das sogenannte Migration Tool über die Paket-Manager-Konsole. Dazu klicken wir im Menü von Visual Studio auf Extras/NuGet-Paket-Manager/Paket-Manager-Konsole.

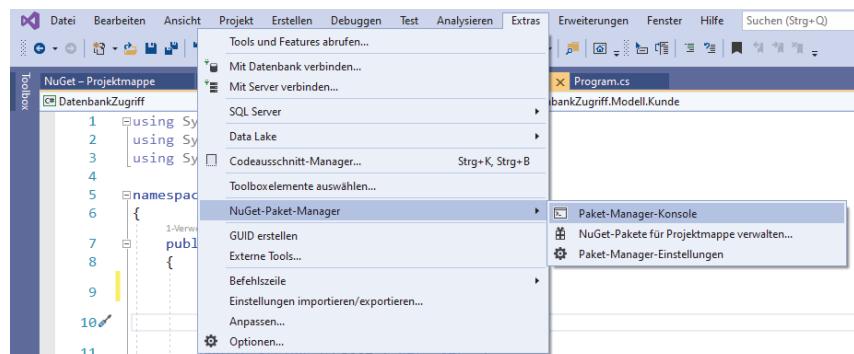


Abb. 15.4.4 Öffnen der Paket-Manager-Konsole

In der Paket-Manager-Konsole geben wir das Kommando „Add-Migration Initial“ ein und drücken die Enter-Taste.

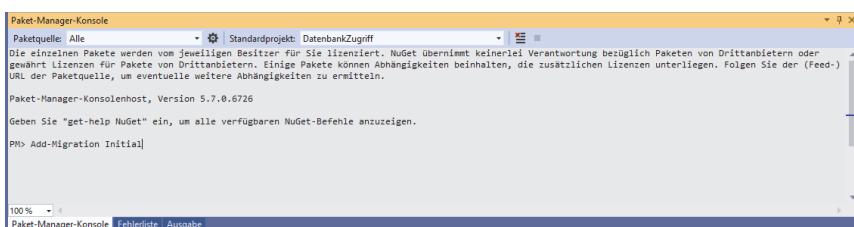
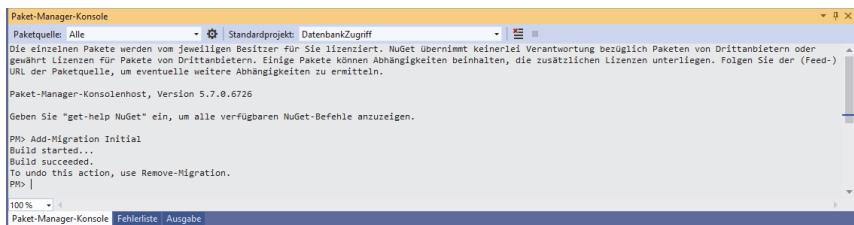


Abb. 15.4.5 Initialisieren der Migration

Bei erfolgreicher Ausführung des Kommandos sehen wir die folgende Ausgabe in der Paket-Manager-Konsole.



```
Paket-Manager-Konsole
Paketquelle: Alle - Standardprojekt: DatenbankZugriff
Die einzelnen Pakete werden vom jeweiligen Besitzer für Sie lizenziert. NuGet übernimmt keinerlei Verantwortung bezüglich Paketen von Drittanbietern oder gewährt Lizenzen für Pakete von Drittanbietern. Einige Pakete können Abhängigkeiten beinhalten, die zusätzlichen Lizenzen unterliegen. Folgen Sie der (Feed-)URL der Paketquelle, um eventuelle weitere Abhängigkeiten zu ermitteln.

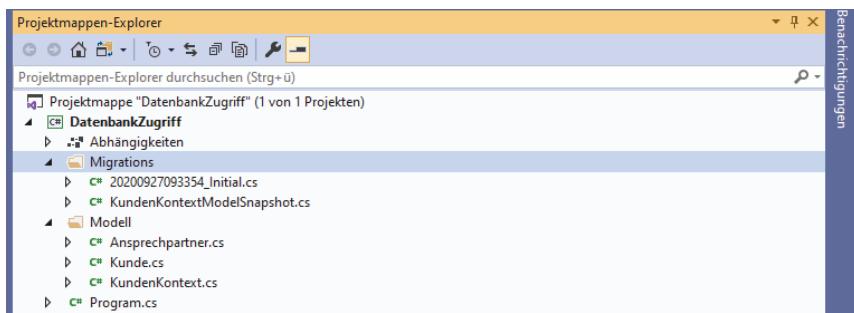
Paket-Manager-Konsolenhost, Version 5.7.0.6726
Geben Sie "get-help NuGet" ein, um alle verfügbaren NuGet-Befehle anzuzeigen.

PM> Add-Migration Initial
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>

100% ▶ Paket-Manager-Konsole Fehlerliste Ausgabe
```

**Abb. 15.4.6** Die Migration wurde erfolgreich initialisiert

Das Migration-Tool legt in unserem Projekt den Ordner Migrations an und darunter die Klassen, die für die Übertragung des Entity-Modells auf den SQL-Server benötigt werden.



**Abb. 15.4.7** Migration im Projektmappen Explorer

Um die Übertragung unseres Modells durchzuführen, verwenden wir dann das Kommando „Update-Database“ in der Paket-Manager-Konsole.



```
Paket-Manager-Konsole
Paketquelle: Alle - Standardprojekt: DatenbankZugriff
Die einzelnen Pakete werden vom jeweiligen Besitzer für Sie lizenziert. NuGet übernimmt keinerlei Verantwortung bezüglich Paketen von Drittanbietern oder gewährt Lizenzen für Pakete von Drittanbietern. Einige Pakete können Abhängigkeiten beinhalten, die zusätzlichen Lizenzen unterliegen. Folgen Sie der (Feed-)URL der Paketquelle, um eventuelle weitere Abhängigkeiten zu ermitteln.

Paket-Manager-Konsolenhost, Version 5.7.0.6726
Geben Sie "get-help NuGet" ein, um alle verfügbaren NuGet-Befehle anzuzeigen.

PM> Add-Migration Initial
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20200927093354_Initial'.
Done.
PM>

100% ▶ Paket-Manager-Konsole Fehlerliste Ausgabe
```

**Abb. 15.4.8** Das Modell wird auf die Datenbank übertragen

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Mit dem SQL-Server Management Studio können wir jetzt überprüfen, was das Migration-Tool in unserer Datenbank angelegt hat.

The screenshot shows the Microsoft SQL Server Management Studio interface. On the left, the Object Explorer displays the database structure under 'SCHIEFELE-NB\SQLExpress'. It shows the 'Kunden' table under the 'Tabellen' node. On the right, the 'SCHIEFELE-NB\SQL...nden - dbo.Kunden' table definition is shown in a grid:

Spaltenname	Datentyp	NULL-Werte...
KundeId	int	<input type="checkbox"/>
Firma	nvarchar(MAX)	<input checked="" type="checkbox"/>
Strasse	nvarchar(MAX)	<input checked="" type="checkbox"/>
PLZ	nvarchar(MAX)	<input checked="" type="checkbox"/>
Ort	nvarchar(MAX)	<input checked="" type="checkbox"/>

Abb. 15.4.9 Die Tabelle Kunden in der Datenbank

Das Migration-Tool hat die Tabelle Kunden in unserer Datenbank angelegt und die Spalte KundeId als Primärschlüssel festgelegt.

The screenshot shows the Microsoft SQL Server Management Studio interface. On the left, the Object Explorer displays the database structure under 'SCHIEFELE-NB\SQLExpress'. It shows the 'Anprechpartner' table under the 'Tabellen' node. On the right, the 'SCHIEFELE-NB\SQL...o.Anprechpartner' table definition is shown in a grid:

Spaltenname	Datentyp	NULL-Werte...
AnsprechpartnerId	int	<input type="checkbox"/>
Vorname	nvarchar(MAX)	<input checked="" type="checkbox"/>
Nachname	nvarchar(MAX)	<input checked="" type="checkbox"/>
Telefonnummer	nvarchar(MAX)	<input checked="" type="checkbox"/>

Abb. 15.4.10 Die Tabelle Anprechpartner in der Datenbank

Auch die Tabelle Anprechpartner wurde in der Datenbank angelegt und die Spalte AnsprechpartnerId als Primärschlüssel festgelegt.

Allerdings hat unser Modell noch einen kleinen logischen Fehler. Die Idee hinter unserem Model ist es, dass wir in der Tabelle Kunden die Adressen von gewerblichen Kunden eines Unternehmens ablegen wollen. Jeder dieser Kunden hat einen oder mehrere Anprechpartner. Wir wollen alle Anprechpartner aller Kunden gemeinsam in der Tabelle Anprechpartner speichern. In unserem bisherigen Modell haben wir keine Möglichkeit, die Information abzuspeichern, welcher Anprechpartner zu welchem Kunden gehört. Um das Speichern dieser Information zu ermöglichen, erweitern wir die Klasse Anprechpartner wie folgt:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace DatenbankZugriff.Modell
6 {
7     public class Ansprechpartner
8     {
9         public int AnsprechpartnerId { get; set; }
10        public int KundeId { get; set; }
11        public string Vorname { get; set; }
12        public string Nachname { get; set; }
13        public string Telefonnummer { get; set; }
14    }
15 }
16 }
```

Die Klasse `Ansprechpartner` erhält die zusätzliche Property `KundeId`. In der Property `KundeId` wollen wir die Id des Kunden, also den Wert der Property `KundeId` eines Objekts vom Typ `Kunde`, speichern, zu dem der jeweilige `Ansprechpartner` gehört.

Unser Modell in C# haben wir angepasst. Jetzt müssen wir diese Änderung nur noch auf die Datenbank übertragen. Dazu benötigen wir eine weitere Migration. In der Paket-Manager-Konsole erledigen wir das mit dem folgenden Kommando:

```
1 Add-Migration AddKundeId
```

Der Name `AddKundeId` ist in der Theorie beliebig, aber in der Praxis sollte er etwas mit der Datenbankänderung der jeweiligen Migration zu tun. Der Name muss zudem den Anforderungen für Klassennamen in C# genügen, da die Migration eine Klasse mit diesem Namen generiert. Um die Änderung zur Datenbank zu übertragen, benötigen wir noch das Kommando:

```
1 Update-Database
```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer on the left, under the database 'SCHIEFELE-NB\SQLEXPRESS', there is a tree view of databases, system databases, and tables. The 'Ansprechpartner' table is selected. On the right, the 'Ansprechpartner' table is displayed in a grid format with columns: Spaltenname, Datentyp, and NULL-Werte.. The new column 'KundeId' is highlighted in blue. The table structure is as follows:

Spaltenname	Datentyp	NULL-Werte..
AnsprechpartnerId	int	<input type="checkbox"/>
Vorname	nvarchar(MAX)	<input checked="" type="checkbox"/>
Nachname	nvarchar(MAX)	<input checked="" type="checkbox"/>
Telefonnummer	nvarchar(MAX)	<input checked="" type="checkbox"/>
KundeId	int	<input type="checkbox"/>

Abb. 15.4.11 Die Tabelle Ansprechpartner mit der neuen Spalte KundId

Wie zu erwarten, hat die Migration die neue Spalte in der Tabelle `Ansprechpartner` angelegt. Theoretisch könnten wir jetzt mit unserem Entity-Modell losprogrammieren und Daten lesen, schreiben, ändern und löschen. Wer aber schon mal mit Datenbanken gearbeitet hat, kennt mit Sicherheit die sogenannte Referentielle Integrität. Damit ist gemeint, dass eine Datenbank beziehungsweise in Ordnung ist. Für unsere Beispieldatenbank bedeutet das, dass wir zum Beispiel in der Spalte `KundeId` in der Tabelle `Ansprechpartner` den Wert 42 nur dann verwenden dürfen, wenn in der Tabelle `Kunde` eine Zeile existiert, die in der Spalte `KundeId` auch den Wert 42 hat. Ansonsten hätten wir einen `Ansprechpartner`, der einem nichtexistierenden Kunden zugeordnet ist. In der Datenbank wird die Referentielle Integrität sichergestellt, indem wir die Spalte `KundeId` in der Tabelle `Ansprechpartner` zum sogenannten Fremdschlüssel erklären und ihn so konfigurieren, dass er mit dem Primärschlüssel `KundeId` der Tabelle `Kunde` verbunden ist. Das Anlegen dieses Fremdschlüssels können wir auch wieder durch eine Migration erledigen. Dazu fügen wir der Klasse `Ansprechpartner` eine sogenannte Navigationproperty hinzu.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace DatenbankZugriff.Modell
6  {
7      public class Ansprechpartner
8      {
9          public int AnsprechpartnerId { get; set; }
10         public int KundeId { get; set; }
11         public string Vorname { get; set; }

```

```

12     public string Nachname { get; set; }
13     public string Telefonnummer { get; set; }
14
15     //Navigation Properties
16     public virtual Kunde Kunde { get; set; }
17 }
18 }
```

Unsere Navigationproperty heißt Kunde und ist vom Typ Kunde, zudem ist sie virtuell deklariert. Das ist nötig, um das sogenannte Lazy Loading des Entity-Frameworks zu unterstützen. Auf Lazy Loading werden wir später noch genauer eingehen. Die Property heißt Navigationproperty, weil dort der Kunde, zu dem der jeweilige Ansprechpartner gehört, gespeichert ist. Wir können damit also von der Entität Ansprechpartner zur Entität Kunde wechseln. Damit wir aus der Navigationproperty Kunde einen Fremdschlüssel in der Datenbank erzeugen können, erstellen wir mit folgendem Kommando in der Paket-Manager-Konsole eine neue Migration.

```
1 Add-Migration AddFremdSchluesselAnsprechPartnerKundeId
```

Danach übertragen wir die Migration mit Update-Database auf die Datenbank.

The screenshot shows two windows from SQL Server Management Studio. On the left, the Object Explorer displays the database structure for 'SCHIEFEL-NB\SQLEXPRESS'. It shows the 'Kunden' table under 'Tabelle' and the 'Ansprechpartner' table under 'Ansprechpartner'. The 'Ansprechpartner' table has a primary key 'AnsprechpartnerId' and a foreign key 'KundenId' which points to the 'Kunden' table's primary key 'KundenId'. On the right, the 'FK\_Anspprechpartner\_Kunden\_KundenId' relationship configuration window is open. It shows the relationship between 'Ansprechpartner' and 'Kunden' tables. The 'Fremdschlüssel- und Spaltenpezifikation' section is selected, showing 'Ansprechpartner' as the foreign key table and 'KundenId' as the foreign key column. The 'Kunden' table is listed as the primary key table with its primary key 'KundenId'.

Abb. 15.4.12 Der Fremdschlüssel FK\_Anspprechpartner\_Kunden\_KundenId in der Datenbank

Der Fremdschlüssel „Beziehung“ zwischen den Tabellen Kunden und Ansprechpartner ist jetzt auch in der Datenbank angelegt.

Durch Navigationproperty Kunde in der Klasse Ansprechpartner können wir von einem Ansprechpartner-Objekt auf den dazugehörigen Kunden zugreifen. Natürlich wollen wir auch den umgekehrten Fall in unserem Modell abbilden. Das heißt, wir

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

wollen in der Klasse Kunde eine Navigationproperty, die alle Ansprechpartner enthält, die zu diesem Kunden gehören. Dafür ändern wir die Klasse Kunde wie folgt ab:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace DatenbankZugriff.Modell
6  {
7      public class Kunde
8      {
9          public int KundeId { get; set; }
10         public string Firma { get; set; }
11         public string Strasse { get; set; }
12         public string PLZ { get; set; }
13         public string Ort { get; set; }
14
15         //Navigation Properties
16         public virtual ICollection<Ansprechpartner>
17             Ansprechpartner { get; set; }
18     }
19 }
```

Die Navigationproperty `Ansprechpartner` ist vom Typ `ICollection<Ansprechpartner>`. Prinzipiell würden hier auch andere Auflistungstypen funktionieren, aber `ICollection<T>` ist hier der von Microsoft empfohlene Typ. Zudem ist die Navigationproperty virtuell deklariert, was wir wiederum für die Unterstützung von Lazy Loading benötigen. Doch dazu später mehr. Für diese Navigationproperty benötigen wir keine Übertragung auf die Datenbank, da der SQL-Server wie die meisten anderen Datenbank dafür keine Entsprechung bereitstellt.

Im Prinzip haben wir jetzt unser Entity-Modell vollständig implementiert. Allerdings hat die Migration für das Erstellen der Datenbank einige Annahmen selbstständig getroffen. Die Namen von Tabellen heißen genauso wie die zugehörigen Properties in der Klasse `KundenKontext`. Die Namen der Tabellenspalten heißen genauso wie die Properties der zugehörigen Entity-Klassen. Wenn der Typ einer Property in einer Entity-Klasse in C# Null-Werte erlaubt, dann erlaubt auch die Datenbank Null-Werte. Die Namen der Schlüssel heißen wie die Entity-Klassen, auf die sich die Schlüssel beziehen. Daher hat die Migration auch die Primär- und Fremdschlüssel richtig angelegt. Wie wir auf diese Zuordnungen Einfluss nehmen können, werden wir jetzt zum Schluss dieses Unterkapitels betrachten.

Die Zuordnung von Namen, also von Tabellennamen und Klassennamen, von Spaltennamen und Propertynamen, sowie die Erstellung und Konfiguration von Fremd- und Primärschlüsseln, können wir in unserem Entity-Modell, ähnlich wie bei der Serialisierung von Objekten zu XML, mit Klassen- und Property-Attributen festlegen. Für die Entity-Klasse sieht das wie folgt aus:

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9      [Table("Kunde")]
10     public class Kunde
11     {
12         [Key]
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         [Column("id")]
15         public int KundenId { get; set; }
16
17         [Required]
18         [MaxLength(50)]
19         [Column("firma")]
20         public string Firma { get; set; }
21
22         [MaxLength(50)]
23         [Column("strasse")]
24         public string Strasse { get; set; }
25
26         [MaxLength(5)]
27         [Column("plz")]
28         public string PLZ { get; set; }
29
30         [MaxLength(50)]
31         [Column("ort")]
32         public string Ort { get; set; }
33
34         //Navigation Properties
35         public virtual ICollection<Ansprechpartner>
36         Ansprechpartner { get; set; }
37     }
38 }
```

Die Klasse Kunde erhält das Klassenattribut `[Table("Kunde")]`, damit legen wir fest, dass die Klasse Kunde mit der Tabelle Kunde verbunden ist. Ohne dieses Attribut hat die Migration den Namen Kunden für die Tabelle verwendet, da die entsprechende Property im Datenbankkontext Kunden heißt. Die Property KundenId bekommt das Propertyattribut `[Column("id")]`, dadurch ordnen wir die Property KundenId der Tabellenspalte id zu. Dieser Name entspricht jetzt nicht mehr der Konvention des Entity-Frameworks für Primärschlüsselnamen. Daher weiß die Migration auch nicht, dass diese Property der Primärschlüssel sein soll. Deswegen legen wir mit dem Propertyattribut `[Key]` fest, dass die Property KundenId der Primärschlüssel ist. Die Property KundenId erhält noch das Propertyattribut `[DatabaseGeneratedOption.Identity]`. Dieses Propertyattribut macht die Spalte id zu einer sogenannten Identitätsspalte. Das heißt, dass wir beim ersten Anlegen eines Kunden in der Datenbank keinen Wert für KundenId vergeben müssen. Das

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

wird von der Datenbank automatisch erledigt. Die Datenbank beginnt bei dem Wert eins und zählt diesen für jeden neu angelegten Kunden um eins hoch. Alle weiteren Properties erhalten über das Property Attribut [Column()] eine explizite Zuordnung zu einer Tabellenspalte. Zwischen einer Datenbank Entität und eines Entity-Objekts in C# gibt es noch einen weiteren bedeutenden Unterschied. Eine C# Property, die Textinformation speichern soll, erhält den Typ `string`. Dabei wird keine Angabe gemacht, wie groß der zu speichernde Text werden kann. Daher hat die Migration bei den Tabellenspalten für String-Properties in der Datenbank bisher den Datentyp `nvarchar(Max)` vergeben. Mit diesem Datentyp können wir Textdaten bis zur maximalen, vom SQL-Server unterstützten Größe speichern. In der aktuellen Version sind das 2 Gigabyte. Wenn wir beim SQL-Server für Textspalten generell die maximale Größe erlauben, wirkt sich das negativ auf die Performance unserer Datenbank aus. Daher verwenden wir für die Properties `Firma`, `Strasse` und `Ort` das Propertyattribut [MaxLength(50)], mit dem wir die maximale Größe auf 50 Zeichen begrenzen. Für die Property `PLZ` begrenzen wir die maximale Größe auf fünf Zeichen. Für die Klasse `Ansprechpartner` legen wir die Klassen- und Eigenschaftsattribute wie folgt fest:

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9      [Table("Ansprechpartner")]
10     public class Ansprechpartner
11     {
12         [Key]
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         [Column("id")]
15         public int AnsprechpartnerId { get; set; }
16
17         [ForeignKey("Kunde")]
18         [Column("kundenid")]
19         public int KundeId { get; set; }
20
21         [Required]
22         [MaxLength(50)]
23         [Column("vorname")]
24         public string Vorname { get; set; }
25
26         [Required]
27         [MaxLength(50)]
28         [Column("nachname")]
29         public string Nachname { get; set; }
30
31         [MaxLength(20)]
32         [Column("telefon")]
33         public string Telefonnummer { get; set; }
34 }
```

```

35     //Navigation Properties
36     public virtual Kunde Kunde { get; set; }
37 }
38 }
```

Mit dem Klassenattribut [Table („Ansprechpartner“)] ordnen wir die Klasse der Tabelle Ansprechpartner zu. Die Property AnsprechpartnerId erhält die Attribute:

```

1 [Key]
2 [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
3 [Column("id")]
```

Damit legen wir fest, dass die Property AnsprechpartnerId mit der Tabellenspalte id verbunden wird, eine Identitätsspalte sowie der Primärschlüssel der Tabelle ist. Die Property KundenId wird mit dem Attribut [Column („kundenid“)] die Tabellenspalte kundenid zugeordnet. Des Weiteren erhält die Property KundenId das Attribut [ForeignKey („Kunde“)]. Damit machen wir die Property zum Fremdschlüssel, der auf die Tabelle Kunde zeigt. Die Properties Vorname, Nachname, Telefonnummer werden mit dem Column-Attribut den Tabellenspalten vorname, nachname und telefon zugeordnet. Die Properties Vorname und Nachname begrenzen wir mit dem MaxLength-Attribute auf 50 Zeichen und die Property Telefonnummer begrenzen wir auf 20 Zeichen. Die Property Vorname erhält zusätzlich noch das Attribut [Required], dadurch erlauben wir für die zugehörige Tabellenspalten keine Null-Werte. Nachdem wir in unserem Model die Klassen- und Eigenschaftsattribute gesetzt haben, erstellen wir eine neue Migration und übertragen sie an die Datenbank.

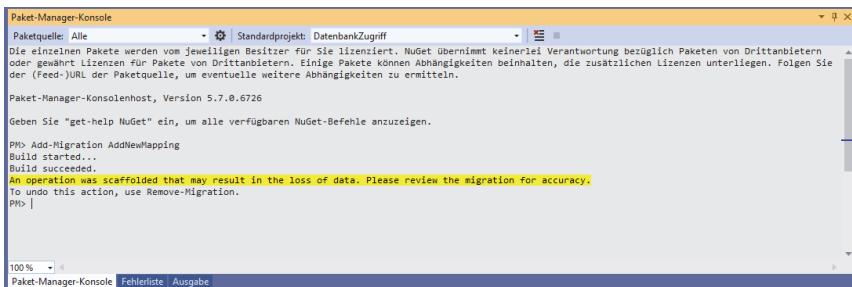


Abb. 15.4.13 Die Migration für die neuen Zuordnungen

Bei dieser Migration erhalten wir eine Warnung. Da wir mit der Migration den Typ von Tabellenspalten von nvarchar(Max) auf nvarchar(50) beziehungsweise nvarchar(20) und nvarchar(5) ändern, könnten wir Daten verlieren, falls wir in unseren Tabellen schon Daten hätten, die mehr als 50 beziehungsweise 20 und 5 Zeichen haben. Aber da wir unsere Tabellen noch keine Daten geschrieben haben, können wir diese Warnung ignorieren. Damit haben wir unser Datenmodell fertiggestellt.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Im nächsten Unterkapitel werden wir mit Hilfe dieses Modells Daten zu unseren Tabellen hinzufügen, lesen, ändern und löschen. Diese vier essenziellen Datenbankoperationen werden auch als die CRUD-Operationen bezeichnet, dabei stehen die Buchstaben CRUD für die englischen Begriffe:

- ▶ Create (erzeugen, hinzufügen)
- ▶ Read (lesen)
- ▶ Update (ändern)
- ▶ Delete (löschen)

### 15.5 Lesen, ändern, hinzufügen und löschen von Daten

In diesem Kapitel werden wir zeigen, wie CRUD-Operationen mit dem Entity-Framework implementiert werden können. Dazu schreiben wir uns eine sogenannte CRUD-Klasse. Unsere CRUD-Klasse für Kunden und die zugehörigen Ansprechpartner nennen wir `KundeCRUD`. Die Implementierung der Klasse und der Methode `Create()` sieht wie folgt aus:

```
1  using DatenbankZugriff.Modell;
2  using System;
3  using System.Collections.Generic;
4  using System.Text;
5
6  namespace DatenbankZugriff.CRUD
7  {
8      public class KundeCRUD
9      {
10          private KundenKontext kontext;
11
12          public KundeCRUD()
13          {
14              kontext = new KundenKontext();
15          }
16
17          public void Create(Kunde kunde)
18          {
19              kontext.Kunden.Add(kunde);
20              kontext.SaveChanges();
21          }
22      }
23  }
```

Die Klasse `KundeCRUD` enthält die private Membervariable `kontext` vom Typ `KundenKontext` aus unserem Entity-Modell. Im Konstruktor wird `kontext` mit einer neuen Instanz von `KundenKontext` initialisiert. Die Methode `Create()` erwartet als Übergabeparameter ein Objekt vom Typ `Kunde` und übergibt es der Methode `Add()` der Property `Kunden` des `KundenKontext`-Objekts, welches in der Member-

## 15.5 Lesen, ändern, hinzufügen und löschen von Daten

variablen Kunde gespeichert ist. Die Property Kunden ist vom Typ `DbSet<Kunde>`, welches die Tabelle Kunde repräsentiert. Die Methode `Add()` fügt der Listenstruktur Kunden ein neues Element hinzu. Allerdings passiert das nur im Hauptspeicher des Computers. Erst der Aufruf `kontext.SaveChanges()` speichert den neuen Kunden in der Datenbank. Wie wir die Klasse `KundeCRUD` verwenden können, zeigt das folgenden Testprogramm:

```

1  using DatenbankZugriff.CRUD;
2  using DatenbankZugriff.Modell;
3  using System;
4
5  namespace DatenbankZugriff
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             KundeCRUD kundeCRUD = new KundeCRUD();
12
13             Kunde neuerKunde = new Kunde
14             {
15                 Firma = "Duck Industries Inc.",
16                 Ort = "Entenhausen",
17                 PLZ = "12345",
18                 Strasse = "Enten-Allee 1-25"
19             };
20
21             kundeCRUD.Create(neuerKunde);
22
23             Console.WriteLine($"Firma: {neuerKunde.Firma} mit
24             Id: {neuerKunde.KundeId} angelegt.");
25         }
26     }
27 }
```

Zuerst erzeugen wir eine neue Instanz von `KundeCRUD`. Danach erstellen wir ein neues Kunde-Objekt und weisen dessen Properties mit Hilfe eines Objektinitialisierer Werte zu. Danach rufen wir die Methode `Create()` des `KundeCRUD`-Objekts auf und übergeben ihr das vorher erstellte Kunde-Objekt. Beachten Sie dabei, dass der Property `KundeId` kein expliziter Wert zugewiesen wurde. Da die Property vom Typ `int` ist, hat sie nach dem Erstellen des Kunde-Objekts den Wert 0. Nach dem Anlegen des neuen Kunden in der Datenbank geben wir die Properties `Firma` und `KundeId` des neuen Kunden am Bildschirm aus.

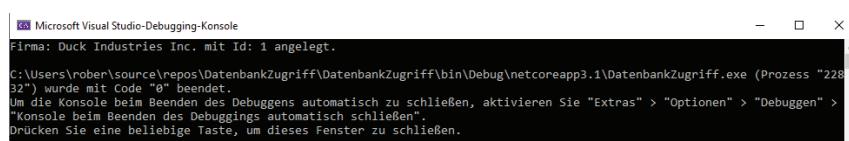


Abb. 15.5.1 Der erste Kunde wurde in der Datenbank angelegt.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Die Property `KundeId` des neuen Kunden hat den Wert 1. Dieser Wert wurde von der Datenbank vergeben, da die zugehörige Tabellenspalte `id` eine Identitätsspalte ist. Der Entity-Framework sorgt auch dafür, dass der von der Datenbank erzeugte Wert für die Tabellenspalte `id` der Property `KundeId` des hinzugefügten Kunde-Objekts zugewiesen wird. Mit dem SQL-Management Studio können wir den Inhalt der Tabelle Kunde überprüfen.

Abb. 15.5.2 Der erste Kunde in der Datenbank

Die Methode `Create()` der Klasse `KundeCRUD` führt zwei Anweisungen nacheinander aus, die erste Anweisung fügt dem Entity-Model ein neues Kunde-Element hinzu und die zweite Anweisung schreibt alle noch nicht gespeicherten Änderungen in die Datenbank. Allerdings müssen wir nicht jeden neuen Kunden einzeln in die Datenbank schreiben, daher erweitern wir die Klasse `KundeCRUD` mit einer weiteren Überladung für die Methode `Create()`:

```
1 public void Create(IEnumerable<Kunde> kunden)
2 {
3     kontext.Kunden.AddRange(kunden);
4     kontext.SaveChanges();
5 }
```

Diese Überladung der Methode `Create()` erwartet einen Parameter vom Typ `IEnumerable<Kunde>` und kann damit die Daten von mehreren Kunden auf einmal entgegennehmen. Mit der Methode `AddRange()` der Property `Kunden` des Kontexts fügen wir die übergebenen Kunden unserem Entity-Modell hinzu. Die Methode `SaveChanges()` schreibt dann alle Kunden auf einmal in die Datenbank. Mit dem folgenden Testprogramm können wir diese Methode überprüfen.

```
1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
```

## 15.5 Lesen, ändern, hinzufügen und löschen von Daten

```

 9     {
10         static void Main(string[] args)
11     {
12         KundeCRUD kundeCRUD = new KundeCRUD();
13
14         var kunden = new List<Kunde>
15         {
16             new Kunde
17             {
18                 Firma = "Kanzlei Müller & Söhne",
19                 Ort = "Neustadt",
20                 PLZ = "54321",
21                 Strasse = "Hauptstrasse 1"
22             },
23             new Kunde
24             {
25                 Firma = "Bürobedarf Meier",
26                 Ort = "Musterhausen",
27                 PLZ = "74851",
28                 Strasse = "Neue Gasse 5"
29             }
30         };
31
32         kundeCRUD.Create(kunden);
33
34         foreach (var kunde in kunden)
35         {
36             Console.WriteLine($"Firma: {kunde.Firma} mit Id: {kunde.KundeId} angelegt.");
37         }
38     }
39 }
40 }
41 }
```

Zuerst erstellen wir ein KundeCRUD-Objekt, dann ein Objekt von Typ `List<Kunde>` und weisen diesem Objekt mit Hilfe eines Objektinitialisierers zwei Kunden-Objekte mit Daten zu. Wie im vorherigen Testprogramm weisen wir hier wieder keine Werte für die `KundeId` Properties der `Kunde`-Objekte zu. Die Listenstruktur `kunden` übergeben wir an die zweite Überladung der Methode `Create()` des `KundeCRUD`-Objekts und speichern sie damit in der Datenbank. Nach dem Speichern laufen wir mit einer `foreach`-Schleife durch unsere Kundenliste und geben die Properties `Firma` und `KundeId` am Bildschirm aus.

15



The screenshot shows the Microsoft Visual Studio Debugging Console window. It displays the output of the C# code execution. The console shows two lines of text output:

```

Microsoft Visual Studio-Debugging-Konsole
Firma: Kanzlei Müller & Söhne mit Id: 2 angelegt.
Firma: Bürobedarf Meier mit Id: 3 angelegt.

```

Below the output, there is some standard Windows command-line cleanup text:

```

C:\Users\rober\source\repos\DatenbankZugriff\DatenbankZugriff\bin\Debug\netcoreapp3.1\DatenbankZugriff.exe (Prozess "3236") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 15.5.3 Zwei weitere Kunden wurden in der Datenbank angelegt.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Auch hier wurden die Werte für die Tabellenspalte `id` durch die Datenbank vergeben und der Property `KundeId` der Kunde-Objekte zugewiesen. Mit dem SQL-Management Studio können wir überprüfen, ob die neuen Kunden auch tatsächlich in der Datenbank gespeichert wurden.

	id	firma	strasse	plz	ort
1	1	Duck Industries Inc.	Erten-Allee 1-25	12345	Entenhausen
2	2	Kanzel Müller & Söhne	Hauptstrasse 1	54321	Neustadt
3	3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen

Abb. 15.5.4 Weitere Kunden in der Datenbank

Als nächstes wollen wir die drei Kunden, die wir in die Datenbank geschrieben haben, wieder auslesen. Dazu erweitern wir die Klasse `KundeCRUD` um eine weitere Methode.

```

1 public IQueryables<Kunde> LeseAlleKunden()
2 {
3     return kontext.Kunden;
4 }
```

Die Methode `LeseAlleKunden()` ist sehr einfach. Sie gibt lediglich die Property `Kunden` des Kontexts als `IQueryables<Kunde>` zurück. `IQueryables<T>` ist ein Interface, das von `DbSet<T>` implementiert wird. Das Interface funktioniert so ähnlich wie `IEnumerable<T>`, ist aber speziell für den Zugriff auf Datenbanken zugeschnitten. Wie `IEnumerable<T>` wird `IQueryables<T>` erst dann ausgeführt, wenn auf die Daten zugegriffen wird, also wenn wir die entsprechende Auflistung mit einer `foreach`-Schleife durchlaufen oder wenn wir die Erweiterungsmethode `ToList()` ausführen, die auch für `IQuerables<T>` existiert. `IQueryables<T>` ist auch das zentrale Interface des Linq-Providers „Linq to Entities“, welcher eine essenzielle Komponente des Entity-Frameworks ist. Mit dem folgenden Testprogramm können wir die Methode `LeseAlleKunden()` ausprobieren.

```

1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
9     {
10         static void Main(string[] args)
```

## 15.5 Lesen, ändern, hinzufügen und löschen von Daten

```

11     {
12         KundeCRUD kundeCRUD = new KundeCRUD();
13
14         var kunden = kundeCRUD.LeseAlleKunden();
15
16         foreach(var kunde in kunden)
17         {
18             Console.WriteLine(${"kunde.KundeId": {kunde.
19                 Firma}; {kunde.Strasse}; {kunde.PLZ} {kunde.
20                 Ort}}");
21         }
22     }
23 }
24 }
```

Wir erzeugen eine Instanz der Klasse `KundeCRUD`, rufen die Methode `LeseAlleKunden()` und speichern das Ergebnis in der Variablen `kunden`. Dann durchlaufen wir die Listenstruktur `kunden` mit einer `foreach`-Schleife. In der Schleife geben wir den aktuellen Kunden am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
1: Duck Industries Inc.; Enten-Allee 1-25; 12345 Entenhause
2: Kanzlei Müller & Söhne; Hauptstrasse 1; 54321 Neustadt
3: Bürobedarf Meier; Neue Gasse 5; 74851 Musterhausen

C:\Users\rober\source\repos\DatensbankZugriff\DatensbankZugriff\bin\Debug\netcoreapp3.1\DatensbankZugriff.exe (Prozess "141
20") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 15.5.5 Alle Kunden aus der Datenbank

Als nächstes schreiben wir die Methode `LeseKunde()`, mit der wir einen einzelnen Kunden anhand seines Primärschlüssel aus der Datenbank lesen.

```

1 public Kunde LeseKunde(int id)
2 {
3     return LeseAlleKunden().First(k => k.KundeId == id);
4 }
```

Die Methode `LeseKunde()` erwartet die Id des gesuchten Kunden als Übergabeparameter und ruft die Methode `LeseAlleKunden()` auf und verkettet diese mit der Linq-Methode `First()`. Der Methode `First()` übergeben wir den Lambda-Ausdruck `k => k.KundeId == id`, damit wählen wir den Kunden aus, dessen Wert der Property `KundeId` dem Übergabeparameter `id` entspricht. Diese Methode wird Ihr Ergebnis immer sehr schnell liefern, auch wenn die Tabelle `Kunde` beispielsweise zehn Millionen Kunden enthält. Das liegt an der speziellen, für Datenbanken optimierten Implementierung des Linq-Providers „Linq to Entities“. Es werden nämlich nicht zuerst alle Kunden aus der Datenbank gelesen und dann auf diesen Kunden der Ausdruck `First(k => k.KundeId == id)` ausgeführt, sondern Linq to Entities erzeugt aus dem gesamten Ausdruck eine SQL-Abfrage, sendet diese Abfrage an den SQL-Server und erhält den gesuchten Kunden vom SQL-Server zurück. Das heißt, die

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Suche nach dem Kunden mit der gewünschten Id erledigt der SQL-Server und der ist für derartige Aufgaben sehr stark optimiert. Mit dem folgenden Programm können wir die Methode `LeseKunde()` testen.

```

1  using DatenbankZugriff.CRUD;
2  using DatenbankZugriff.Modell;
3  using System;
4  using System.Collections.Generic;
5
6  namespace DatenbankZugriff
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             KundeCRUD kundeCRUD = new KundeCRUD();
13             var kunde = kundeCRUD.LeseKunde(2);
14             Console.WriteLine($"{{kunde.KundeId}}: {{kunde.Firma}};
15             {{kunde.Strasse}}; {{kunde.PLZ}} {{kunde.Ort}}");
16         }
17     }
18 }
```

Wir erzeugen ein `KundeCRUD`-Objekt und rufen dessen Methode `LeseKunde()` auf. Wir übergeben der Methode den Wert zwei und lesen damit den Kunden mit der Id zwei aus der Datenbank. Zum Schluss geben wir den Kunden am Bildschirm aus.

**Abb. 15.5.6** Ein Kunde aus der Datenbank

Nachdem wir Daten in die Datenbank geschrieben und wieder gelesen haben, wenden wir uns erneut dem Schreiben in die Datenbank zu. Diesmal untersuchen wir, wie wir für Kunden, die bereits in der Datenbank gespeichert sind, Ansprechpartner in die Datenbank schreiben können. Dazu erweitern wir die Klasse `KundeCRUD` um die Methode `CreateAnsprechpartner()`.

```

1  public void CreateAnsprechpartner(int kundeId, Ansprechpartner
2  ansprechpartner)
3  {
4      var kunde = kontext.Kunden
5          .Include(a => a.Ansprechpartner)
6          .First(k => k.KundeId == kundeId);
7      kunde.Ansprechpartner.Add(ansprechpartner);
8      kontext.SaveChanges();
9  }
```

Die Methode `CreateAnsprechpartner()` erwartet als Übergabeparameter die Id eines Kunden und ein Ansprechpartner-Objekt. Das Ansprechpartner-Objekt soll in die Tabelle Ansprechpartner in die Datenbank geschrieben werden und in der Property `KundeId` die übergebene Id eines Kunden erhalten. Das können wir auf verschiedene Arten implementieren. Eine alternative Implementierung werden wir später betrachten. Als erstes lesen wir den Kunden mit der übergebenen Id aus der Datenbank mit einem relativ komplizierten Ausdruck aus, obwohl wir bereits die Methode `LeseKunde()` geschrieben haben, die den Job vermeintlich auch erledigen könnte. Aber hier sehen wir zum ersten Mal die Auswirkungen des sogenannten Lazy Loadings (zu Deutsch: faules Laden). Das bedeutet, dass bei einem Datenbankzugriff mit dem Entity-Framework nur die Daten geladen werden, die unbedingt notwendig sind. Wenn wir also nur Kunden-Objekte laden, enthalten die Kunden-Objekte nur die Daten aus der Tabelle Kunde, auch wenn wir eine Navigationproperty `Ansprechpartner` vom Typ `ICollection<Ansprechpartner>` haben, hat diese zunächst den Wert null, auch wenn für den Kunden `Ansprechpartner` in der Datenbank gespeichert sind. Was auf den ersten Blick seltsam klingt, ist aber bei genauerer Betrachtung sehr sinnvoll. Stellen Sie sich vor, wir erstellen eine Datenbank für einen Autohändler und implementieren ein Entity-Modell für seine gewerblichen Kunden. Eine Kundenentität hätte eine Navigationproperty für mehrere Niederlassungen. Eine Niederlassungsentität hätte eine Navigationproperty für mehrere Fahrzeug. Eine Fahrzeugentität hätte eine Navigationproperty für mehrere Extras. Wenn wir jetzt davon ausgehen, dass der Händler 1000 Kunden hat und jeder Kunde im Schnitt 5 Niederlassungen und jede Niederlassung im Schnitt 100 Fahrzeuge und jedes Fahrzeug im Schnitt 20 Extras, dann hat die Kundentabelle 1000 Zeilen und die Niederlassungstabelle  $1000 \cdot 5 = 5000$  Zeilen. Die Fahrzeugtabelle hätte  $1000 \cdot 100 = 100.000$  Zeilen. Die Tabelle für Extras hätte  $1000 \cdot 5 \cdot 100 = 500.000$  Zeilen. Wenn wir kein Lazy Loading hätten, und wir würden mit einem Datenzugriff alle 1000 Kunden lesen, würden wir in Wirklichkeit die ganze Datenbank in den Hauptspeicher laden. In diesem Beispiel wären das  $10.000.000$  Extras +  $500.000$  Fahrzeuge +  $5000$  Niederlassungen + 1000 Kunden =  $10.506.000$  Objekte. Das würde zu sehr langen Ladezeiten führen. Und wenn mehrere Mitarbeiter des Händlers gleichzeitig auf die 1000 Kunden zugreifen, wäre das Netzwerk schnell überlastet.

Jetzt betrachten wir die Anweisung der Methode `CreateAnsprechpartner()`, die den Kunden mit der übergebenen Id aus der Datenbank liest:

```
1 var kunde = kontext.Kunden
2     .Include(a => a.Anprechpartner)
3     .First(k => k.KundeId == kundeId);
```

Den ersten Teil `kontext.Kunden` kennen wir bereits. Er bedeutet: „Nimm alle Kunden“. Als nächstes folgt die Erweiterungsmethode `Include()`. Das ist eine Methode, die das Interface `IQueryable<T>` erweitert und vom Linq-Provider „Linq to entities“ bereitgestellt wird. Der Methode `Include()` übergeben wir den Lambda-Ausdruck

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

a => a.Anprechpartner. Damit legen wir fest, dass wir die Navigationproperty Anprechpartner mit laden wollen, beziehungsweise neue Anprechpartner in die Navigationproperty schreiben wollen. Dann verketten wir noch mit der Methode First(), der wir den Lambda-Ausdruck k => k.KundeId == kundeId übergeben. Damit selektieren wir nur den Kunden, der die der Methode CreateAnprechpartner() übergebene Id, hat. Jetzt haben wir ein Kunde-Objekt erstellt, das mit den passenden Daten aus der Datenbank gefüllt ist und darauf vorbereitet wurde, neue Anprechpartner entgegenzunehmen. Als nächstes rufen wir die Methode Add() der Naviagationproperty Anprechpartner und übergeben ihr das der Methode CreateAnprechpartner() übergebene Anprechpartner-Objekt. Zum Schluss schreiben wir mit kontext.SaveChanges() alle Änderungen in die Datenbank. Mit dem folgenden Testprogramm können wir die neue Methode überprüfen.

```
1  using DatenbankZugriff.CRUD;
2  using DatenbankZugriff.Modell;
3  using System;
4  using System.Collections.Generic;
5
6  namespace DatenbankZugriff
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             var ansprechpartner = new Anprechpartner
13             {
14                 Vorname = "Karl",
15                 Nachname = "Müller",
16                 Telefonnummer = "017258962354"
17             };
18
19             KundeCRUD kundeCRUD = new KundeCRUD();
20             kundeCRUD.CreateAnprechpartner(2, ansprechpartner);
21
22             Console.WriteLine($"ansprechpartner.
23                 AnprechpartnerId: {ansprechpartner.
24                 AnprechpartnerId}; ansprechpartner.
25                 KundId: {ansprechpartner.KundeId}; ansprechpartner.
26                 Nachname: {ansprechpartner.Nachname}");
27         }
28     }
29 }
```

Wir erstellen ein neues Anprechpartner-Objekt und weisen ihm mit einem Objektinitialisierer Daten zu. Als nächstes erstellen wir ein KundeCRUD-Objekt und rufen dessen Methode CreateAnprechpartner() auf und übergeben ihr den Wert 2 als Id für den Kunden und das vorher erzeugte Anprechpartner-Objekt. Zum Schluss geben wir die Properties AnprechpartnerId, KundId und Nachname, des Anprechpartner-Objekts am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
ansprechpartner.AnspprechpartnerId: 1; ansprechpartner.KundeId: 2; ansprechpartner.Nachname: Müller
C:\Users\rober\source\repos\DatenebankZugriff\DatenebankZugriff\bin\Debug\netcoreapp3.1\DatenebankZugriff.exe (Prozess "17348") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

Abb. 15.5.7 Ein Ansprechpartner in wird in der Datenbank angelegt

Beachten Sie, dass wir den Properties `AnsprechpartnerId` und `KundeId` des Ansprechpartner-Objekts keinen Wert zugewiesen haben. Trotzdem sehen wir für beide Properties Werte in der Bildschirmausgabe. Diese Werte wurden von der Datenbank und Entity-Framework automatisch zugewiesen.

Mit dem SQL-Server Management Studio können wir wieder den neu erzeugten Ansprechpartner in der Datenbank überprüfen.

The screenshot shows the SQL Server Management Studio interface. On the left, the Object Explorer tree view shows the database structure, including the 'Kunden' table under the 'Schiffele-NB' database. On the right, the results of a query are displayed in the 'SQLQuery2.sql - SC\_ELE-NB(rober(55))' window. The query is:

```

SELECT TOP (1000) [id]
      ,[vorname]
      ,[nachname]
      ,[telefon]
      ,[kundenid]
  FROM [Kunden].[dbo].[Ansprechpartner]

```

The results pane shows the following data:

	id	vorname	nachname	telefon	kundenid
1	1	Karl	Müller	017258962354	2

Abb. 15.5.8 Ein Ansprechpartner in der Datenbank

Der Entity-Framework ist sehr flexibel und oft gibt es mehrere Strategien, die zum Ziel führen. In der vorherigen Methode haben wir ein Kunde-Objekt geladen und mit der Methode `Include()` die von diesem Kunden abhängigen Ansprechpartner in die zugehörige Navigationproperty des Kunde-Objekts geladen. Dabei spielt es keine Rolle, ob in der Datenbank Ansprechpartner für diesen Kunden existieren oder nicht. Wenn es keine Ansprechpartner gibt, enthält die Navigationproperty `Ansprechpartner` eine Liste mit null Elementen. Wenn wir dagegen einen Kunden ohne Verkettung mit der Methode `Include()` laden, enthält die Navigationproperty immer den Wert `null`. Dann haben wir einen weiteren Ansprechpartner über die Navigationproperty hinzugefügt. Wir hätten auch mehrere Ansprechpartner hinzufügen können. Dabei haben wir in Bezug auf das Ansprechpartner-Objekt weder dem Primärschlüssel noch dem Fremdschlüssel einen Wert zugewiesen. Mit `kontext.SaveChanges()` haben wir alle Änderungen in die Datenbank gespeichert und der Entity-Framework hat die Fremdschlüssel zugewiesen. Der Primärschlüssel hat von der Datenbank automatisch einen Wert erhalten. Mit dem folgenden Beispiel betrachten wir eine Möglichkeit, einem Kunden einen Ansprechpartner hinzuzufügen, ohne vorher ein Kunde-

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

Objekt laden zu müssen. In diesem Beispiel implementieren wir eine Variante, die einem Kunden mehrere Ansprechpartner auf einmal hinzufügt.

```
1 public void CreateAnsprechpartner(int kundeId,
2 IEnumerable<Ansprechpartner> ansprechpartner)
3 {
4     foreach(var ap in ansprechpartner)
5     {
6         ap.KundeId = kundeId;
7         kontext.Anprechpartner.Add(ap);
8     }
9
10    kontext.SaveChanges();
11 }
```

Diese weitere Überladung der Methode `CreateAnsprechpartner()` nimmt neben der Id des Kunden, an die die Ansprechpartner angefügt werden sollen, noch eine Listenstruktur mit Ansprechpartnern als `IEnumerable<Ansprechpartner>` entgegen. Wir durchlaufen diese Liste mit einer `foreach`-Schleife und weisen für jeden Ansprechpartner der Property `KundeId` die übergebene Id des Kunden zu. Nach der Zuweisung übergeben wir den jeweiligen Ansprechpartner der Methode `Add()` des Ansprechpartner-DbSets des Kontexts. Nach der Schleife speichern wir alle übergebenen Ansprechpartner auf einmal mit der Anweisung `kontext.SaveChanges()`. Mit folgenden Programm können wir diese neue Überladung der Methode `CreateAnsprechpartner()` testen.

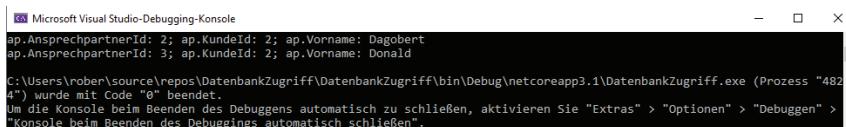
```
1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12
13             var ansprechpartner = new List<Ansprechpartner>
14             {
15                 new Ansprechpartner
16                 {
17                     Vorname = "Dagobert",
18                     Nachname = "Duck",
19                     Telefonnummer = "01715629936"
20                 },
21                 new Ansprechpartner
22                 {
23                     Vorname = "Donald",
24                     Nachname = "Duck",
25                     Telefonnummer = "01515298136"
26                 },
27             };
28 }
```

## 15.5 Lesen, ändern, hinzufügen und löschen von Daten

```

29     KundenCRUD kundeCRUD = new KundenCRUD();
30     kundeCRUD.CreateAnsprechpartner(1, ansprechpartner);
31
32     foreach (var ap in ansprechpartner)
33     {
34         Console.WriteLine($"ap.AnspprechpartnerId: {ap.
35             AnsprechpartnerId}; ap.KundeId: {ap.KundeId};
36             ap.Vorname: {ap.Vorname}");
37     }
38 }
39 }
40 }
```

Zuerst erzeugen wir eine Listenstruktur mit zwei Ansprechpartnern und weisen ihnen mit Hilfe eines Objektinitialisierers Daten zu. Danach erzeugen wir ein KundenCRUD-Objekt und rufen die Methode `CreateAnsprechpartner()` des KundenCRUD-Objekts auf. Dieser Methode übergeben wir die Id des Kunden, dem wir die beiden Ansprechpartner hinzufügen wollen, und die Listenstruktur mit den Ansprechpartnern. Danach durchlaufen wir die Listenstruktur der Ansprechpartner mit einer `foreach`-Schleife und geben für jeden Ansprechpartner die Properties `AnspprechpartnerId`, `KundeId` und `Vorname` am Bildschirm aus.



```

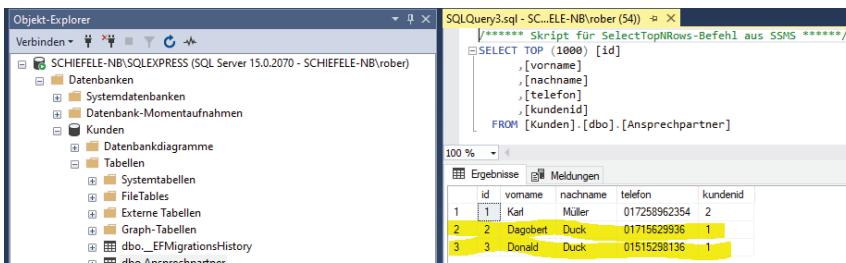
Microsoft Visual Studio-Debugging-Konsole
ap.AnspprechpartnerId: 2; ap.KundeId: 2; ap.Vorname: Dagobert
ap.AnspprechpartnerId: 3; ap.KundeId: 2; ap.Vorname: Donald

C:\Users\rober\source\repos\DatensbankZugriff\DatensbankZugriff\bin\Debug\netcoreapp3.1\DatensbankZugriff.exe (Prozess "4824") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konnekt beim Beenden des Debuggens automatisch schließen".

```

**Abb. 15.5.9** Zwei neue Ansprechpartner werden angelegt.

Beachten Sie: Auch hier vergeben wir für die Ansprechpartner keinen Wert in der Property `AnspprechpartnerId`, da dieser Wert von der Datenbank vergeben wird. Mit dem SQL-Management Studio können wir die beiden neu angelegten Ansprechpartner in der Datenbank überprüfen.



The screenshot shows the SSMS interface with two panes. The left pane, 'Object Explorer', shows the database structure for 'SCHIEFELE-NB\SQLEXPRESS'. The right pane, 'SQLQuery3.sql - SC-ELE-NB\rober (\$4)', contains a T-SQL script to select top 1000 rows from the 'Ansprechpartner' table. The results pane shows three rows of data:

	id	vorname	nachname	telefon	kundenid
1	1	Karl	Müller	017258962354	2
2	2	Dagobert	Duck	017156289356	1
3	3	Donald	Duck	01515298136	1

**Abb. 15.5.10** Die zwei neuen Ansprechpartner in der Datenbank

Als nächstes betrachten wir, wie wir Daten in der Datenbank ändern können. Dazu erweitern wir die Klasse `KundenCRUD` um eine weitere Methode.

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```

1 public void SaveChanges()
2 {
3     kontext.SaveChanges();
4 }
```

Die Methode `SaveChanges()` der Klasse `KundeCRUD` schleift lediglich die Methode `SaveChanges()` des Kontexts durch. Eine Datenbankänderung erfolgt, indem wir die nötigen `DbSets` des Kontexts verändern und danach die Methode `SaveChanges()` des Kontexts aufrufen. Mit dem folgenden Testprogramm können wir dieses Vorgehen demonstrieren.

```

1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12             KundeCRUD kundeCRUD = new KundeCRUD();
13
14             var kunde = kundeCRUD.LeseKunde(1);
15             kunde.Strasse = "Entenplatz 1";
16             kundeCRUD.SaveChanges();
17
18             Console.WriteLine($"kunde.KundeId: {kunde.KundeId};
19                         kunde.Firma: {kunde.Firma}; kunde.Strasse:
20                         {kunde.Strasse}");
21         }
22     }
23 }
```

Wir erzeugen ein `KundeCRUD`-Objekt und lesen mit Hilfe der Methode `LeseKunde()` den Kunden mit der Id 1 aus der Datenbank aus und speichern ihn in der Variablen `kunde`. Danach weisen wir der Property `Strasse` einen neuen Wert zu. Dann schreiben wir mit der Methode `SaveChanges()` die Änderungen in die Datenbank. Zum Schluss geben wir die Properties `KundeId`, `Firma` und `Strasse` des Objekts `kunde` am Bildschirm aus.

```

kunde.KundeId: 1; kunde.Firma: Duck Industries Inc.; kunde.Strasse: Entenplatz 1
C:\Users\rober\source\repos\DatenbankZugriff\DatenbankZugriff\bin\Debug\netcoreapp3.1\DatenbankZugriff.exe (Prozess "106
24") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

**Abb. 15.5.11** Ein Kunde wurde geändert.

Mit dem SQL-Management Studio können wir wieder die Änderung in unserer Datenbank überprüfen.

The screenshot shows the Object Explorer on the left with the database 'LAPTOP-5B692H2P\SQLEXPRESS' selected. In the center, a query window titled 'SQLQuery1.sql - LA...692H2P\rober (66)' displays the following script:

```
***** Skript für SelectTopNRows-Befehl aus SSMS *****/
SELECT TOP (1000) [id]
    , [firma]
    , [strasse]
    , [plz]
    , [ort]
FROM [Kunden].[dbo].[Kunde]
```

The results pane shows the following data:

	<b>id</b>	<b>firma</b>	<b>strasse</b>	<b>plz</b>	<b>ort</b>
1	1	Duck Industries Inc	Entenplatz 1	12345	Entenhausen
2	2	Kanzlei Müller & Söhne	Hauptstrasse 1	54321	Neustadt
3	3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen

Abb. 15.5.12 Der geänderte Kunde in der Datenbank

Als nächstes werden wir den Datenbankkontext etwas genauer betrachten. Eine Instanz eines Datenbankkontexts, der von der Klasse `DbContext` abgeleitet ist, kann keine gleichzeitigen Zugriffe auf eine Datenbank durchführen. Bei unseren Beispielprogrammen kommt es nicht zu mehreren Datenbankzugriffen, die gleichzeitig ablaufen, aber bei Programmen, die moderne Frameworks für grafische Benutzeroberflächen verwenden, kann dies durchaus vorkommen. Des Weiteren kommt es auch bei Programmen, die als Dienste auf einem Server laufen, zu gleichzeitigen Abläufen und damit sind auch gleichzeitige Datenzugriffe möglich. Deshalb sollte jede sequenziell ablaufende Kette von CRUD-Operationen ihre eigene Instanz des Datenkontexts bekommen. Deshalb ist es eine gute Idee, den Datenbankkontext in einem CRUD-Objekt zu kapseln und alle Datenbankoperationen in einem CRUD-Objekt zusammenzuhalten. Ein Datenbankzugriff ist ein unverwalteter Vorgang. Das heißt, wir müssen uns als Programmierer selbst um die Freigabe nicht mehr benötigter Ressourcen kümmern. Um die Ressourcen eines Kontexts wieder frei zu geben, müssen wir die Methode `Dispose()` aufrufen, die den Kontext von `DbContext` geerbt hat. Um das zu vereinfachen, gibt es im .Net-Framework das Interface `IDisposable`.

```
1 public interface IDisposable
2 {
3     void Dispose();
4 }
```

15

Das Interface verlangt lediglich die Implementierung der Methode `Dispose()`, welche aufgerufen wird, wenn der Garbage-Collector das Objekt aus dem Speicher entfernt. Unsere Klasse `KundeCRUD` muss daher `IDisposable` implementieren, damit wir sie in professionellen Projekten verwenden können.

Die Methode `Dispose()` für die Klasse `KundeCRUD` gestaltet sich sehr einfach:

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```
1 public void Dispose()
2 {
3     kontext.Dispose();
4 }
```

In der Methode `Dispose()` der Klasse `KundeCRUD` rufen wir lediglich die Methode `Dispose()` des Kontexts auf. Damit können wir Instanzen der Klasse `KundeCRUD` in einem `using`-Block erzeugen, so wie wir es beim Zugriff auf Dateien bereits kennengelernt haben.

```
1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12             using (var kundeCRUD = new KundeCRUD())
13             {
14                 var kunde = kundeCRUD.LeseKunde(1);
15                 kunde.Strasse = "Entenplatz 1";
16                 kundeCRUD.SaveChanges();
17
18                 Console.WriteLine($"kunde.KundeId: {kunde.
19                 KundeId}; kunde.Firma: {kunde.Firma}; kunde.
20                 Strasse: {kunde.Strasse}");
21             }
22         }
23     }
24 }
```

Die Datenbankoperation dieses Testprogramms besteht aus zwei sequenziell ausgeführten Teiloerationen: zuerst das Lesen einer Tabellenzeile und dann das Zurückschreiben in die Datenbank. Durch das Ausführen der Datenbankoperationen in einem `using`-Block stellen wir sicher, dass alle vom Datenbankkontext belegten Ressourcen wieder freigegeben werden, nachdem sie nicht mehr gebraucht werden.

Zum Schluss dieses Unterkapitels befassen wir uns mit dem Löschen in Datenbanken. Das Löschen funktioniert genauso wie das Ändern. Zuerst werden die gewünschten Daten aus den `DbSets` des Kontexts entfernt und dann werden die Änderungen mit der Methode `SaveChanges()` in die Datenbank geschrieben. Da Löschoperationen aber sensible Operationen sind, sollten wir zum Löschen eigene Methoden in einer `CRUD`-Klasse implementieren. Beim Löschen müssen wir auch die bereits angesprochene Referentielle Integrität beachten. Wenn wir zum Beispiel einen Kunden aus dem Kontext entfernen und dann `SaveChanges()` aufrufen, meldet uns die Datenbank sehr wahrscheinlich einen Fehler. Wir können nämlich keinen Kunden löschen,

so lange noch Ansprechpartner für diesen Kunden in der Datenbank existieren. Wenn wir also wirklich einen Kunden löschen wollen, müssen wir zuerst die Ansprechpartner des Kunden löschen. Eine Methode zum Löschen eines Kunden können wir wie folgt implementieren:

```

1 public void DeleteKundeInklusiveAnsprechpartner(int id)
2     {
3         var kunde = kontext.Kunden.Include(kd =>
4             kd.Ansprechpartner).First(kd => kd.KundeId == id);
5         if(kunde.Ansprechpartner != null && kunde.
6             Ansprechpartner.Any())
7         {
8             kunde.Ansprechpartner.Clear();
9             SaveChanges();
10        }
11
12        kontext.Kunden.Remove(kunde);
13        SaveChanges();
14    }

```

Die Methode `DeleteKundeInklusiveAnsprechpartner()` erwartet als Übergabeparameter die Id des zu löschen Kunden. Zuerst lesen wir den zu löschen Kunden aus der Datenbank ein, wobei wir mit der Methode `Include()` die Ansprechpartner des Kunden mit lesen. Die Methode `First()` selektiert den gewünschten Kunden mit einem Lambda-Ausdruck. Danach überprüfen wir, ob `Ansprechpartner` für den Kunden existieren. Falls es `Ansprechpartner` gibt, rufen wir die Methode `Clear()` der Property `Ansprechpartner` des `Kunde`-Objekts auf. Diese Methode entfernt alle Elemente aus der Listenstruktur `Ansprechpartner` und somit auch alle `Ansprechpartner` des Kunden. Mit einem Aufruf von `SaveChanges()` speichern wir die Änderungen in der Datenbank. Da der Kunde jetzt keine `Ansprechpartner` mehr besitzt, können wir den Kunden mit Hilfe der Methode `Remove()` aus dem `DbSet` Kunden des Kontexts entfernen. Ein weiterer Aufruf von `SaveChanges()` schreibt die Änderungen in die Datenbank. Mit dem folgenden Programm können wir die Methode `DeleteKundeInklusiveAnsprechpartner()` überprüfen.

```

1 using DatenbankZugriff.CRUD;
2 using DatenbankZugriff.Modell;
3 using System;
4 using System.Collections.Generic;
5
6 namespace DatenbankZugriff
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12             using (var kundeCRUD = new KundeCRUD())
13             {
14                 kundeCRUD.DeleteKundeInklusiveAnsprechpartner(1);
15             }
16         }
17     }
18 }

```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```

16         foreach (var kunde in kundeCRUD.LeseAlleKunden())
17     {
18         Console.WriteLine($"kunde.KundeId: {kunde.
19             KundeId}; kunde.Firma: {kunde.Firma}");
20     }
21 }
22 }
23 }
24 }
```

Zuerst erzeugen wir ein KundeCRUD-Objekt in einem using-Block. Dann rufen wir dessen Methode DeleteKundeInklusiveAnsprechpartner() auf und übergeben ihr die Id 1. Danach lesen wir alle Kunden aus der Datenbank und geben sie am Bildschirm aus.

```

Microsoft Visual Studio-Debugging-Konsole
kunde.KundeId: 2; kunde.Firma: Kanzlei Müller & Söhne
kunde.KundeId: 3; kunde.Firma: Bürobedarf Meier

C:\Users\rober\source\repos\DatenbankZugriff\DatenbankZugriff\bin\Debug\netcoreapp3.1\DatenbankZugriff.exe (Prozess "1872") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggens automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

```

**Abb. 15.5.13** Ein Kunde wird aus der Datenbank gelöscht.

Es werden nur die Kunden mit der Id 2 und 3 ausgegeben, da der Kunde mit der Id 1 gelöscht wurde.

Mit dem SQL-Management Studio können wir die Tabellen Kunden und Ansprechpartner überprüfen.

SQLQuery1.sql - LA...692H2P\rober(61) # X

```

/***** Skript für SelectTopNRows-Befehl aus SSMS *****/
SELECT TOP (1000) [id]
      ,[firma]
      ,[strasse]
      ,[plz]
      ,[ort]
  FROM [Kunden].[dbo].[Kunde]
```

	id	firma	strasse	plz	ort
1	2	Kanzlei Müller & Söhne	Hauptstrasse 1	54321	Neustadt
2	3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen

**Abb. 15.5.14** Kunde 1 existiert nicht mehr in der Datenbank.

## 15.6 Übungsaufgaben: Programmierung mit Datenbanken

```

SELECT TOP (1000) [id]
    , [vorname]
    , [nachname]
    , [telefon]
    , [kundenid]
  FROM [Kunden].[dbo].[Ansprechpartner]
  
```

	id	vorname	nachname	telefon	kundenid
1	1	Karl	Müller	0172-58962354	2

Abb. 15.5.15 Für Kunde 1 existieren keine Ansprechpartner in der Datenbank.

## 15.6 Übungsaufgaben: Programmierung mit Datenbanken

In dieser Übung erweitern wir unsere Kunden-Datenbank schrittweise um die Tabellen „Produkte“ und „Kaeufe“ und erstellen die zugehörigen Entity-Klassen. Außerdem erstellten wir eine ProduktCRUD-Klasse zur Pflege von Produkten und erweitern die KundeCRUD-Klasse um ein paar weitere Methoden.

### Teilaufgabe 1:

Erstellen Sie die Entity-Klasse `Produkt`. Die Klasse benötigt geeignete Properties für einen Primärschlüssel, der automatisch von der Datenbank vergeben wird, sowie für eine Produktbezeichnung. Die zugehörige Tabelle in der Datenbank soll `Produkte` heißen. Die Spalte für den Primärschlüssel in der Datenbank soll `id` heißen und die Spalte für die Bezeichnung soll `bezeichnung` heißen. Die Tabellenspalte `bezeichnung` darf keine Null-Werte enthalten. Die maximale Länge für die Bezeichnung beträgt 100 Zeichen. Erweitern Sie den Datenbankkontext `KundeKontext` um ein `DbSet` für die Tabelle `Produkte`. Erstellen Sie eine Migration und übertragen Sie die Änderungen zur Datenbank.

### Teilaufgabe 2:

Erstellen Sie die Entity-Klasse `Kauf` mit Properties für einen Primärschlüssel, der automatisch von der Datenbank vergeben wird, einen Fremdschlüssel, der auf die Tabelle `Kunde` zeigt, einen Fremdschlüssel, der auf die Tabelle `Produkte` zeigt, für die Anzahl der Produkte in einem Kauf und für das Kaufdatum. Die Anzahl darf nur ganze Zahlen und keine Nullwerte enthalten, das Kaufdatum darf keine Nullwerte enthalten. Die zugehörige Datenbanktabelle soll `Kaeufe` heißen. Die Tabellenspalte für den Primärschlüssel soll `id` heißen. Die Tabellenspalten für die Fremdschlüssel sollen `kundeid` und `produktid` heißen. Die Tabellenspalten für die Anzahl und das Kauf-

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

datum sollen anzahl und kaufdatum heißen. Erweitern Sie zusätzlich die Klasse KundeKontext um das DbSet Kaeufe. Erstellen sie zwei Navigationproperties in der Klasse Kauf für den zugehörigen Kunden und das zugehörige Produkt. Erweitern Sie die Klasse Kunde um eine Navigationproperty für alle Käufe eines Kunden. Erweitern sie die Klasse Produkt um eine Navigationproperty für alle Käufe, in denen dieses Produkt gekauft wurde. Beachten Sie dabei, dass alle Fremdschlüssel auch in der Datenbank gesetzt sein sollen. Erstellen Sie eine Migration und übertragen Sie die Änderungen zur Datenbank.

### Teilaufgabe 3:

Erstellen Sie - analog zur Klasse KundeCRUD - die Klasse ProduktCRUD, die zur Pflege der Tabelle Produkte dienen soll. Die Klasse ProduktCRUD soll folgende Methoden enthalten:

```
1 public void Create(Produkt produkt)
```

Diese Methode erwartet einen Übergabeparameter vom Typ Produkt und legt für diesen eine Zeile in der Tabelle Produkte an.

```
1 public void Create(IEnumerable<Produkt> produkte)
```

Diese Methode erwartet einen Übergabeparameter vom Typ IEnumerable<Produkt> und legt für jedes Element in der Auflistung eine Zeile in der Tabelle Produkte an.

```
1 public IQueryable<Produkt> LeseAlleProdukte()
```

Diese Methode gibt alle Produkte der Tabelle Produkte als IQueryable<Produkt> zurück.

```
1 public Produkt LeseProdukt(int id)
```

Diese Methode erwartet die Id eines Produkts als Übergabeparameter und gibt das Produkt mit dieser Id zurück.

```
1 public void SaveChanges()
```

Diese Methode ruft die Methode SaveChanges () des Kontexts auf.

```
1 public void DeleteProduktInklusiveKaeufe(int id)
```

Diese Methode erwartet die Id eines Produkts als Übergabeparameter und löscht das Produkt mit dieser Id. Beachten Sie dabei, dass Sie ein Produkt nur dann löschen können, wenn es in der Tabelle Kaeufe nicht mehr vorkommt.

**Teilaufgabe 4:**

Erweitern Sie die Klasse `KundeCRUD` um folgende Methoden:

```
1 public void CreateKaeufe(int kundeId, IEnumerable<Kauf> kaeufe)
```

Die Methode nimmt eine Id eines Kunden und mehrere Käufe für diesen Kunden vom Typ `IEnumerable<Kauf>` entgegen und legt für jedes Element im Übergabeparameter `kaeufe` eine Zeile in der Tabelle `Kaeufe` an.

```
1 public IQueryable<Kauf> LeseAlleKaeufe()
```

Diese Methode gibt alle Käufe der Tabelle `Kaeufe` als `IQueryable<Kauf>` zurück.

```
1 public void DeleteKauf(int id)
```

Diese Methode nimmt eine Id eines Kaufes entgegen und löscht die dazugehörige Zeile in der Tabelle `Kaeufe`:

```
1 public void DeleteAlleKaeufeEinesKunden(int id)
```

Diese Methode nimmt die Id eines Kunden entgegen und löscht alle Käufe dieses Kunden.

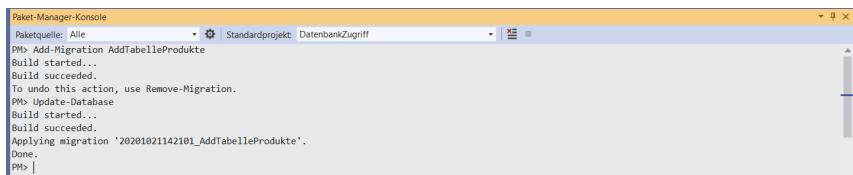
Schreiben Sie ein Hauptprogramm, um das erweiterte Entity-Modell zu testen. Dieses Hauptprogramm soll folgenden Ablauf implementieren:

1. Anlegen von 3 Produkten
2. Anlegen von je einem Kauf für jedes in Punkt 1 angelegte Produkt pro Kunde
3. Löschen von allen Käufen des Kunden von Punkt 2
4. Anlegen von einem Kauf für den Kunden von Punkt 2
5. Löschen von einem Produkt
6. Ausgeben aller Produkte am Bildschirm
7. Ausgeben aller Käufe am Bildschirm

**Musterlösung für Teilaufgabe 1:**

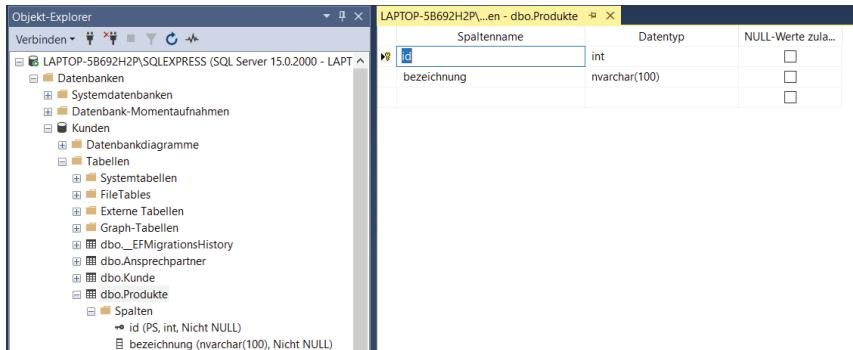
```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9      [Table("Produkte")]
10     public class Produkt
11     {
12         [Key]
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         [Column("id")]
15         public int ProduktId { get; set; }
16
17         [Required]
18         [MaxLength(100)]
19         [Column("bezeichnung")]
20         public string Bezeichnung { get; set; }
21     }
22 }
23
24 using Microsoft.EntityFrameworkCore;
25 using System;
26 using System.Collections.Generic;
27 using System.Text;
28
29 namespace DatenbankZugriff.Modell
30 {
31     public class KundenKontext : DbContext
32     {
33         public DbSet<Kunde> Kunden { get; set; }
34         public DbSet<Ansprechpartner> Ansprechpartner { get;
35             set; }
36         public DbSet<Produkt> Produkte { get; set; }
37
38         protected override void OnConfiguring
39         (DbContextOptionsBuilder optionsBuilder)
40         {
41             optionsBuilder.UseSqlServer(@"Server=.\SQLEXPRESS;
42             Database=Kunden;Trusted_Connection=True;");
43         }
44     }
45 }
```

## 15.6 Übungsaufgaben: Programmierung mit Datenbanken



```
Paket-Manager-Konsole
Paketquelle: Alle Standardprojekt: DatenBankZugriff
PM> Add-Migration AddTabelleProdukte
Build started..
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started..
Build succeeded.
Applying migration '20201021142101_AddTabelleProdukte'.
Done.
PM>
```

Abb. 15.6.1 Migration für die Tabelle Produkte



The screenshot shows the Object Explorer and the Table Designer side-by-side.

**Object Explorer:** Shows the database structure. Under 'LAPTOP-5B692H2P\...en - dbo' (SQL Server 15.0.2000), there are several objects: Datenbanken, Systemdatenbanken, Datenbank-Momentaufnahmen, Kunden, Datenbankdiagramme, Tabelles, Systemtabellen, FileTables, Externe Tabellen, Graph-Tabellen, and the 'Produkte' table. The 'Produkte' table is selected in the Table Designer.

**Table Designer:** Displays the structure of the 'Produkte' table. It has two columns: 'id' (int, primary key) and 'bezeichnung' (nvarchar(100)).

Spaltenname	Datentyp	NULL-Werte zula...
id	int	<input type="checkbox"/>
bezeichnung	nvarchar(100)	<input type="checkbox"/>

Abb. 15.6.2 Die Tabelle Produkte in der Datenbank

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9      [Table("Kaeufe")]
10     public class Kauf
11     {
12         [Key]
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         [Column("id")]
15         public int KaufId { get; set; }
16
17         [ForeignKey("Kunde")]
18         [Column("kundeid")]
19         public int KundeId { get; set; }
20
21         [ForeignKey("Produkt")]
22         [Column("produktid")]
23         public int ProduktId { get; set; }
24
25         [Required]
26         [Column("anzahl")]
27         public int Anzahl { get; set; }
28
29         [Required]
30         [Column("kaufdatum")]
31         public DateTime Kaufdatum { get; set; }
32
33         //Navigation Properties
34         public virtual Kunde Kunde { get; set; }
35         public virtual Produkt Produkt { get; set; }
36     }
37 }
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9      [Table("Kunde")]
10     public class Kunde
11     {
12         [Key]
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         [Column("id")]
15         public int KundeId { get; set; }
```

## 15.6 Übungsaufgaben: Programmierung mit Datenbanken

```

16     [Required]
17     [MaxLength(50)]
18     [Column("firma")]
19     public string Firma { get; set; }
20
21
22     [MaxLength(50)]
23     [Column("strasse")]
24     public string Strasse { get; set; }
25
26     [MaxLength(5)]
27     [Column("plz")]
28     public string PLZ { get; set; }
29
30     [MaxLength(50)]
31     [Column("ort")]
32     public string Ort { get; set; }
33
34     //Navigation Properties
35     public virtual ICollection<Ansprechpartner>
36     Ansprechpartner { get; set; }
37     public virtual ICollection<Kauf> Kaeufe { get; set; }
38   }
39 }
```

```

1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Text;
6
7  namespace DatenbankZugriff.Modell
8  {
9    [Table("Produkte")]
10   public class Produkt
11   {
12     [Key]
13     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14     [Column("id")]
15     public int ProduktId { get; set; }
16
17     [Required]
18     [MaxLength(100)]
19     [Column("bezeichnung")]
20     public string Bezeichnung { get; set; }
21
22     //Navigation Properties
23     public virtual ICollection<Kauf> Kaeufe { get; set; }
24   }
25 }
```

```

1  using Microsoft.EntityFrameworkCore;
2  using System;
3  using System.Collections.Generic;
4  using System.Text;
```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```

5
6 namespace DatenbankZugriff.Modell
7 {
8     public class KundenKontext : DbContext
9     {
10         public DbSet<Kunde> Kunden { get; set; }
11         public DbSet<Ansprechpartner> Ansprechpartner { get;
12             set; }
13         public DbSet<Produkt> Produkte { get; set; }
14         public DbSet<Kauf> Kaeufe { get; set; }
15
16         protected override void OnConfiguring
17             (DbContextOptionsBuilder optionsBuilder)
18         {
19             optionsBuilder.UseSqlServer(@"Server=.\ SQLEXPRESS;
20             Database=Kunden;Trusted_Connection=True;");
21         }
22     }
23 }
```

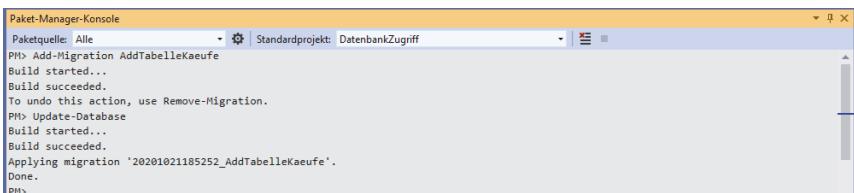


Abb. 15.6.3 Migration für die Tabelle Kaeufe

The screenshot shows the 'Objekt-Explorer' (Object Explorer) and a table definition for the 'Kaeufe' table in the 'LAPTOP-5B692H2P\...nden - dbo.Kaeufe' database.

**Object Explorer:**

- Connected to: LAPTOP-5B692H2P\SQLEXPRESS (SQL Server 15.0.2000 - LAPTOP-5B692H2P)
- Databases:
  - Datenbanken
  - Systemdatenbanken
  - Datenbank-Momentaufnahmen
- Kunden:
  - Datenbankdiagramme
  - Tabellen:
    - Systemtabellen
    - FileTables
    - Externe Tabellen
    - Graph-Tabellen
    - dbo.\_EFMigrationsHistory
    - dbo.Ansprechpartner
    - dbo.Kaeufe
  - Spalten

**Table Definition:**

Spaltenname	Datentyp	NULL-Werte...
<b>id</b>	int	<input type="checkbox"/>
kundeid	int	<input type="checkbox"/>
produktid	int	<input type="checkbox"/>
anzahl	int	<input type="checkbox"/>
kaufdatum	datetime2(7)	<input type="checkbox"/>

Abb. 15.6.4 Die Tabelle Kaeufe in der Datenbank

**Musterlösung für Teilaufgabe 3:**

```
1  using DatenbankZugriff.Modell;
2  using Microsoft.EntityFrameworkCore;
3  using System;
4  using System.Collections.Generic;
5  using System.Linq;
6  using System.Text;
7
8  namespace DatenbankZugriff.CRUD
9  {
10     public class ProduktCRUD : IDisposable
11     {
12         private KundenKontext kontext;
13
14         public ProduktCRUD()
15         {
16             kontext = new KundenKontext();
17         }
18
19         public void Create(Produkt produkt)
20         {
21             kontext.Produkte.Add(produkt);
22             kontext.SaveChanges();
23         }
24
25         public void Create(IEnumerable<Produkt> produkte)
26         {
27             kontext.Produkte.AddRange(produkte);
28             kontext.SaveChanges();
29         }
30
31         public IQueryable<Produkt> LeseAlleKunden()
32         {
33             return kontext.Produkte;
34         }
35
36         public Produkt LeseProdukt(int id)
37         {
38             return LeseAlleKunden().First(k => k.ProduktId ==
39             id);
40         }
41
42         public void SaveChanges()
43         {
44             kontext.SaveChanges();
45         }
46
47         public void DeleteProduktInklusiveKaeufe(int id)
48         {
49             var produkt = kontext.Produkte.Include(p =>
50                 p.Kaeufe).First(p => p.ProduktId == id);
51             if (produkt.Kaeufe != null && produkt.Kaeufe.Any())
52             {
53                 produkt.Kaeufe.Clear();
54                 SaveChanges();
55             }
56         }
57     }
58 }
```

**15 Datenbankzugriff mit dem Microsoft SQL-Server**

```
55     }
56
57     kontext.Produkte.Remove(produkt);
58     SaveChanges();
59
60 }
61
62 public void Dispose()
63 {
64     kontext.Dispose();
65 }
66 }
67 }
```

**Musterlösung für Teilaufgabe 4:**

```
1  using DatenbankZugriff.Modell;
2  using Microsoft.EntityFrameworkCore;
3  using Microsoft.EntityFrameworkCore.Metadata.Conventions;
4  using System;
5  using System.Collections.Generic;
6  using System.Linq;
7  using System.Text;
8
9  namespace DatenbankZugriff.CRUD
10 {
11     public class KundeCRUD : IDisposable
12     {
13         private KundenKontext kontext;
14
15         public KundeCRUD()
16         {
17             kontext = new KundenKontext();
18         }
19
20         public void Create(Kunde kunde)
21         {
22             kontext.Kunden.Add(kunde);
23             kontext.SaveChanges();
24         }
25
26         public void Create(IEnumerable<Kunde> kunden)
27         {
28             kontext.Kunden.AddRange(kunden);
29             kontext.SaveChanges();
30         }
31
32         public IQueryable<Kunde> LeseAlleKunden()
33         {
34             return kontext.Kunden;
35         }
36
37         public Kunde LeseKunde(int id)
38         {
39             return LeseAlleKunden().First(k => k.KundeId == id);
40         }
41
42         public void CreateAnsprechpartner(int kundeId,
43             Ansprechpartner ansprechpartner)
44         {
45             var kunde = kontext.Kunden
46                 .Include(a => a.Ansprechpartner)
47                 .First(k => k.KundeId == kundeId);
48
49             kunde.Ansprechpartner.Add(ansprechpartner);
50             kontext.SaveChanges();
51         }
52
53         public void CreateAnsprechpartner(int kundeId,
54             IEnumerable<Ansprechpartner> ansprechpartner)
```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```
55     {
56         foreach(var ap in ansprechpartner)
57         {
58             ap.KundeId = kundeId;
59             kontext.Anprechpartner.Add(ap);
60         }
61
62         kontext.SaveChanges();
63     }
64
65     public void SaveChanges()
66     {
67         kontext.SaveChanges();
68     }
69
70     public void DeleteKundeInklusiveAnsprechpartner(int id)
71     {
72         var kunde = kontext.Kunden.Include(kd =>
73             kd.Anprechpartner).First(kd => kd.KundeId == id);
74         if(kunde.Anprechpartner != null && kunde.
75             Anprechpartner.Any())
76         {
77             kunde.Anprechpartner.Clear();
78             SaveChanges();
79         }
80
81         kontext.Kunden.Remove(kunde);
82         SaveChanges();
83     }
84
85     public void CreateKaeufe(int kundeId, IEnumerable<Kauf>
86     kaeufe)
87     {
88         var kunden = kontext.Kunden.Include(k => k.Kaeufe);
89         var debug = kunden.ToList();
90         var kunde = kunden.First(k => k.KundeId == kundeId);
91         if(kunde.Kaeufe != null)
92         {
93             foreach (var kauf in kaeufe)
94             {
95                 kunde.Kaeufe.Add(kauf);
96             }
97
98             kontext.SaveChanges();
99         }
100    }
101
102    public IQueryable<Kauf> LeseAlleKaeufe()
103    {
104        return kontext.Kaeufe;
105    }
106
107    public void DeleteKauf(int id)
108    {
109        var kauf = kontext.Kaeufe.First(k => k.KaufId ==
110            id);
```

## 15.6 Übungsaufgaben: Programmierung mit Datenbanken

```

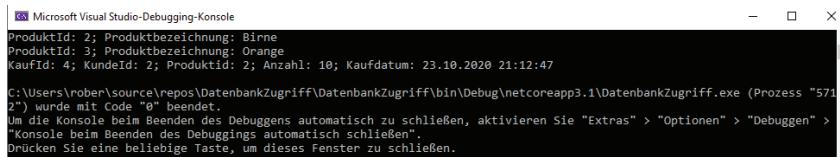
111         kontext.Kaeufe.Remove(kauf);
112         kontext.SaveChanges();
113     }
114
115     public void DeleteAlleKaeufeEinesKunden(int id)
116     {
117         var kunde = kontext.Kunden.Include(k => k.Kaeufe).
118             First(k => k.KundeId == id);
119         if(kunde.Kaeufe != null)
120         {
121             kunde.Kaeufe.Clear();
122             kontext.SaveChanges();
123         }
124     }
125
126     public void Dispose()
127     {
128         kontext.Dispose();
129     }
130 }
131 }
```

```

1  using DatenbankZugriff.CRUD;
2  using DatenbankZugriff.Modell;
3  using System;
4  using System.Collections.Generic;
5
6  namespace DatenbankZugriff
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             using (var produktCRUD = new ProduktCRUD())
13             {
14                 using (var kundeCRUD = new KundeCRUD())
15                 {
16                     var produkte = new List<Produkt>
17                     {
18                         new Produkt
19                         {
20                             Bezeichnung = "Apfel"
21                         },
22                         new Produkt
23                         {
24                             Bezeichnung = "Birne"
25                         },
26                         new Produkt
27                         {
28                             Bezeichnung = "Orange"
29                         }
30                     };
31                     produktCRUD.Create(produkte);
32
33                     var kaeufe = new List<Kauf>();
34                     foreach(var produkt in produkte)
```

## 15 Datenbankzugriff mit dem Microsoft SQL-Server

```
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79 }
```



The screenshot shows the Microsoft Visual Studio Debugging Console window. The output text is as follows:

```
Microsoft Visual Studio-Debugging-Konsole
ProduktId: 2; Produktbezeichnung: Birne
ProduktId: 3; Produktbezeichnung: Orange
KaufId: 4; KundeId: 2; ProduktId: 2; Anzahl: 10; Kaufdatum: 23.10.2020 21:12:47

C:\Users\rober\source\repos\DatenbankZugriff\bin\Debug\netcoreapp3.1\DatenbankZugriff.exe (Prozess "5712") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggings automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Abb. 15.6.5 Die Bildschirmausgabe der Musterlösung

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 16

# ASP.NET MVC – Anwendungen fürs Web

Bisher haben wir nur zwei Projekttypen betrachtet: Die Klassenbibliothek zum Erstellen von Klassen, die in anderen Projekten wiederverwendet werden können, und die Konsolen-App, mit der wir Benutzereingaben und Bildschirmausgaben nur in einem Windows-Command-Fenster tätigen können. In diesem Kapitel betrachten wir den Projekttyp ASP.NET Core-Webanwendung. Mit diesem Projekttyp können wir Anwendungen entwickeln, die in einem Internetbrowser wie Microsoft Edge oder Google Chrome laufen und auf einem Webserver gehostet werden können.

Die Entwicklung einer professionellen Webanwendung mit Visual Studio kann mitunter sehr komplex werden. Und neben Kenntnissen in C# benötigt man noch Kenntnisse in HTML, CSS und JavaScript. Da die Vermittlung von detaillierten Kenntnissen in diesen Sprachen den Rahmen dieses Buches bei weitem sprengen würden, schreiben wir Code in HTML, CSS und JavaScript so wenig wie möglich und erklären den generierten Code in diesen Sprachen auch nur soweit es zum Verständnis dieses Kapitels nötig ist. Wer professionell in die Programmierung von Webanwendungen mit Visual Studio einsteigen möchte, sollte sich - neben C# - unbedingt mit HTML, CSS und JavaScript beschäftigen.

In den folgenden Unterkapiteln werden wir exemplarisch eine ASP.NET MVC-Webanwendung erstellen, die mit den Daten aus der im Kapitel 15 erstellten SQL-Server Datenbank arbeitet.

### 16.1 Eine neue ASP.NET MVC-Anwendung erstellen

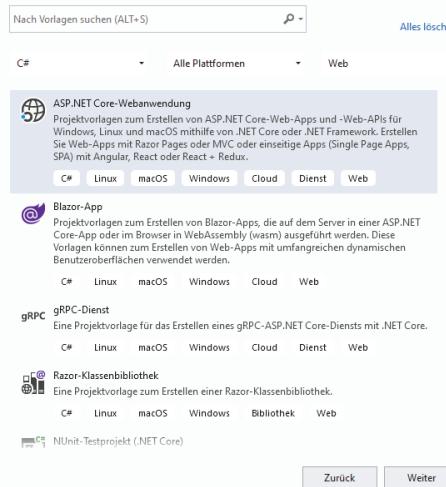
Um eine ASP.NET Core-Webanwendung zu erstellen, starten wir Visual Studio und klicken auf „Neues Projekt erstellen“.

## 16.1 Eine neue ASP.NET MVC-Anwendung erstellen

## Neues Projekt erstellen

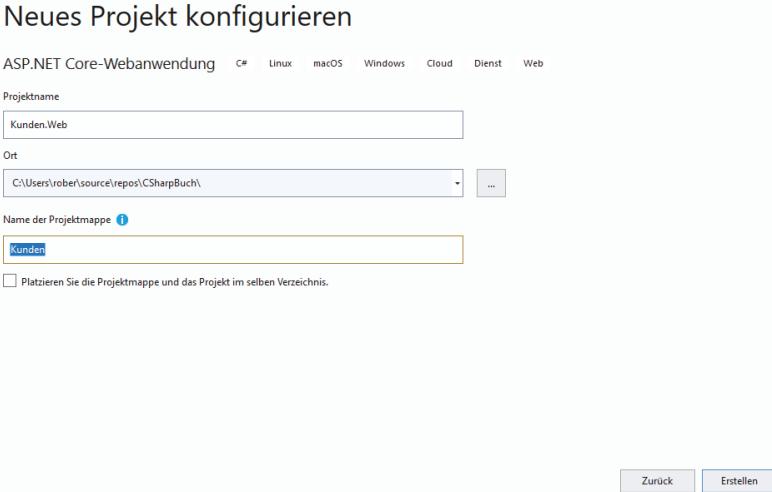
Zuletzt verwendete Projektvorlagen

Eine Liste der zuletzt verwendeten Vorlagen wird hier angezeigt.



**Abb. 16.1.1** Eine ASP.NET Core-Webanwendung erstellen

Im Dialog „Neues Projekt erstellen“ wählen wir für den Sprachfilter C#, für den Plattformfilter alle Plattformen und für den Projekttypfilter Web aus. Dann wählen wir die ASP.NET Core-Webanwendung aus und klicken auf die Schaltfläche „Weiter“.



**Abb. 16.1.2** Eine ASP.NET Core-Webanwendung konfigurieren

Im Dialog „Neues Projekt konfigurieren“ vergeben wir als Projektname Kunden.Web und für die Projektmappe vergeben wir den Namen Kunden. Danach klicken wir auf die Schaltfläche „Erstellen“.

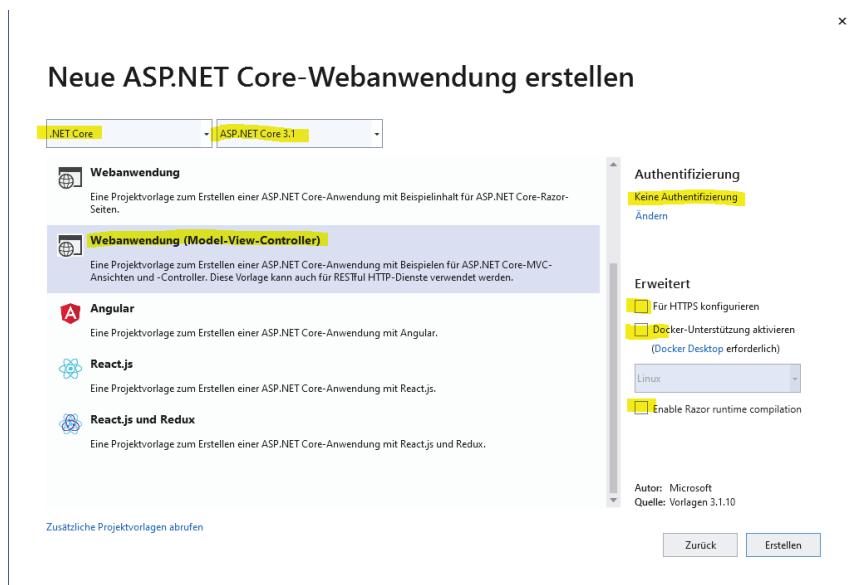
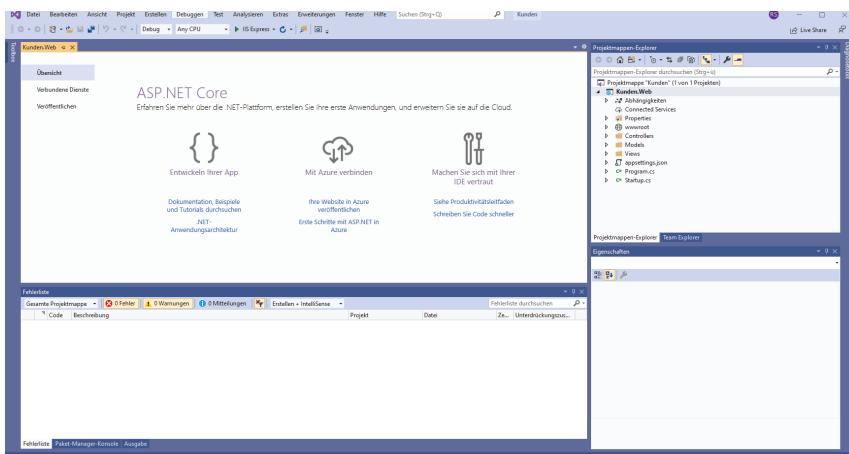


Abb. 16.1.3 Der Dialog Neue ASP.NET Core-Webanwendung erstellen

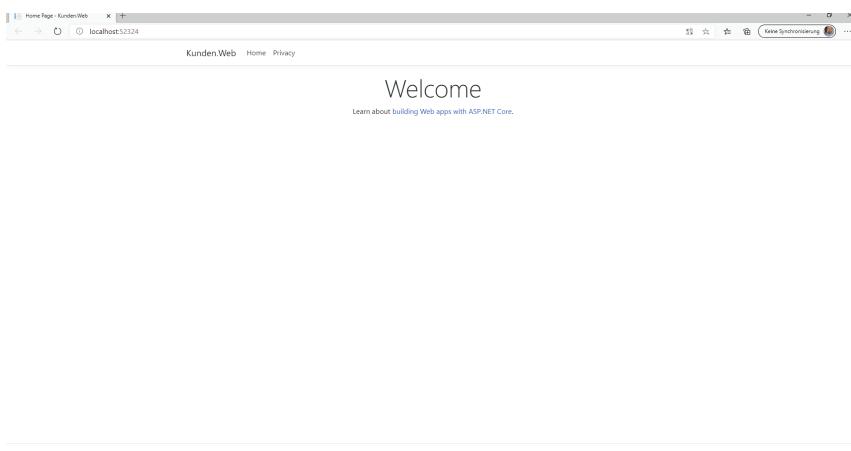
Im Dialog „Neue ASP.NET Core-Webanwendung erstellen“, wählen wir in der linken der oberen Auswahllisten .NET Core aus und in der rechten Auswahlliste wählen wir ASP.NET Core 3.1 aus. Des Weiteren markieren wir „Webanwendung (Model-View-Controller)“. Die Einstellung für die Authentifizierung belassen wir auf keine Authentifizierung. Und in der Checkbox „Für HTTPS konfigurieren“ entfernen wir den Haken. Die Docker-Unterstützung und Razor Runtime compilation aktivieren wir ebenfalls nicht. Dadurch sorgen wir dafür, dass keine Features aktiviert werden, die den Rahmen dieses Buches sprengen würden. Dann klicken wir auf die Schaltfläche „Erstellen“.

## 16 ASP.NET MVC – Anwendungen fürs Web



**Abb. 16.1.4** Eine neue ASP.NET Core-Webanwendung

Nachdem Visual Studio für uns die Webanwendung erstellt hat, können wir - wie gewohnt - die Anwendung mit einem Klick auf das grüne Dreieck starten.



**Abb. 16.1.5** Eine neue ASP.NET Core-Anwendung im Browser

Beim Starten der Webanwendung öffnet Visual Studio einen Internetbrowser. Da wir die Voreinstellung für die Broweresauswahl nicht verändert haben, wird Microsoft Edge gestartet. Microsoft Edge öffnet die Internetadresse localhost:52324. Das ist ein Webserver, der auf dem Computer, auf dem die Webanwendung entwickelt wird, läuft. Die Installation von Visual Studio sorgt dafür, dass dieser Webserver verfügbar ist. Die Zahl

nach dem Doppelpunkt ist die sogenannte Portnummer oder kurz der Port. Die Portnummer wird von Visual Studio vergeben und kann durchaus eine andere Nummer sein, wenn Sie dieses Beispiel auf Ihrem Computer ausprobieren. Jeder Webserver verwendet eine Portnummer. Damit wird ermöglicht, dass mehrere Webserver auf einem Computer gleichzeitig laufen können. Jeder mit seinem eigenen Port. Im Folgenden werden Internetadressen immer ohne Portnummer angegeben. Wenn Sie die Beispiele des Buches nachvollziehen, müssen Sie bei den Internetadressen immer die Portnummer ergänzen, die von Ihrer Umgebung verwendet wird. Falls sie sich jetzt wundern, dass keine Portnummer angegeben werden muss, wenn Sie eine Seite im Internet wie zum Beispiel [www.bmu-verlag.de](http://www.bmu-verlag.de) besuchen, liegt das daran, dass Webserver im Internet üblicherweise den Port 80 verwenden und dass ein Webbrower immer den Port 80 verwendet, wenn kein Port angegeben wird. Die Webanwendung, die von Visual Studio erzeugt wurde, hat am oberen Rand ein Menü mit zwei Menüpunkten: Home und Privacy. Bei einem Klick auf Home wird eine Welcome-Seite angezeigt und bei einem Klick auf Privacy wird eine Privacy-Seite angezeigt. Die Abbildung 16.1-5 zeigt die Welcome-Seite, welche auch beim Start der Webanwendung angezeigt wird. Die folgende Abbildung zeigt die Privacy-Seite:

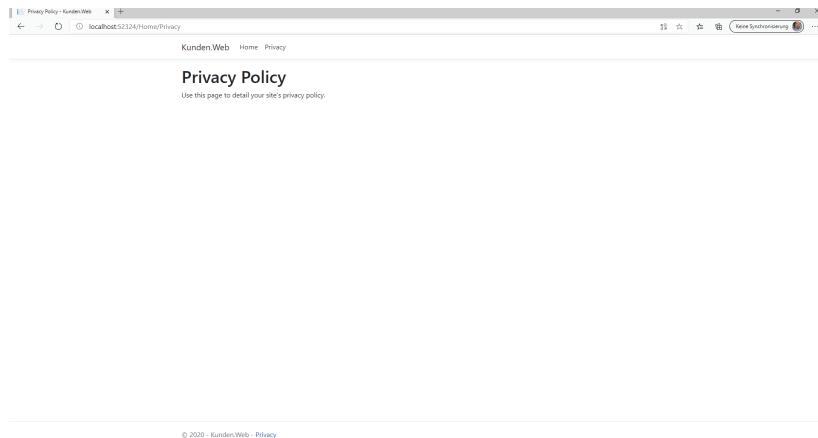


Abb. 16.1.6 Die Privacy-Seite der Webanwendung im Browser

In dem hier verwendeten Projektyp wird das sogenannte MVC-Entwurfsmuster verwendet. MVC steht für Model View Controller. Das Paradigma dieses Entwurfsmusters fordert die Einteilung des Programmcodes in eine Model-Schicht, eine View-Schicht und eine Controller-Schicht. Die Model-Schicht enthält Klassen für die Repräsentation der Daten. Die Klassen der Modelschicht befinden sich im Projektverzeichnis Model. Die View-Schicht enthält Klassen für die Darstellung der Daten am Bildschirm. Bei einer Visual Studio-Webanwendung besteht die View-Schicht aus HTML-Templates, die mit C#-Syntax angereichert sind. Die Dateien, die diese Templates enthalten, ha-

ben die Endung cshtml und befinden sich im Projektverzeichnis „Views“. Die Controller-Schicht enthält bei Visual Studio MVC-Projekten standardisierte Klassen, die die Modelklassen instanzieren und an die Templates der View-Schicht weiterleiten.

Der Einstiegspunkt, um die Architektur von Asp.NET MVC-Anwendungen zu verstehen, ist das sogenannte URL-Routing. Jede Webanwendung wird durch eine sogenannte URL aufgerufen. Zum Beispiel: <https://bmu-verlag.de>. Das ist die Startseite des Internetauftritts des BMU-Verlags. Man muss eine Webanwendung aber nicht zwingend über die Startseite aufrufen. Die untergeordnete URL <https://bmu-verlag.de/products> führt direkt zur Produktseite des BMU-Verlags. Nachdem Servernamen folgt bei einer untergeordneten URL das Zeichen „/“ und dann ein weiterer Name in obigem Beispiel „products“. Nach dem Servernamen können auch mehrere, durch „/“ getrennte Namen folgen. Zum Beispiel führt die URL <https://bmu-verlag.de/books/cs-kompendium> zur Produktseite dieses Buchs.

Das URL-Routing legt fest, wie eine URL einer Methode einer Controllerklasse zugeordnet wird. Das URL-Routing kann vom Programmierer beeinflusst werden. In einer ASP.NET-Webanwendung finden Sie das URL-Routing in der Klasse Startup.cs im Wurzelverzeichnis des Projekts. Am Ende der Klasse steht der Block:

```
1 app.UseEndpoints(endpoints =>
2 {
3     endpoints.MapControllerRoute(
4         name: "default",
5         pattern: "{controller=Home}/{action=Index}/{id?}");
6 }
```

Nach dem Erstellen eines ASP.NET MVC-Projekts ist genau eine Route definiert. Die Methode `MapControllerRoute()` erwartet als ersten Übergabeparameter den Namen der Route, in diesem Beispiel „default“, und als zweiten Parameter einen String, der ein Muster zur Steuerung des URL-Routings enthält. Das obige Muster definiert ein Routing, bei dem maximal drei mit „/“ getrennte Namen dem Servernamen folgen dürfen. Der erste Name wird durch das Muster `{controller=Home}` repräsentiert. Der erste Name ist der Name des Controllers, in dem eine Methode zum Erzeugen der gewünschten HTML-Seite gesucht wird. Der Standardwert für den Namen des Controllers ist Home. Der zweite Name wird durch das Muster `{action=Index}` dargestellt. Dadurch wird der zweite Name als Namen der Methode der Controllerklasse interpretiert, die die gewünschte HTML-Seite erzeugen soll. Diese Methode wird bei ASP.NET MVC auch als Action bezeichnet. Der Standardwert für die Action ist Index. Der dritte Name wird durch das Muster `{id?}` dargestellt. Dadurch kann als dritter Name ein beliebiger Wert verwendet werden. Das Fragezeichen bedeutet, dass dieser Wert optional ist. Wird dieser Wert angegeben, so sucht ASP.NET MVC im Controller eine Überladung der Action, die einen Übergabeparameter entgegennimmt und versucht, dieser Methode den Wert zu übergeben. Dabei wird eine Typkonvertierung durchgeführt. Hat der Übergabeparameter zum Beispiel den Typ `int` und der dritte

Name ist 42, dann wird die Konvertierung erfolgreich sein. Ist der dritte Name dagegen ein Text, wie zum Beispiel abc, schlägt die Konvertierung fehl und wir erhalten einen Fehler. In unserer Beispielanwendung gibt es einen Controller mit dem Namen Home, der eine Methode mit dem Namen Index enthält, welche die Start-Seite der Webanwendung erzeugt. Damit können wir die Startseite mit folgender URL aufrufen:

<http://localhost/Home/Index>

Die Portnummer lassen wir einfacheheitshalber weg. Da aber Index für unsere Route der Standardwert für die Action ist, können wir die Startseite unserer Webanwendung auch mit der folgenden verkürzten URL aufrufen.

<http://localhost/Home>

Wenn keine Action angegeben ist, wird der Standardwert verwendet. Da Home der Standardwert für den Controller ist, kann dieser Teil der URL auch entfallen. Damit verkürzt sich die URL für die Start-Seite der Webanwendung zu:

<http://localhost>

Als nächstes wollen wir die Klasse betrachten, die für den Controller Home zuständig ist. Beim Namen dieser Klasse wird an den Namen des Controllers noch der String Controller angehängt. Der Name der für den Controller Home zuständigen Klasse heißt also HomeController.cs. Sie befindet sich im Projektverzeichnis „Controllers“.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using System.Threading.Tasks;
6  using Microsoft.AspNetCore.Mvc;
7  using Microsoft.Extensions.Logging;
8  using Kunden.Web.Models;
9
10 namespace Kunden.Web.Controllers
11 {
12     public class HomeController : Controller
13     {
14         private readonly ILogger<HomeController> _logger;
15
16         public HomeController(ILogger<HomeController> logger)
17         {
18             _logger = logger;
19         }
20
21         public IActionResult Index()
22         {
23             return View();
24         }
25     }
26 }
```

## 16 ASP.NET MVC – Anwendungen für Web

```
24     }
25
26     public IActionResult Privacy()
27     {
28         return View();
29     }
30
31     [ResponseCache(Duration = 0, Location =
32     ResponseCacheLocation.None, NoStore = true)]
33     public IActionResult Error()
34     {
35         return View(new ErrorViewModel { RequestId =
36             Activity.Current?.Id ?? HttpContext.TraceIdentifier
37         });
38     }
39 }
40 }
```

Die Klasse HomeController wird von der Klasse Controller abgeleitet, welche vom .NET-Framework zur Verfügung gestellt wird.

Ein Controller wird vom System beim Aufruf einer zugehörigen URL instanziiert. Dabei wird dem Konstruktor der Controller-Klasse ein Objekt vom Typ `ILogger<T>` übergeben. Wobei T der Typ des Controllers ist, in unserem Beispiel `HomeController`. Der Konstruktor legt dieses Objekt in der private-Membervariablen `_logger` ab. Dieses Logger-Objekt kann innerhalb des Controllers verwendet werden, um Einträge bei evtl. auftretenden Fehlern oder zu Dokumentationszwecken in ein Log zu schreiben. Auf das Thema Logging können wir leider aus Platzgründen nicht näher eingehen.

Auf den Konstruktor folgt die Methode `Index()`, die für die Action `Index` zuständig ist. Eine Action-Methode, die eine HTML-Seite erzeugt, hat den Rückgabetyp `IActionResult`. Die Methode `Index()` gibt lediglich das Ergebnis der Methode `View()` zurück. Die Methode `View()` wird von der Klasse `Controller` geerbt und sucht nach einem Template mit dem gleichen Namen wie die rufende Action. In unserem Fall `Index.cshtml`. Aus dem Template wird eine HTML-Seite erzeugt und an den Browser gesendet, der diese Seite am Bildschirm darstellt. Wo wir die View `Index.cshtml` finden und wie sie aussieht, werden wir später betrachten. Die nächste Action-Methode des Controllers heißt `Privacy()` und erzeugt die Privacy-Seite. Genauso wie die Methode `Index()` gibt sie das Ergebnis der Methode `View()` zurück. Um die Privacy-Seite zu erzeugen, wird das Template `Privacy.cshtml` verwendet. Auch dieses Template betrachten wir später. Die letzte Action des Controllers ist die Methode `Error()`. Sie wird von ASP:NET MVC verwendet, um einen Fehler mit einem Template als HTML-Seite anzuzeigen. Die Methode `Error()` hat ein paar Besonderheiten. Sie erhält das Methodenattribut `[ResponseCache]`, welches so konfiguriert ist, dass die Methode beim Erzeugen der zugehörigen HTML-Seite keinen Cache verwendet, um im Bedarfsfall aktuelle Fehlermeldungen zu gewährleisten. Die Methode `Error()` gibt ebenfalls das Ergebnis der Methode `View()` zurück. Allerdings verwendet die Methode dabei

eine weitere Überladung der Methode `View()`, die ein Objekt von Typ `object` als Übergabeparameter entgegennimmt. Die Methode `Error()` übergibt der Methode `View()` eine neue Instanz der Klasse `ErrorViewModel` und setzt mit Hilfe eines Objektinitialisierers die Property `RequestId` der Klasse `ErrorViewModel` auf die Id des aktuellen http-Requests. Die Klasse `ErrorViewModel` betrachten wir später, wenn wir auf die Model-Schicht unseres ASP.NET MVC-Projekts näher eingehen.

Die Klasse `HomeController` hat drei Actions, die jeweils mit Hilfe von Templates verschiedene HTML-Seiten erzeugen. Die Templates für Actions `Index` und `Privacy` befinden sich im Projektverzeichnis `Views\Home`. Das Template für die `Error`-Action befindet sich im Projektverzeichnis `Views\Shared`. Wenn die Methode `View()` der Klasse `HomeController` ein Template sucht, sucht sie zuerst im Verzeichnis `Views\Home` ein Template mit dem gleichen Namen wie die rufende Action: zum Beispiel `Index.cshtml` für die `Index`-Action. Die Actionmethode `Error()` findet im Verzeichnis `Views\Home` kein Template, daher sucht sie weiter im Verzeichnis `Views\Shared`. Hier findet sie das Template `Error.cshtml`. In der Welt von ASP.NET MVC werden diese Templates als `Views` bezeichnet, daher wollen wir diese Bezeichnung im weiteren Verlauf verwenden. Betrachten wir die erste View: `Index.cshtml`

```
1 @{
2     ViewData["Title"] = "Home Page";
3 }
4
5 <div class="text-center">
6     <h1 class="display-4">Welcome</h1>
7     <p>Learn about <a href="https://docs.microsoft.com/aspnet/
8         core">building Web apps with ASP.NET Core</a>.</p>
9 </div>
```

Die View ist trivial aufgebaut. Sie besteht aus zwei Blöcken. Einem C#-Block und einem HTML-Block. Der C#-Block beginnt mit dem Zeichen `@` und ist in geschweifte Klammern eingeschlossen. Im C#-Block wird dem Dictionary `ViewData` für den Schlüssel „`Title`“ der Wert „`Home Page`“ zugewiesen. Das Dictionary `ViewData` wird in jeder View von ASP.NET MVC bereitgestellt. Wozu wir diesen Wert im Dictionary `ViewData` speichern, werden wir später sehen. Der HTML-Block der View stellt lediglich die Überschrift `Welcome` und einen Link zur ASP.NET-Dokumentation von Microsoft am Bildschirm dar.

Die View `Privacy.cshtml` ist ähnlich wie die `Index`-View aufgebaut:

```
1 @{
2     ViewData["Title"] = "Privacy Policy";
3 }
4 <h1>@ViewData["Title"]</h1>
5
6 <p>Use this page to detail your site's privacy policy.</p>
```

## 16 ASP.NET MVC – Anwendungen für Web

Die View beginnt ebenfalls mit einem C#-Block, der dem Dictionary ViewData für den Schlüssel „Title“ den Wert „Privacy Policy“ zuweist. Der HTML-Block verwendet diesen Wert aus dem Dictionary und stellt ihn am Bildschirm dar. Damit ASP.NET MVC den Text im HTML-Tag `<h1></h1>` nicht als String Literal interpretiert, muss dem Text das Zeichen @ vorangestellt werden. Unter der Überschrift gibt der HTML-Block der View noch den Text „Use this page to detail your site's privacy policy.“ aus.

In einer View können wir HTML und C# kombinieren. Die Softwarekomponente, die eine cshtml-Datei verarbeitet und daraus HTML für den Browser generiert, hat von Microsoft den Namen Razor-Engine erhalten. Daher wird eine View in ASP.NET MVC auch als Razor-Page bezeichnet. Wenn die Razor-Engine die Zeichen @{} findet, werden die folgenden Zeichen als C# interpretiert, - solange bis die Razor-Engine das Zeichen } findet. Innerhalb eines solchen Blocks können mehrere C#-Anweisungen stehen. Wenn die Razor-Engine das Zeichen @ findet, erwartet sie einen C#-Ausdruck. Die Engine wertet den Ausdruck aus und falls der Ausdruck nicht vom Typ String ist, ruft sie die `ToString()`-Methode des Ausdrucks auf. Danach ersetzt die Engine den C#-Ausdruck mit dem String, den die Auswertung zurückliefert.

Bevor wir die letzte View untersuchen, die von Visual Studio für unser ASP.NET MVC-Projekt erzeugt wurde, werfen wir einen kurzen Blick auf die Modelschicht. Die Klassen der Modelschicht befinden sich im Projektverzeichnis Models. In ASP.NET MVC werden Modelklassen auch als sogenannte ViewModels bezeichnet. ViewModels enthalten Properties und Methoden, die mit C#-Ausdrücken in Views ausgewertet werden und Daten für eine HTML-Seite bereitstellen. In unserem ersten Beispielprojekt gibt es zunächst nur ein ViewModel: ErrorViewModel.cs.

```
1  using System;
2
3  namespace Kunden.Web.Models
4  {
5      public class ErrorViewModel
6      {
7          public string RequestId { get; set; }
8
9          public bool ShowRequestId => !string.
10             IsNullOrEmpty(RequestId);
11      }
12 }
```

Ein ViewModel ist eine ganz normale C#-Klasse. In unserem Beispiel hat das ViewModel lediglich zwei Properties. Die erste Property heißt `RequestId` und soll die Id des aktuellen HTML-Request speichern. Die zweite Property heißt `ShowRequestId` und ist abhängig von `RequestId`. `ShowRequestId` gibt den Wert `false` zurück, wenn `RequestId` null ist, und `true`, wenn `RequestId` nicht null ist. In der Action `Error()` des Home-Controllers wird dieses ViewModel instanziiert, mit Daten versorgt und der View übergeben.

## 16.1 Eine neue ASP.NET MVC-Anwendung erstellen

Als nächstes betrachten wir die View Error.cshtml, die sich im Projektverzeichnis Views\Shared befindet:

```

1  @model ErrorViewModel
2  @{
3      ViewData["Title"] = "Error";
4  }
5
6  <h1 class="text-danger">Error.</h1>
7  <h2 class="text-danger">An error occurred while processing your
8  request.</h2>
9
10 @if (Model.ShowRequestId)
11 {
12     <p>
13         <strong>Request ID:</strong> <code>@Model.RequestId</
14         code>
15     </p>
16 }
17
18 <h3>Development Mode</h3>
19 <p>
20     Swapping to <strong>Development</strong> environment will
21     display more detailed information about the error that
22     occurred.
23 </p>
24 <p>
25     <strong>The Development environment shouldn't be enabled for
26     deployed applications.</strong>
27     It can result in displaying sensitive information from
28     exceptions to end users.
29     For local debugging, enable the <strong>Development</strong>
30     environment by setting the <strong>ASPNETCORE_ENVIRONMENT</
31     strong> environment variable to <strong>Development</strong>
32     and restarting the app.
33 </p>
```

Die View beginnt mit der Anweisung `@model ErrorViewModel`. Damit teilen wir der Razor-Engine mit, dass die View ein Model vom Typ ErrorViewModel verwendet. Danach folgt ein C#-Block, der dem Dictionary Element `ViewData[„Title“]` den Wert „Error“ zuweist. Dann werden mit den HTML-Tags `<h1>` und `<h2>` zwei Überschriften ausgegeben. Als nächstes folgt eine C# If-Anweisung:

```

1  @if (Model.ShowRequestId)
2  {
3 }
```

16

Mit dem Schlüsselwort `Model` greifen wir auf das der View übergebene ViewModel vom Typ `ErrorViewModel` zu und überprüfen mit der Property `ShowRequestId`, ob die Property `RequestId` des ViewModels gesetzt ist. Wenn die Property `RequestId` gesetzt ist, fügt die Razor-Engine den HTML-Block:

## 16 ASP.NET MVC – Anwendungen fürs Web

```
1 <p>
2   <strong>Request ID:</strong> <code>@Model.RequestId</code>
3 </p>
```

in die zu erzeugende HTML-Seite ein. Wenn `RequestId` null ist, wird dieser HTML-Block übersprungen. Dieser HTML-Block zeigt die Id des http-Request an, indem mit dem C#-Ausdruck `@Model.RequestId` die Property `RequestId` des ViewModels ausgewertet wird. Zum Schluss wird noch ein Erklärungstext für die Environment-Variablen `ASPNETCORE_ENVIRONMENT` ausgegeben.

Bisher haben wir gesehen, dass Visual Studio für eine ASP.NET MVC-Webanwendung eine Error-Action, eine Error-View und ein Error-ViewModel erzeugt. Wie das in der Webanwendung verwendet wird, wollen wir jetzt untersuchen. Dazu ändern wir die Index-Action wie folgt ab:

```
1 public IActionResult Index()
2 {
3     int a = 5;
4     int b = 0;
5     int c = a / b;
6
7     return View();
8 }
```

Mit diesem Code erzeugen wir eine `DivideByZeroException` und simulieren so einen Laufzeitfehler, der vom Programmierer nicht vorausgesehen und somit auch nicht mit einer try-catch-Anweisung abgefangen wurde. Um zu sehen, wie unsere Webanwendung sich bei unseren Benutzern verhalten würde, starten wir die Webanwendung ohne Debugger. Dazu wählen wir im Hauptmenü von Visual Studio den Menüpunkt „Debuggen/Starten ohne Debuggen“ aus oder drücken die Tastenkombination Strg F5.

## 16.1 Eine neue ASP.NET MVC-Anwendung erstellen

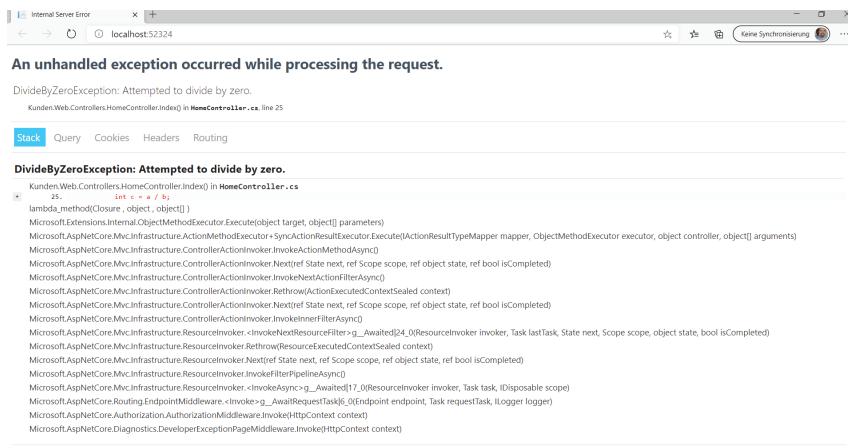


Abb. 16.1.7 Ein Laufzeitfehler in der Webanwendung

Unsere Webanwendung zeigt eine Seite mit detaillierten technischen Informationen über den aufgetretenen Fehler. In den meisten Fällen ist eine derartige Anzeige unerwünscht, da normale Benutzer dadurch nur verwirrt werden und Hacker zudem wertvolle Informationen finden, um unsere Webanwendung anzugreifen. Um zu verstehen, warum im Fehlerfall diese Seite angezeigt wird, betrachten wir noch einmal die Datei Startup.cs im Wurzelverzeichnis des Projekts. Am Anfang der Methode Configure () befindet sich folgender Programmcode:

```

1 if (env.IsDevelopment ())
2 {
3     app.UseDeveloperExceptionPage ();
4 }
5 else
6 {
7     app.UseExceptionHandler ("~/Home/Error");
8 }

```

Dieser Code weist ASP.NET MVC an, im Fehlerfall die sogenannte Developer Exception Page, also die obige HTML-Seite, anzuzeigen, falls wir uns in einer Entwicklungsumgebung befinden, ansonsten soll die Adresse /Home/Error verwendet werden. Diese Adresse führt zur Action Error im Controller Home und somit zu unserer Error-View. Da wir beim Starten der Webanwendung die Developer Exception Page sehen, befinden wir uns also in einer Entwicklungsumgebung. Wie wir das Steuern können, sehen wir, wenn wir im Projektmappen-Explorer mit der rechten Maustaste auf unser Projekt klicken und das Kontextmenü Eigenschaften auswählen.

## 16 ASP.NET MVC – Anwendungen für Web

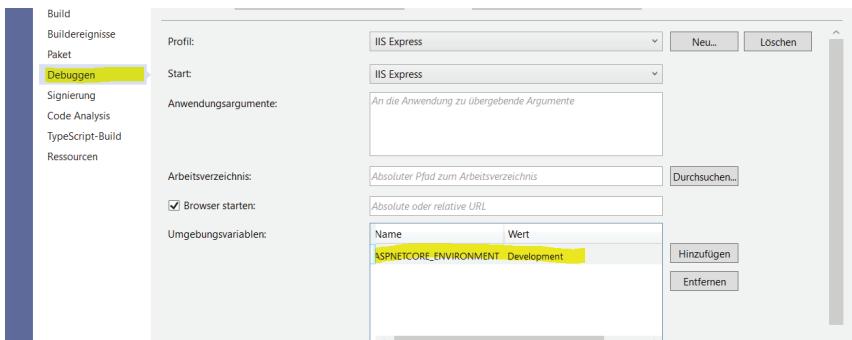


Abb. 16.1.8 Die Debug-Einstellungen einer Webanwendung

Wenn wir den Reiter „Debuggen“ auswählen, sehen wir, dass die Umgebungsvariable `ASPNETCORE_ENVIRONMENT` auf den Wert `Development` gesetzt ist. Das heißt, wenn wir unsere Webanwendung aus Visual Studio starten, befindet sie sich in einer Entwicklungsumgebung. Als nächstes ändern wir den Wert „`Development`“ zu „`Production`“ und starten die Webanwendung wieder ohne Debuggen.

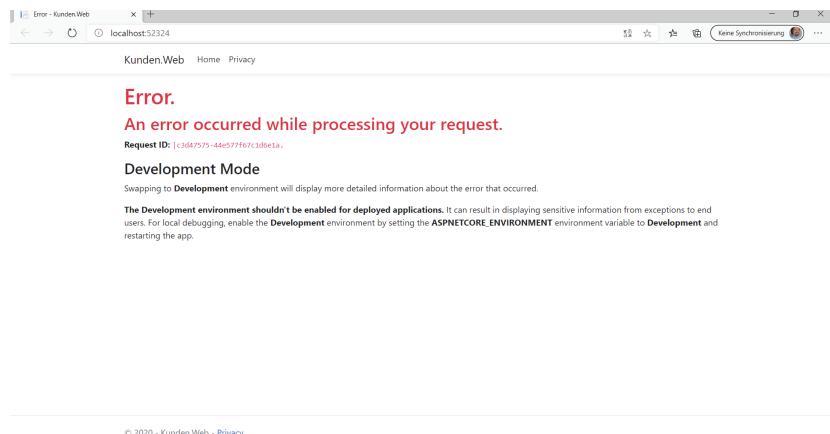


Abb. 16.1.9 Die Error-View im Browser

Jetzt wird im Fehlerfall unsere Error-View angezeigt, die keine technischen Details preisgibt. Zudem können wir als Programmierer diese Error-View nach unseren Vorstellungen anpassen. Nach diesem Exkurs in die Error-View ändern wir die `Index`-Action des Home-Controllers wieder zurück:

```
1 public IActionResult Index()
2 {
```

## 16.1 Eine neue ASP.NET MVC-Anwendung erstellen

```

3     return View();
4 }
```

Als nächstes wollen wir eine weitere Komponente der Architektur einer ASP.NET MVC-Webanwendung betrachten: Das sogenannte Layout. Wenn wir unsere Webanwendung im Browser betrachten, hat sie ein Menü mit den Punkten Kunden.Web, Home und Privacy. Zudem gibt es eine Fußzeile die einen Link zur Privacy-Seite enthält. Von dem Menü und der Fußzeile ist aber in den Views nicht zu sehen. Außerdem wäre es umständlich, wenn sich in allen Views der HTML-Code für Menü und Fußzeile wiederholen würde. Im Projektverzeichnis Views befindet sich die Datei \_ViewStart.cshtml.

```

1 @{
2     Layout = "_Layout";
3 }
```

In dieser Datei befindet sich ein C#-Block, der zum Start von jeder View ausgeführt wird. In diesem C#-Block wird schließlich der Property Layout der Wert „\_Layout“ zugewiesen. Damit teilen wir ASP.NET MVC mit, dass jede View das Layout \_Layout verwenden soll. Dieses Layout finden wir in der Datei \_Layout.cshtml im Projektverzeichnis Views\Shared.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width, initial-
6         scale=1.0" />
7     <title>@ ViewData["Title"] - Kunden.Web</title>
8     <link rel="stylesheet" href="~/lib/bootstrap/dist/css/
9         bootstrap.min.css" />
10    <link rel="stylesheet" href="~/css/site.css" />
11 </head>
12 <body>
13     <header>
14         <nav class="navbar navbar-expand-sm navbar-toggleable-sm
15             navbar-light bg-white border-bottom box-shadow mb-3">
16             <div class="container">
17                 <a class="navbar-brand" asp-area="" asp-controller
18                     ="Home" asp-action="Index">Kunden.Web</a>
19                 <button class="navbar-toggler" type="button"
20                     data-toggle="collapse" data-target=".navbar-
21                     collapse" aria-controls="navbarSupportedContent"
22                     aria-expanded="false" aria-label="Toggle
23                     navigation">
24                     <span class="navbar-toggler-icon"></span>
25                 </button>
26                 <div class="navbar-collapse collapse d-sm-inline-
27                     flex flex-sm-row-reverse">
28                     <ul class="navbar-nav flex-grow-1">
29                         <li class="nav-item">
30                             <a class="nav-link text-dark" asp-
31                                 area="" asp-controller="Home" asp-
```

## 16 ASP.NET MVC – Anwendungen fürs Web

```

32             action="Index">Home</a>
33         </li>
34         <li class="nav-item">
35             <a class="nav-link text-dark" asp-
36                 area="" asp-controller="Home" asp-
37                 action="Privacy">Privacy</a>
38         </li>
39     </ul>
40   </div>
41 </nav>
42 </header>
43 <div class="container">
44     <main role="main" class="pb-3">
45         @RenderBody()
46     </main>
47 </div>
48
49
50 <footer class="border-top footer text-muted">
51     <div class="container">
52         &copy; 2020 - Kunden.Web - <a asp-area="" asp-
53             controller="Home" asp-action="Privacy">Privacy</a>
54     </div>
55 </footer>
56 <script src="~/lib/jquery/dist/jquery.min.js"></script>
57 <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.
58 js"></script>
59 <script src="~/js/site.js" asp-append-version="true"></
60 script>
61 @RenderSection("Scripts", required: false)
62 </body>
63 </html>
```

Den vollständigen HTML-Code dieser Datei zu erklären, würde den Rahmen dieses Buches sprengen. Daher werden wir nur die relevanten Teile besprechen.

Zeile 6:

```
1 <title>@ViewData["Title"] - Kunden.Web</title>
```

Dieses HTML-Tag legt den Titel der HTML Seite fest, der oben im Browserfenster angezeigt wird. Er wird zusammengesetzt aus dem Titel, den eine View in das ViewData-Dictionary schreibt, und dem Text „- Kunden.Web“.

Zeile 14:

```
1 <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
2     action="Index">Kunden.Web</a>
```

Das ist der Link hinter dem Menüpunkt Kunden.Web. Er ist so konfiguriert, dass mit diesem Link der Action Index des Controllers Home aufgerufen wird.

Zeile 22:

```
1 <a class="nav-link text-dark" asp-area="" asp-controller="Home"
2 asp-action="Index">Home</a>
```

Dieser Link befindet sich hinter dem Menüpunkt Home und ruft ebenfalls die Action Index des Controllers Home auf.

Zeile 25:

```
1 <a class="nav-link text-dark" asp-area="" asp-controller="Home"
2 asp-action="Privacy">Privacy</a>
```

Dieser Link steht hinter dem Menüpunkt Privacy und ruft die Action Privacy des Controllers Home auf.

Zeile 34:

@RenderBody()

An dieser Stelle wird das generierte HTML der jeweiligen View (Home, Privacy oder Error) eingefügt.

Durch diesen Layout-Mechanismus schreiben wir den HTML-Code für Menü und Fußzeile nur einmal und verwenden ihn automatisch bei jeder neuen View.

## 16.2 Einen Controller und eine View hinzufügen

Als nächstes werden wir einen weiteren Controller und eine weitere View für unsere Webanwendung erstellen und in das Menü einbinden. Dazu klicken wir mit der rechten Maustaste auf das Projektverzeichnis Controllers und wählen Hinzufügen\Controller ... aus dem Kontextmenü.

## 16 ASP.NET MVC – Anwendungen fürs Web

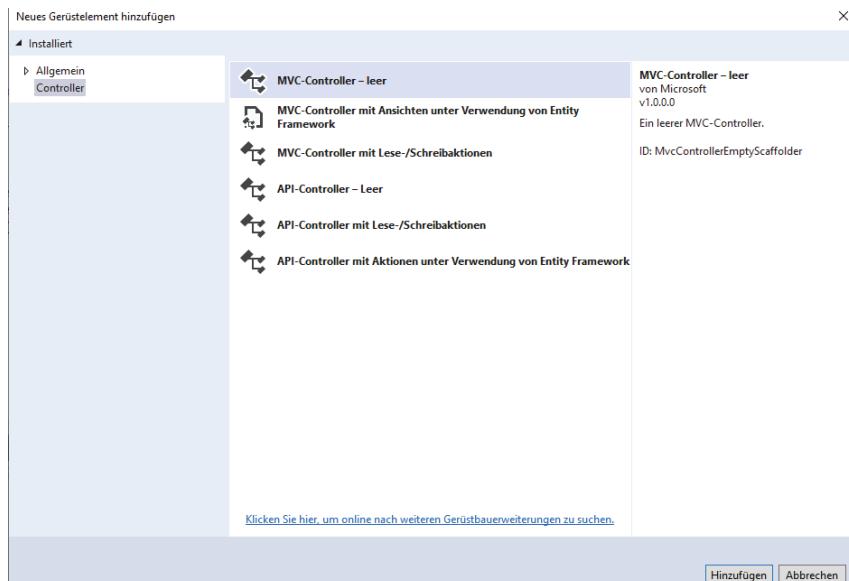


Abb. 16.2.1 Hinzufügen eines neuen Controllers

Wir wählen MVC-Controller-leer aus und klicken auf die Schaltfläche „Hinzufügen“.

## 16.2 Einen Controller und eine View hinzufügen

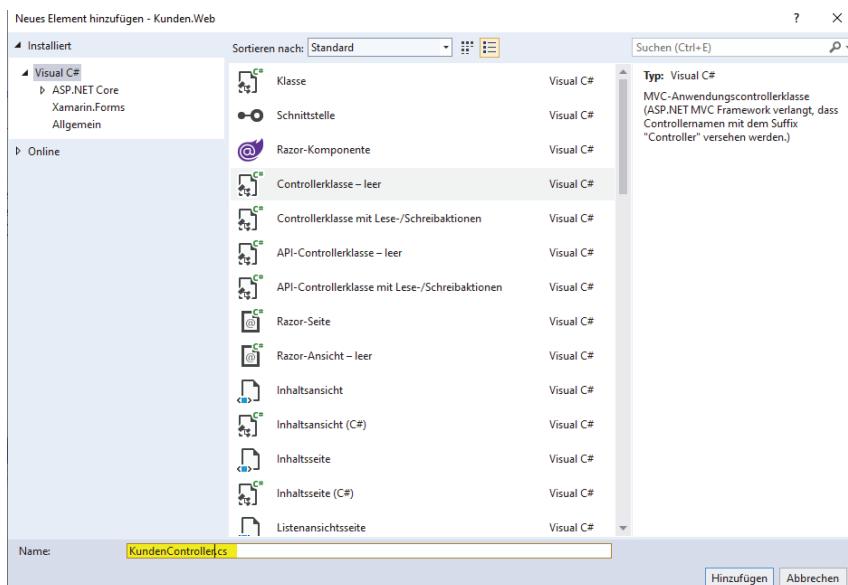


Abb. 16.2.2 Erstellen des KundenControllers

Als Namen für den Controller vergeben wir KundenController.cs.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6
7  namespace Kunden.Web.Controllers
8  {
9      public class KundenController : Controller
10     {
11         public IActionResult Index()
12         {
13             return View();
14         }
15     }
16 }
```

16

Visual Studio hat bereits die Action-Methode `Index()` für uns erstellt. Diese Action wird gerufen, wenn wir im Browser die Adresse <http://localhost/Kunden> eingeben. Sie erinnern sich: Die Action wird im Projektverzeichnis `Views\Kunden` eine View mit dem Namen `Index.cshtml` suchen. Damit die Action fündig wird, erstellen wir unter dem Projektverzeichnis `Views` das Verzeichnis `Kunden`. Dann klicken wir mit der rech-

## 16 ASP.NET MVC – Anwendungen fürs Web

ten Maustaste auf das neu erstellte Projektverzeichnis Kunden und wählen „Hinzufügen/Ansicht ...“ aus dem Kontextmenü aus.

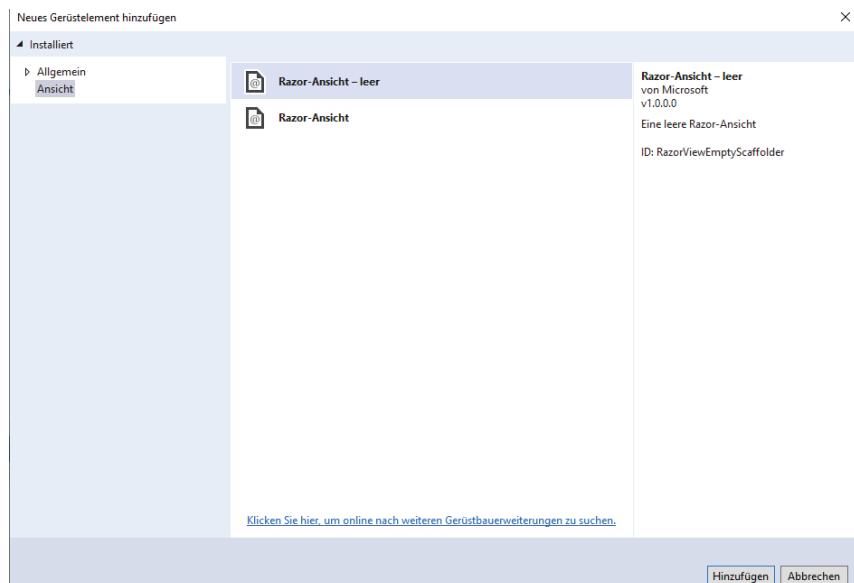


Abb. 16.2.3 Hinzufügen einer neuen View

Wir wählen Razor-Ansicht-leer aus und klicken auf die Schaltfläche „Hinzufügen“.

## 16.2 Einen Controller und eine View hinzufügen

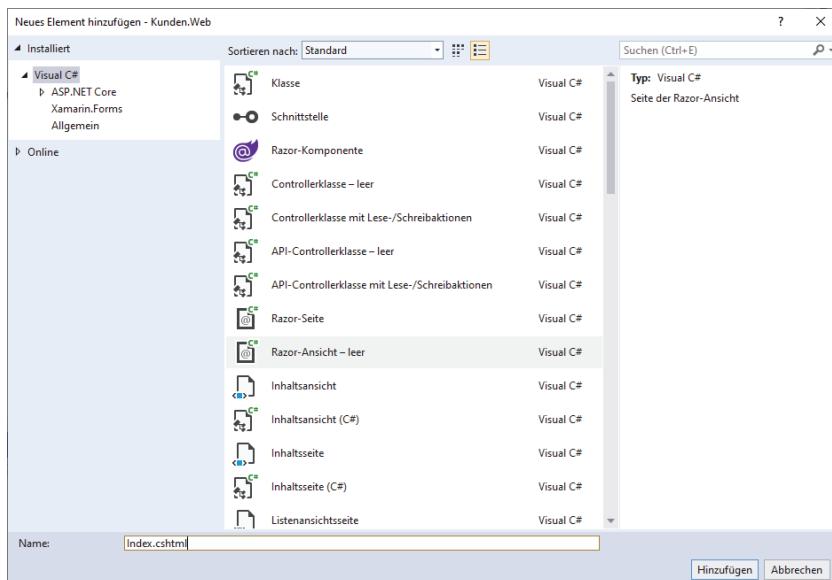


Abb. 16.2.4 Erstellen der Kunden-Index-View

Visual Studio schlägt uns für die View den Namen Index.cshtml vor, welchen wir akzeptieren und mit einem Click auf die Schaltfläche „Hinzufügen“ bestätigen.

Die erstellte Datei Views\Kunden\Index.cshtml ändern wir wie folgt ab:

```

1  @{
2      ViewData["Title"] = "Kundenliste";
3  }
4  <h1>@ViewData["Title"]</h1>

```

Zu Beginn weisen wir in einem C#-Block dem ViewData Dictionary für den Schlüssel „Title“ den Wert „Kundenliste“ zu. Im darauffolgenden HTML-Block geben wir ViewData["Title"], also den Text Kundenliste, mit einem <h1>-Tag als Überschrift aus.

16

Als nächstes binden wir die neue Seite Kundenliste in das Menü der Webanwendung ein. Dazu ändern wir die Datei Views\Shared\\_Layout.cshtml.

Hinter dem Block:

```

1  <li class="nav-item">
2      <a class="nav-link text-dark" asp-area="" asp-
3          controller="Home" asp-action="Privacy">Privacy</a>

```

## 16 ASP.NET MVC – Anwendungen fürs Web

```
4 </li>
```

fügen wir den folgenden Block ein:

```
1 <li class="nav-item">
2   <a class="nav-link text-dark" asp-area="" asp-
3     controller="Kunden" asp-action="Index">Kundenliste</a>
4 </li>
```

Damit erweitern wir das Menü um einen weiteren Link, der die Action Index des Controllers Kunden aufruft. Wenn wir unsere Webanwendung starten, sehen wir den neuen Menüpunkt.

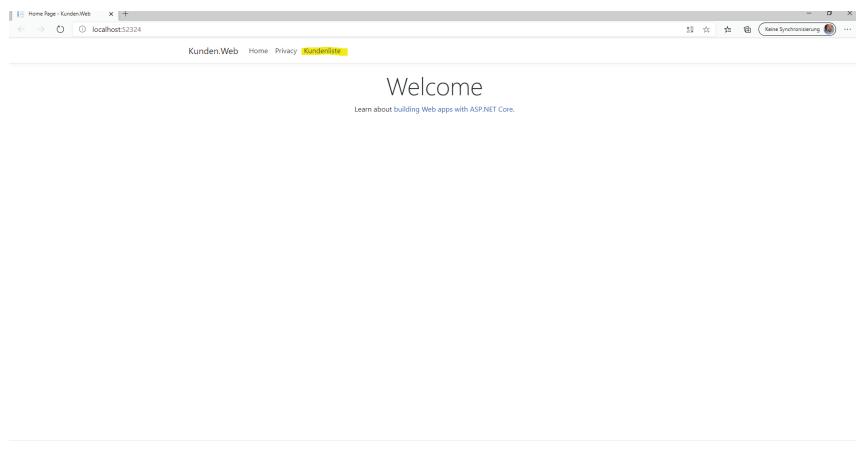


Abb. 16.2.5 Der neue Menüpunkt Kundenliste

Mit einem Klick auf den Menüpunkt können wir die neue Seite Kundenliste anzeigen.

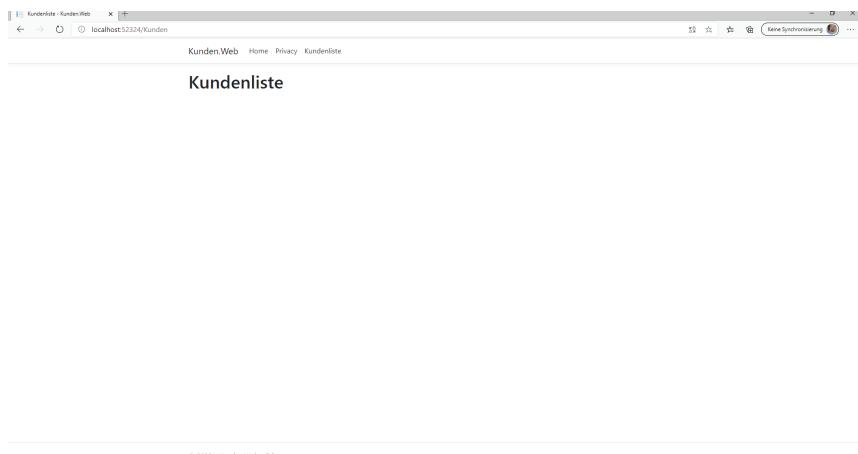


Abb. 16.2.6 Die neue Seite Kundenliste

Wir hätten die Seite Kundenliste auch implementieren können, wenn wir auf einen zweiten Controller verzichtet hätten und lediglich für den HomeController eine neue Action und eine zugehörige View erstellt hätten. In der Praxis haben Webanwendungen aber sehr viele Seiten und Unterseiten. Wenn wir alle Seiten in einen Controller packen würden, wäre dieser Controller sehr schnell sehr unübersichtlich. Daher ist es ratsam, sein Projekt nach fachlichen Themen zu strukturieren und für jedes Thema einen eigenen Controller zu erstellen. Wir haben jetzt einen KundenController erstellt. Für einen weiteren Ausbau der Webanwendung könnte man sich zum Beispiel einen AnsprechpartnerController vorstellen.

### 16.3 Daten mit dem Controller bereitstellen

In diesem Unterkapitel wollen wir mit unserem `KundenController` Daten aus der Kunden-Datenbank aus Kapitel 15 bereitstellen. Damit wir auf diese Datenbank zugreifen können, müssen wir zuerst den Microsoft Entity-Framework zu unserem ASP.NET MVC-Projekt hinzufügen. Eine Anleitung dazu finden Sie in Kapitel 15.3.

Als nächstes erstellen wir im Projekt `Kunden.Web` das Verzeichnis `DatenbankModell` und fügen dem Verzeichnis die folgende Klasse `Kunde` hinzu:

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.ComponentModel.DataAnnotations.Schema;
5  using System.Linq;
6  using System.Threading.Tasks;
7
```

## 16 ASP.NET MVC – Anwendungen fürs Web

```

8  namespace Kunden.Web.DatenbankModel
9  {
10     [Table("Kunde")]
11     public class Kunde
12     {
13         [Key]
14         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
15         [Column("id")]
16         public int KundeId { get; set; }
17
18         [Required]
19         [MaxLength(50)]
20         [Column("firma")]
21         public string Firma { get; set; }
22
23         [MaxLength(50)]
24         [Column("strasse")]
25         public string Strasse { get; set; }
26
27         [MaxLength(5)]
28         [Column("plz")]
29         public string PLZ { get; set; }
30
31         [MaxLength(50)]
32         [Column("ort")]
33         public string Ort { get; set; }
34     }
35 }
36 }
```

Die Klasse Kunde ist die Entity-Klasse Kunde aus dem Kapitel 15.4. Nur die Navigation-property für die Ansprechpartner haben wir weggelassen, da wir uns in dieser Webanwendung zunächst nur mit Kunden beschäftigen wollen. Als nächstes fügen wir die Klasse KundenKontext dem Projektverzeichnis DatenbankModell hinzu.

```

1  using Microsoft.EntityFrameworkCore;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace Kunden.Web.DatenbankModel
8  {
9      public class KundenKontext : DbContext
10     {
11         public KundenKontext(DbContextOptions<KundenKontext>
12             options)
13             : base(options)
14         {
15         }
16
17         public DbSet<Kunde> Kunden { get; set; }
18     }
19 }
```

Die Klasse KundenKontext sieht etwas anders aus als wir sie im Kapitel 15.4 erstellt haben. Das ist nötig, da im Rahmen einer ASP.NET MVC-Webanwendung ein Datenbank-Kontext nicht einfach mit dem Schlüssel new erstellt, sondern bei der Webanwendung registriert wird. Doch dazu später mehr. In diesem KundenKontext verzichten wir auf das Überschreiben der Methode OnConfiguring(), um dem Kontext einen Datenbank-Connection-String mitzuteilen. Das erledigen wir später, wenn wir den Kontext registrieren. Für diese Variante des Datenbankkontexts erstellen wir einen Konstruktor, der ein Objekt vom Typ DbContextOptions<KundenKontext> entgegennimmt. Alles, was dieser Konstruktor tun muss, ist den Konstruktor der Basisklasse DbContext, der die gleiche Signatur besitzt, zu rufen und den Übergabeparameter options weiterzureichen. Diesen Konstruktor benötigt ASP.NET MVC für die Registrierung des Datenbankkontexts. Als nächstes registrieren wir den Datenbankkontext bei unserer ASP.NET MVC-Anwendung. Dazu fügen wir der Klasse Startup.cs im Wurzelverzeichnis des Projekts das folgende using-Statement hinzu:

```
1 using Microsoft.EntityFrameworkCore;
```

Als nächstes fügen wir am Ende der Methode ConfigureServices() den folgenden Code ein:

```
1 services.AddDbContext<KundenKontext>(options => options.  
2 UseSqlServer(Configuration.GetConnectionString("KundenContext")));
```

Die Methode ConfigureServices() wird von ASP.NET MVC aufgerufen und bekommt das Objekt services vom Typ IServiceCollection übergeben. Wir rufen die Methode AddDbContext<KundenKontext>() von services auf und übergeben ihr den Lambda Ausdruck:

```
1 options => options.UseSqlServer(Configuration.  
2 GetConnectionString("KundenContext"))
```

Das ist eine Methode, die ein Objekt vom Typ DbContextOptionsBuilder übergeben bekommt und die für die Klasse DbContextOptionsBuilder die Erweiterungsmethode UseSqlServer() aufruft. Damit uns diese Erweiterungsmethode zur Verfügung steht, haben wir zuvor die Klassenbibliothek Microsoft.EntityFrameworkCore eingebunden. Die Methode UseSqlServer() erwartet als Übergabeparameter einen Datenbank-Connectionstring. Hier könnten wir direkt den Connectionstring für unsere Datenbank übergeben. Aber dann steht der Connectionstring im Programmcode. Das heißt, wenn unsere Datenbank auf einen anderen Server umzieht, müssen wir den Connectionstring im Programmcode ändern, unser Projekt neu kompilieren und neu ausliefern. Deshalb verwenden wir eine elegantere Lösung. Wir beschaffen uns den Connectionstring, indem wir die Methode GetConnectionString() der Property Configuration der Klasse Startup aufrufen. Der Methode GetConnectionString() übergeben wir den selbstgewählten String „KundenContext“ als Namen für unseren ConnectionString. Jetzt müssen wir

## 16 ASP.NET MVC – Anwendungen fürs Web

nur noch den richtigen Connectionstring diesem Namen zuordnen. Das können wir in der Datei appsettings.json vornehmen. Diese Datei befindet sich im Wurzelverzeichnis unseres Projekts. In dieser Datei können verschiedene Konfigurationseinstellungen für unsere Webanwendung getätigt werden. Die Datei liegt im sogenannten json-Format vor. Das ist ein lesbares Textformat, das nicht kompiliert wird. Diese Datei wird mit unserer Webanwendung ausgeliefert. Das heißt, ein Administrator kann dort Konfigurationsänderungen vornehmen: zum Beispiel den Connectionstring anpassen, wenn die Datenbank auf einen anderen Server umzieht. Daher ändern wir die Datei appsettings.json wie folgt ab:

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "AllowedHosts": "*",
10   "ConnectionStrings": {
11     "KundenContext": "Server=.\\\
12       SQLEXPRESS;Database=Kunden;Trusted_
13       Connection=True;MultipleActiveResultSets=true"
14   }
15 }
```

Leider können wir aus Platzgründen in diesem Buch das json-Format nicht erklären. Für alle, die sich näher mit diesem Format beschäftigen wollen, empfiehlt sich die Webseite:

[https://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](https://de.wikipedia.org/wiki/JavaScript_Object_Notation)

Wie man einen Connectionstring in der Datei appsettings.json konfiguriert, lässt sich im oben genannten Beispiel aber leicht erkennen. Beachten Sie, dass die gleichen Escape-Regeln gelten wie für C# Stringliterale. Deshalb müssen wir im oben genannten Beispiel das Zeichen \ doppelt schreiben. Nachdem wir gesehen haben, wie wir den Datenbankkontext bei ASP.NET MVC registrieren können, sehen wir uns jetzt an, welchen Vorteil wir daraus gewinnen können. Dazu ändern wir den KundenController wie folgt ab:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Kunden.Web.DatenbankModel;
6  using Microsoft.AspNetCore.Mvc;
7
8  namespace Kunden.Web.Controllers
9  {
```

```
10 public class KundenController : Controller
11 {
12     private KundenKontext kontext;
13
14     public KundenController(KundenKontext kontext)
15     {
16         this.kontext = kontext;
17     }
18
19     public IActionResult Index()
20     {
21         return View();
22     }
23 }
24 }
```

Der KundenController hat jetzt einen Konstruktor, der ein KundenKontext-Objekt als Übergabeparameter erwartet und diesen KundenKontext in der private-Membervariablen kontext ablegt. Der KundenController wird von ASP.NET MVC erzeugt und da wir den KundenKontext bei AS.NET MVC registriert haben, kann ASP.NET MVC auch einen neuen KundenKontext erstellen und an den Konstruktor des KundenControllers übergeben. Für uns bedeutet das, dass wir bei jedem Controller, der auf die Datenbank zugreifen möchte, einfach nur einen Parameter vom Typ KundenKontext im Konstruktor einführen müssen. Wenn wir diesen übergebenen KundenKontext in einer private-Membervariablen ablegen, kann er von jeder Action des Controllers verwendet werden. Wir müssen uns auch nicht mehr um das Freigeben der nicht verwalteten Ressourcen des Datenbankkontexts kümmern. Das erledigt ASP.NET MVC für uns. Dieses Entwurfsmuster, bei dem eine registrierte Klasse einfach dem Konstruktor einer anderen Klasse übergeben wird, nennt man Dependency Injection. Man sagt, eine Klasse wird dem Konstruktor injiziert. Die Komponente von ASP.NET MVC, die sich um das Registrieren und Injizieren von Klassen kümmert, wird Dependency Injection Container oder kurz DI-Container genannt.

Damit wir in einer View eine Kundenliste anzeigen können, muss die Index Action des KundenControllers ein ViewModel an die Methode View() übergeben. Für dieses ViewModel könnten wir die Entity-Klasse Kunde verwenden und zum Beispiel ein ViewModel vom Type `IEnumerable<Kunde>` übergeben. In der Praxis ist so ein Ansatz aber zu kurz gegriffen. Ein ViewModel soll eine HTML-Seite vollständig mit Daten versorgen und nicht nur eine Kundenliste bereitstellen. Auch wenn wir außer der Kundenliste in unserem ersten Beispiel zunächst keine weiteren Daten benötigen, erstellen wir uns im Projektverzeichnis Models die ViewModels KundenViewModel und KundenListeViewModel.

```
1 using Kunden.Web.DatenbankModel;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Threading.Tasks;
```

## 16 ASP.NET MVC – Anwendungen für Web

```
6  namespace Kunden.Web.Models
7  {
8      public class KundenViewModel
9      {
10         public int KundeId { get; set; }
11
12         public string Firma { get; set; }
13
14         public string Strasse { get; set; }
15
16         public string PLZ { get; set; }
17
18         public string Ort { get; set; }
19
20         public static KundenViewModel AusEntity(Kunde kunde)
21         {
22             return new KundenViewModel
23             {
24                 KundeId = kunde.KundeId,
25                 Firma = kunde.Firma,
26                 Strasse = kunde.Strasse,
27                 PLZ = kunde.PLZ,
28                 Ort = kunde.Ort,
29             };
30         }
31     }
32 }
33 }
```

Die Klasse KundenViewModel hat in unserem Beispiel die gleichen Properties wie die Entity-Klasse Kunde. Sie könnte aber noch weitere Properties haben, die zur Anzeige in der Kundenliste benötigt werden. Zusätzlich hat die Klasse noch die statische Methode AusEntity(). Diese Methode erledigt das sogenannte Mapping. Das bedeutet die Zuordnung der Daten der Entity-Klasse zu den Daten der ViewModel-Klasse. In unserem einfachen Beispiel ist das lediglich eine Eins-zu-eins-Zuordnung der Properties der beiden Klassen. Als nächstes benötigen wir die Klasse KundenListeViewModel:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace Kunden.Web.Models
7  {
8      public class KundenListeViewModel
9      {
10         public IEnumerable<KundenViewModel> Kunden { get; set; }
11     }
12 }
```

Die Klasse KundenListeViewModel hat nur eine Property von Typ IEnumerable<KundenViewModel>. Mit dieser stellt sie eine Kundenliste zur Verfügung.

Damit nun die Index Action des KundenControllers Daten zur Verfügung stellen kann, ändern wir sie wie folgt ab:

```
1 public IActionResult Index()
2 {
3     var kundenListeViewModel = new KundenListeViewModel();
4     kundenListeViewModel.Kunden = kontext.Kunden.Select(k =>
5         KundenViewModel.AusEntity(k));
6     return View(kundenListeViewModel);
7 }
```

Zuerst erzeugen wir ein neues Objekt vom Type KundenListeViewModel. Dann weisen wir der Property Kunden des KundenListeViewModel-Objekts eine Liste aller Kunden aus der Datenbank zu. Dazu verwenden wir den KundenKontext, den der KundenController via Dependency Injection erhalten hat, und die Linq-Methode Select(). Der Methode Select() übergeben wir einen Lambda-Ausdruck, der mit Hilfe der statischen Methode AusEntity() KundenViewModel-Objekte aus den Kunde-Objekten erstellt, die uns vom KundenKontext aus der Datenbank geliefert werden.

Zum Schluss übergeben wir das KundenListeViewModel-Objekt der Methode View() und geben deren Ergebnis zurück.

Damit übergibt die Index Action des KundenControllers seiner zugehörigen View Index.cshtml im Projektverzeichnis Views\Kunden ein ViewModel, das Daten für eine Kundenliste bereitstellt. Wie wir diese Daten auf der Kundenliste-Seite unserer Webanwendung anzeigen können, betrachten wir im nächsten Unterkapitel.

## 16.4 Daten mit der View anzeigen

Damit wir mit der Index View, im Verzeichnis Views\Kunden eine Kundenliste anzeigen können, ändern wir diese View wie folgt:

```
1 @model KundenListeViewModel
2 @{
3     ViewData["Title"] = "Kundenliste";
4 }
5
6 <h1>@ViewData["Title"]</h1>
7
8 <table class="table">
9     <thead>
10        <tr>
```

## 16 ASP.NET MVC – Anwendungen fürs Web

```

11      <th>Kunden-Id</th>
12      <th>Firma</th>
13      <th>Strasse</th>
14      <th>PLZ</th>
15      <th>Ort</th>
16    </tr>
17  </thead>
18  <tbody>
19    @foreach (var kunde in Model.Kunden)
20    {
21      <tr>
22        <td>@kunde.KundeId</td>
23        <td>@kunde.Firma</td>
24        <td>@kunde.Strasse</td>
25        <td>@kunde.PLZ</td>
26        <td>@kunde.Ort</td>
27      </tr>
28    }
29  </tbody>
30 </table>
31

```

In der ersten Zeile legen wir fest, dass die View ein Model vom Typ KundenListeView-Model verarbeiten soll. Das ist genau der Typ der, der View von ihrer zugehörigen Action übergeben wird. Danach setzen wir - wie in unseren anderen Views auch - ViewData[„Title“] und geben dann diesen Titel als Überschrift mit einem <h1>-Tag aus. Danach folgt eine Html-Tabelle, die durch ein <table>-Tag angegeben wird. Das <table>-Tag erhält noch das class-Attribut table, das aber nur dazu da ist, die Optik der Tabelle etwas aufzuhübschen. Die Definition des class-Attributs wurde von Visual Studio beim Erstellen der Webanwendung erzeugt. Um Html-Elemente mit class-Attributen formatieren zu können, benötigt man Kenntnisse in HTML/CSS. Dieses Thema würde den Rahmen dieses Buchs sprengen. Wer sich mit diesem Thema beschäftigen möchte, solle sich die folgende Webseite ansehen.

<https://wiki.selfhtml.org/wiki/CSS>

Das <table>-Tag enthält ein <thead>-Tag, welches den sogenannten Header der HTML-Tabelle enthält. Der Header stellt die Spaltenüberschriften der Tabelle dar. Im <thead>-Tag befindet sich ein <tr>-Tag, das eine Tabellenzeile repräsentiert. Im <tr>-Tag befinden sich die Überschriftenfelder der Tabelle, wobei jedes Feld mit einem <th>-Tag dargestellt wird. Die einzelnen <th>-Tags enthalten die Überschriften im Klartext. Nachdem <thead>-Tag folgt ein <tbody>-Tag, das den Tabellenkörper enthält. Also die Tabellenzeilen, die auf die Überschriftenzeile folgen. Das <tbody>-Tag enthält eine foreach-Schleife. Da wir an dieser Stelle C# innerhalb von HTML verwenden, müssen wir unserer foreach-Schleife das Zeichen @ voranstellen. Mit der foreach-Schleife laufen wir über die Property Kunden des Models der View. Die Property Kunden ist vom Typ IEnumerable<KundeViewModel> und enthält alle Kunden aus der Datenbank. Innerhalb der Schleife wiederholen wir

## 16.5 Eine eigene View, um Daten zu ändern

für jeden Kunden ein <tr>-Tag. Damit geben wir für jeden Kunden eine Zeile aus. Das <tr>-Tag enthält für jede Property eines Kunden ein <td>-Tag, was einer Tabelenzelle entspricht. Im <td>-Tag geben wir die jeweilige Property des im aktuellen Schleifendurchgang zu verarbeitenden Kunden aus. Da es sich hier wieder um C# in HTML handelt, müssen wir auch hier wieder den C#-Ausdruck mit dem Zeichen @ beginnen. Wenn wir unsere Webanwendung jetzt starten und im Menü auf Kundenliste klicken, können wir die neue Index View überprüfen.

Kunden-Id	Firma	Strasse	PLZ	Ort
2	Kanzlei Müller & Söhne	Hauptstrasse 1	54321	Neustadt
3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen

Abb. 16.4.1 Die Kundenliste der Webanwendung

## 16.5 Eine eigene View, um Daten zu ändern

Nachdem wir mit unserer Webanwendung eine Kundenliste anzeigen können, wollen wir auch einzelne Kunden ändern können. Dazu benötigen wir eine eigene View, die Daten eines einzelnen Kunden mit HTML-Elementen anzeigt, die uns auch das Ändern der Daten gestatten. Für diese View benötigen wir auch eine Action, damit wir diese View aufrufen können. Daher fügen wir dem KundenController die folgende Action hinzu:

```

1  public IActionResult Aendern(int id)
2  {
3      var kunde = kontext.Kunden.First(k => k.KundeId == id);
4      return View(KundenViewModel.AusEntity(kunde));
5  }

```

16

Die Action `Aendern()` erwartet eine Kunden Id als Übergabeparameter und selektiert den Kunden mit dieser Id aus der Datenbank. Danach übergibt sie das gefundene Kunden-Objekt an die statische Methode `AusEntity()` der Klasse `KundenViewModel`,

## 16 ASP.NET MVC – Anwendungen fürs Web

um ein KundenViewModel-Objekt zu erzeugen. Dieses Objekt übergeben wir der Methode `view()` und somit als Model an die zugehörige View. Bevor wir uns um die View ändern kümmern, wollen wir zuerst den Aufruf dieser View unserer Webanwendung hinzufügen. Die URL für diesen Aufruf könnte zum Beispiel wie folgt aussehen:

<http://localhost/Kunden/Andern/2> damit würde man eine View zum Ändern des Kunden mit der Id 2 abrufen. Diesen Aufruf bauen wir in Kundenliste, also die View `Views\Kunden\Index.cshtml`, wie folgt ein:

```
1 @model KundenListeViewModel
2 @{
3     ViewData["Title"] = "Kundenliste";
4 }
5
6 <h1>@ViewData["Title"]</h1>
7
8 <table class="table">
9     <thead>
10        <tr>
11            <th>Kunden-Id</th>
12            <th>Firma</th>
13            <th>Strasse</th>
14            <th>PLZ</th>
15            <th>Ort</th>
16            <th></th>
17        </tr>
18    </thead>
19    <tbody>
20        @foreach (var kunde in Model.Kunden)
21        {
22            <tr>
23                <td>@kunde.KundeId</td>
24                <td>@kunde.Firma</td>
25                <td>@kunde.Strasse</td>
26                <td>@kunde.PLZ</td>
27                <td>@kunde.Ort</td>
28                <td>
29                    <a asp-controller="Kunden" asp-
30                        action="Andern" asp-route-id="@kunde.
31                        KundeId">Ändern</a>
32                </td>
33            </tr>
34        }
35    </tbody>
36 </table>
```

Die Kundenliste erhält eine weitere Spalte. Dazu fügen wir im HTML-Element `<tr>` innerhalb des Elements `<thead>` am Ende ein leeres `<th>`-Element ein. Das `<th>`-Element ist leer da wir für dieses Spalte keine Überschrift ausgeben wollen. Im `<tr>`-Element innerhalb der der `foreach`-Schleife fügen wir am Ende ein `<td>`-Element ein, das den folgenden Link enthält:

## 16.5 Eine eigene View, um Daten zu ändern

```

1 <a asp-controller="Kunden" asp-action="Aendern" asp-route-id="@
2 kunde.KundeId">Ändern</a>

```

Dieser Link ist so konfiguriert, dass er die Action `Aendern()` des Controllers `Kunden` aufruft und ihr den Parameter `id` mit der Id des jeweiligen Kunden übergibt.

Diese View legen wir im Verzeichnis `Views\Kunden` mit dem Namen `Aendern.cshtml` und folgendem Inhalt an:

```

1 @model KundenViewModel
2 @{
3     ViewData["Title"] = "Ändern";
4 }
5
6 <h1>Ändern</h1>
7
8 <h4>Kunde</h4>
9 <hr />
10 <div class="row">
11     <div class="col-md-4">
12         <form asp-controller="Kunden" asp-action="Aendern">
13             <input type="hidden" asp-for="KundeId" />
14             <div class="form-group">
15                 <label class="control-label">Firma:</label>
16                 <input asp-for="Firma" class="form-control" />
17             </div>
18             <div class="form-group">
19                 <label class="control-label">Strasse:</label>
20                 <input asp-for="Strasse" class="form-control" />
21             </div>
22             <div class="form-group">
23                 <label class="control-label">PLZ:</label>
24                 <input asp-for="PLZ" class="form-control" />
25             </div>
26             <div class="form-group">
27                 <label class="control-label">Ort:</label>
28                 <input asp-for="Ort" class="form-control" />
29             </div>
30             <div class="form-group">
31                 <input type="submit" value="Speichern" class="btn
32                     btn-primary" />
33             </div>
34         </form>
35     </div>
36 </div>
37
38 <div>
39     <a asp-action="Index">Zur Kundenliste</a>
40 </div>

```

Die View legt zu Beginn fest, dass sie ein Model vom Typ `KundenViewModel` erwartet. Es folgt ein C#-Block mit der bereits bekannten Technik des Setzens von `ViewData["Title"]`. Dann geben wir noch die Texte Ändern und Kunden als Überschrift und

untergeordnete Überschrift mit den HTML-Elementen `<h1>` und `<h4>` aus. Mit dem Element `<hr/>` sorgen wir für eine horizontale Linie. Als nächstes folgen zwei geschachtelte `<div>`-Elemente. Diese beiden `<div>`-Elemente enthalten die HTML-Attribute `class="row"` und `class="col-md-4"`. Damit sorgen wir dafür, dass das in ihnen enthaltene `<form>`-Element ein Drittel des zur Verfügung stehenden Platz einnimmt. Die beiden HTML-Attribute stammen aus dem HTML/CSS und JavaScript Framework Bootstrap, welches von Visual Studio beim Erzeugen der Webanwendung automatisch eingebunden wird. Leider können wir hier aus Platzgründen nicht näher auf Bootstrap eingehen. Wer sich näher mit diesem Framework beschäftigen möchte, findet unter der folgenden URL einen guten Einstieg:

<https://bmu-verlag.de/bootstrap/>

Innerhalb der beiden `<div>`-Elemente befindet sich ein `<form>`-Element, welches ein sogenanntes HTML-Formular repräsentiert. Alle Aufrufe an den Webserver, die wir bisher kennengelernt haben, waren sogenannte HTTPGET-Aufrufe. Das heißt, wir senden eine URL an den Server und erhalten vom Server eine Antwort in Form einer HTML-Seite. Die Informationen, die wir dabei an den Server übermitteln, sind Bestandteile der URL. Mit einem `<form>`-Element können wir eine weitere Variante eines Serveraufrufs, die HTTPPOST genannt wird, implementieren. Das `<form>`-Element ist über die Attribute `asp-controller` und `asp-action` so konfiguriert, dass es der Action `Aendern()` im Controller `Kunden` zugeordnet wird. Bei dieser Action handelt es sich nicht um die bereits bekannte Action `Aendern()`, sondern um eine weitere Überladung der Action `Aendern()`, die wir im Zuge dieses Kapitels noch behandeln werden. Der Inhalt des `<form>`-Elements beginnt mit einem `<input>`-Element, das die Attribute `type="hidden"` und `asp-for="KundenId"` enthält. Dadurch enthält dieses Formular-Element den Inhalt der Property `KundenId` des Models der View. Allerdings ist das Element für den Benutzer nicht sichtbar, da wir die Id eines Kunden nicht verändern wollen. Danach folgt für jede Property des Models der View ein `<div>`-Element mit dem Attribut `class="form-group"`. Das Attribut stammt ebenfalls aus dem Bootstrap-Framework und dient lediglich der Optik des Formulars. Die `<div>`-Elemente enthalten jeweils ein `<label>`-Element und ein `<input>`-Element. Das `<label>`-Element enthält das Attribut `class="control-label"` und das `<input>`-Element enthält das Attribut `class="form-control"`. Diese beiden Bootstrap-Attribute sind wiederum nur für die Optik zuständig. Das `<label>`-Element enthält eine Beschriftung für das darauffolgende `<input>`-Element. Das `<input>`-Element wird als Eingabefeld am Bildschirm dargestellt. Über ein `asp-for` Attribut ist das `<input>`-Element mit dem Inhalt der jeweiligen Property des Models der View verbunden. Zum Schluss enthält das `<form>`-Element ein weiteres `<div>`-Element mit dem Bootstrap-Attribut `class="form-group"` zur optischen Gestaltung. Das `<div>` Element enthält ein `<input>`-Element mit den Attributen `type="submit"` und `value="Speichern"`. Dadurch wird in unserem Formular eine Schaltfläche mit der Beschriftung „Speichern“ dargestellt. Das Attribut

class="btn btn-primary" dient wiederum nur der Optik. Wenn der Benutzer diese Schaltfläche klickt, wird das Formular mit allen Daten aus den Eingabefeldern an den Server gesendet. In unserem Fall an die Action Ändern() des Controllers Kunden. Als nächstes erweitern wir den KundenController um eine neue Überladung der Action Ändern().

```
1 [HttpPost]
2 public IActionResult
3 Ändern([Bind("KundeId,Firma,Strasse,PLZ,Ort")] KundenViewModel
4 kundeViewModel)
5 {
6     var kunde = kontext.Kunden.FirstOrDefault(k => k.KundeId ==
7 kundeViewModel.KundeId);
8
9     kunde.Firma = kundeViewModel.Firma;
10    kunde.Strasse = kundeViewModel.Strasse;
11    kunde.PLZ = kundeViewModel.PLZ;
12    kunde.Ort = kundeViewModel.Ort;
13
14    kontext.SaveChanges();
15
16    return RedirectToAction(nameof(Index));
17 }
```

Die Action erhält das Methoden-Attribut [HttpPost] um ASP.NET MVC mitzuteilen, dass diese Action über einen HTTPPOST-Aufruf angesprochen wird. Die Action erwartet den Parameter kundeViewModel vom Typ KundenViewModel. Er ist mit dem Attribut [Bind("KundeId,Firma,Strasse,PLZ,Ort")] versehen. Dadurch werden die Properties KundeId, Firma, Strasse, PLZ und Ort des Parameters in dieser Reihenfolge mit den Inhalten der <input>-Elemente im <form>-Element der rufenden View befüllt. Das heißt, die Property KundeId erhält den Inhalt des ersten <input>-Elements, die Property Firma erhält den Inhalt des zweiten <input>-Elements und so weiter. In der Action wird zuerst der Kunde mit der gleichen Id wie der als KundenViewModel-Objekt übergebene Kunde gelesen. Danach werden den Properties dieses Kunden die Inhalte der gleichnamigen Properties des der Action übergebenen KundenViewModel-Objekts zugewiesen. Dann werden die Änderungen mit kontext.SaveChanges() in die Datenbank geschrieben. Zum Schluss der Action rufen wir die vom Controller geerbte Methode RedirectToAction() auf und geben ihr Ergebnis zurück. Der Methode RedirectToAction() übergeben wir den String „Index“. Dadurch rufen wir die Index View des KundenControllers und zeigen somit nach dem Ändern eines Kunden die Kundenliste an.

Zum Schluss dieses Unterkapitels starten wir unsere Webanwendung und überprüfen unsere neue View zum Ändern eines Kunden.

## 16 ASP.NET MVC – Anwendungen für Web

Kundenliste

Kunden-Id	Firma	Strasse	PLZ	Ort
2	Kondit Müller & Söhne	Hauptstrasse 1	54321	Neustadt
3	Bürobedarf Meier	Neue Gasse 5	74851	Mutterhausen

**Abb. 16.5.1** Die Kundeliste mit dem Link Ändern

Wenn wir auf einen Ändern-Link klicken, wird die neue View zum Ändern eines Kunden angezeigt.

Ändern

Kunde

Firma:  
Kondit Müller & Söhne

Strasse:  
[Hauptstrasse 1](#)

PLZ:  
54321

Ort:  
Neustadt

**Speichern**

Zur Kundeliste

**Abb. 16.5.2** Die View zum Ändern eines Kunden

Nach dem Ändern der Straße klicken wir auf die Schaltfläche „Speichern“ und die Kundeliste mit den geänderten Daten wird angezeigt.

**16.6 Übungsaufgabe: Die Webanwendung erweitern.**

Kunden-id	Firma	Strasse	PLZ	Ort
2	Kondit Müller & Söhne	<a href="#">Neue Allee 42</a>	54321	Neustadt
3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen

**Abb. 16.5.3** Die Kundenliste mit den geänderten Daten**16.6 Übungsaufgabe: Die Webanwendung erweitern.**

In dieser Übung werden wir die Kunden-Webanwendung erweitern. Wir werden Funktionen zum Anlegen eines neuen Kunden und zum Löschen eines Kunden hinzufügen.

**Teilaufgabe 1:**

Erweitern sie die View Views\Kunden\Index.cshtml, so dass für jeden Kunden in der Liste ein Link zum Löschen dieses Kunden angezeigt wird. Fügen sie über der Kundenliste einen Link zum Anlegen eines neuen Kunden ein. Verwenden sie für die aufzurufenden Actions den Namen „Neu“ für das Anlegen eines neuen Kunden und den Namen „Loeschen“ für das Löschen eines Kunden.

**Teilaufgabe 2:**

Erweitern Sie den KundenController, um eine Action zum Löschen von Kunden auszulösen. Verwenden Sie für die Action zum Löschen den Namen „Loeschen“. Die Action Loeschen soll nach dem eigentlichen Löschen des Kunden die aktualisierte Kundenliste anzeigen.

16

**Teilaufgabe 3:**

Erweitern Sie den KundenController um eine Action, die ein Formular zum Neuanlegen eines Kunden anzeigt. Verwenden Sie für die Action den Namen „Neu“. Erstellen

Sie die View Views\Kunden\Neu.cshtml, die ein Formular zum Neuanlegen eines Kunden anzeigen. Das Formular soll genauso wie das Formular zum Ändern des Kunden gestaltet sein. Die Eingabefelder für die Daten des Kunden sollen beim Aufruf leer sein. Das Formular soll auch eine Schaltfläche zum Speichern des neuen Kunden enthalten. Erweitern Sie den KundenController um eine weitere Überladung der Action „Neu“, die bei einem Klick auf die Schaltfläche „Speichern“ aufgerufen wird und den neu erfassten Kunden in die Datenbank schreibt. Diese Überladung der Action Neu soll nach dem eigentlichen Anlegen des neuen Kunden die aktualisierte Kundenliste anzeigen.

**Musterlösung für Teilaufgabe 1:**

```
1 @model KundenListeViewModel
2 @{
3     ViewData["Title"] = "Kundenliste";
4 }
5
6 <h1>@ViewData["Title"]</h1>
7
8 <p>
9     <a asp-controller="Kunden" asp-action="Neu">Neuer Kunde</a>
10 </p>
11
12 <table class="table">
13     <thead>
14         <tr>
15             <th>Kunden-Id</th>
16             <th>Firma</th>
17             <th>Strasse</th>
18             <th>PLZ</th>
19             <th>Ort</th>
20             <th></th>
21             <th></th>
22         </tr>
23     </thead>
24     <tbody>
25         @foreach (var kunde in Model.Kunden)
26         {
27             <tr>
28                 <td>@kunde.KundeId</td>
29                 <td>@kunde.Firma</td>
30                 <td>@kunde.Strasse</td>
31                 <td>@kunde.PLZ</td>
32                 <td>@kunde.Ort</td>
33                 <td>
34                     <a asp-controller="Kunden" asp-action="Aendern"
35                         asp-route-id="@kunde.KundeId">Ändern</a>
36                 </td>
37                 <td>
38                     <a asp-controller="Kunden" asp-action="Loeschen"
39                         asp-route-id="@kunde.KundeId">Löschen</a>
40                 </td>
41             </tr>
42         }
43     </tbody>
44 </table>
```

## 16 ASP.NET MVC – Anwendungen für Web

Kunden-Id	Firma	Strasse	PLZ	Ort	Aktionen
2	Karlsruhe Müller & Sohne	Neue Allee 42	54321	Neustadt	<a href="#">Ändern</a> <a href="#">Löschen</a>
3	Bürobedarf Meier	Neue Gasse 5	74851	Musterhausen	<a href="#">Ändern</a> <a href="#">Löschen</a>

Abb. 16.6.1 Links zum neu Anlegen und löschen von Kunden

**Musterlösung für Teilaufgabe 2:**

```

1 public IActionResult Loeschen(int id)
2 {
3     var kunde = kontext.Kunden.First(k => k.KundeId == id);
4     kontext.Kunden.Remove(kunde);
5     kontext.SaveChanges();
6
7     return RedirectToAction("Index");
8 }
```

Kundenliste

Neuer Kunde				
Kunden-Id	Firma	Strasse	PLZ	Ort
2	Kandel Müller & Söhne	Neuer Allee 42	54321	Neustadt
3	Bürobedarf Meier	Neue Gasse 5	74851	Musthausen

© 2020 - Kunden.Web - Privacy

**Abb. 16.6.2** Löschen eines Kunden

Kundenliste

Neuer Kunde				
Kunden-Id	Firma	Strasse	PLZ	Ort
2	Kandel Müller & Söhne	Neuer Allee 42	54321	Neustadt
3	Bürobedarf Meier	Neue Gasse 5	74851	Musthausen

© 2020 - Kunden.Web - Privacy

**Abb. 16.6.3** Der Kunde wurde gelöscht.

**Musterlösung für Teilaufgabe 3**

```

1 public IActionResult Neu()
2 {
3     return View(new KundenViewModel());
4 }
5
6 [HttpPost]
7 public IActionResult Neu([Bind("Firma,Strasse,PLZ,Ort")]
8 KundenViewModel kundeViewModel)
9 {
10     var kunde = new Kunde
11     {
12         Firma = kundeViewModel.Firma,
13         Strasse = kundeViewModel.Strasse,
14         PLZ = kundeViewModel.PLZ,
15         Ort = kundeViewModel.Ort
16     };
17
18     kontext.Kunden.Add(kunde);
19
20     kontext.SaveChanges();
21
22     return RedirectToAction("Index");
23 }
```

```

1 @model KundenViewModel
2 @{
3     ViewData["Title"] = "Neuer Kunde";
4 }
5
6 <h1>Neu</h1>
7
8 <h4>Kunde</h4>
9 <hr />
10 <div class="row">
11     <div class="col-md-4">
12         <form asp-controller="Kunden" asp-action="Neu">
13             <div class="form-group">
14                 <label class="control-label">Firma:</label>
15                 <input asp-for="Firma" class="form-control" />
16             </div>
17             <div class="form-group">
18                 <label class="control-label">Strasse:</label>
19                 <input asp-for="Strasse" class="form-control" />
20             </div>
21             <div class="form-group">
22                 <label class="control-label">PLZ:</label>
23                 <input asp-for="PLZ" class="form-control" />
24             </div>
25             <div class="form-group">
26                 <label class="control-label">Ort:</label>
27                 <input asp-for="Ort" class="form-control" />
28             </div>
29         <div class="form-group">
```

## 16.6 Übungsaufgabe: Die Webanwendung erweitern.

```

30          <input type="submit" value="Speichern" class="btn
31              btn-primary" />
32      </div>
33  </form>
34 </div>
35 </div>
36
37 <div>
38     <a asp-action="Index">Zur Kundenliste</a>
39 </div>

```

Kundenliste

Kunden-ID	Firma	Strasse	PLZ	Ort	
2	Kanzlei Müller & Söhne	Neue Allee 42	54321	Neustadt	<a href="#">Ändern</a> <a href="#">Löschen</a>

Abb. 16.6.4 Anlegen eines neuen Kunden

Neu  
Kunde

Firma:  
Die Bäckerei

Strasse:  
Bäckerstraße 7

PLZ:  
15608

Ort:  
Backhausen

[Speichern](#)

Abb. 16.6.5 Erfassen eines neuen Kunden

## 16 ASP.NET MVC – Anwendungen für Web

Kunden-Id	Firma	Strasse	PLZ	Ort	Ändern	Löschen
2	Kaufhaus Müller & Söhne	Neue Allee 42	54321	Neustadt	<a href="#">Ändern</a>	<a href="#">Löschen</a>
4	Die Bäckerei	Bäckerstraße 7	15688	Borkheide	<a href="#">Ändern</a>	<a href="#">Löschen</a>

Abb. 16.6.6 Der neue erfasste Kunde

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 17

# Grafische Benutzeroberflächen mit WPF

In diesem Kapitel lernen wir wieder einen neuen Projekt-Typ kennen: Die WPF-App. Damit bezeichnet man eine Anwendung mit grafischer Benutzeroberfläche, die unter dem Betriebssystem Windows läuft und auf dem sogenannten Windows Präsentation Framework basiert. Dieses Framework ist ein Teil des .NET-Frameworks, der es uns ermöglicht, sehr leistungsstarke grafische Benutzeroberflächen zu erstellen.

### 17.1 Eine neue WPF-Anwendung erstellen

Um eine neue WPF-App zu erstellen, starten wir Visual Studio und stellen im Dialog „Neues Projekt erstellen“ den Sprachenfilter auf C#, den Plattformfilter auf Windows und den Projekttypfilter auf Desktop.

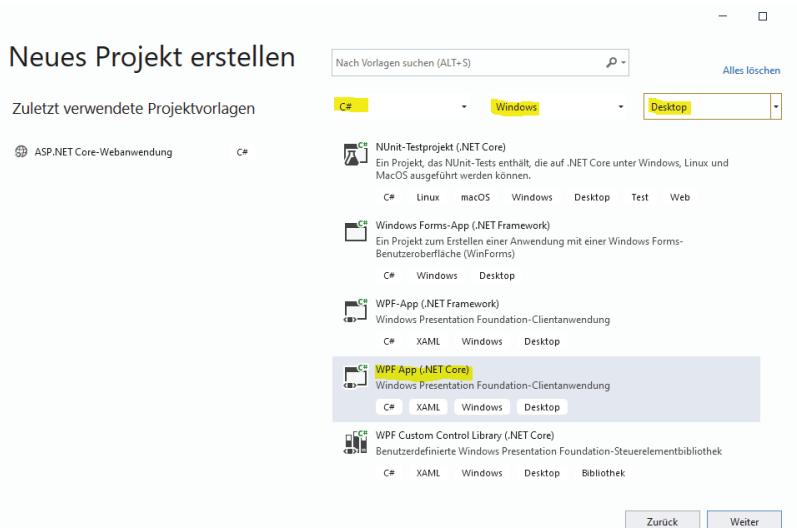


Abb. 17.1.1 Eine neue WPF-App erstellen

Nachdem wir WPF App (.NET Core) ausgewählt haben, klicken wir auf die Schaltfläche „Weiter“. Im darauffolgenden Dialog „Neues Projekt konfigurieren“ vergeben wir den Projektnamen DateiBetrachter und klicken auf die Schaltfläche „Erstellen“.

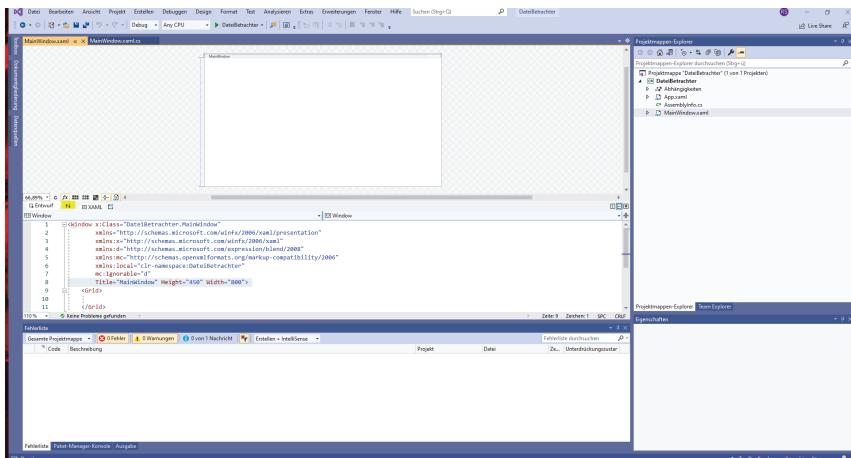
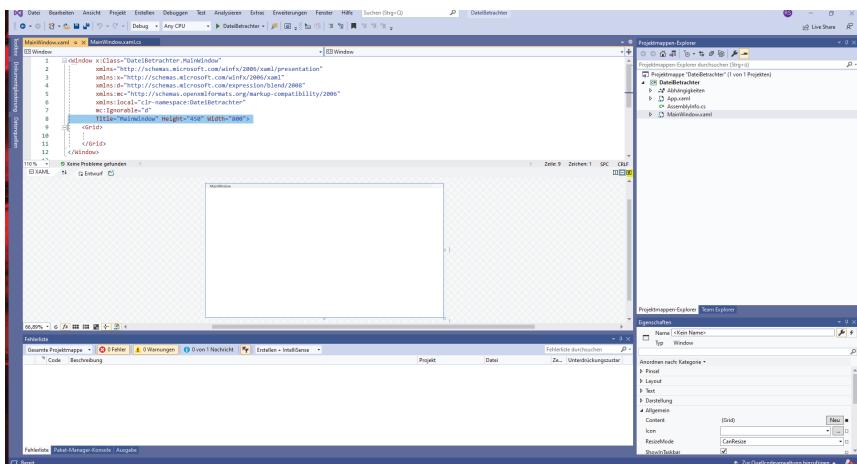


Abb. 17.1.2 Die neue erstellte WPF-App

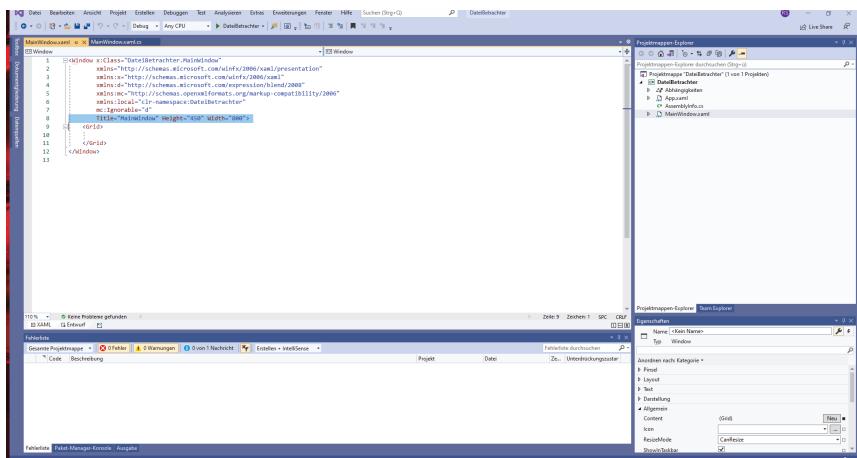
Die neue WPF-App besteht aus zwei Dateien: App.xaml und MainWindow.xaml. Auf diese beiden Dateien werden wir später noch genauer eingehen. Betrachten wir zunächst, wie die Datei MainWindow.xaml im Editor dargestellt wird. Diese Datei repräsentiert das Hauptfenster unserer WPF-App. Im Editor wird sie zweimal angezeigt, einmal als grafische Vorschau und einmal als xml-Datei. Dieses xml wird auch als Extensible Application Markup Language oder kurz XAML bezeichnet. Die grafische Vorschau des Hauptfensters ist nicht nur eine Vorschau, sondern auch ein grafischer Editor für XAML-Dateien. Auf diesen grafischen Editor werden wir aber aus Platzgründen nicht näher eingehen. Zudem ist der grafische Editor lange nicht so hilfreich, wie viele Leser jetzt vermuten werden. In diesem Kapitel über WPF-Apps werden wir uns auf XAML-Dateien in Textform beschränken. Daher stellen wir jetzt den Editor so ein, dass er für XAML-Dateien nur die Textform anzeigt. Dazu klicken wir auf die kleine Schaltfläche mit den beiden vertikalen Pfeilen, die in der Abbildung 17.1-2 gelb markiert ist.

## 17 Grafische Benutzeroberflächen mit WPF



**Abb. 17.1.3** Textdarstellung und Vorschau vertauscht

Die grafische Vorschau und die Textansicht haben jetzt den Platz getauscht. Wenn wir jetzt auf die Schaltfläche mit dem nach unten weisenden Doppelpfeil, der in der Abbildung 17.1-3 markiert ist, klicken, wird die grafische Vorschau des Hauptfensters ausgeblendet.



**Abb. 17.1.4** Die grafische Vorschau ist ausgeblendet

Jetzt wird unsere XAML-Datei nur noch in Textform dargestellt. Zum Schluss wollen wir aus unserer von Visual Studio erstellten WPF-App noch eine „Hello World!“-Anwendung machen. Dazu ändern wir die Datei MainWindow.xaml wie folgt ab:

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3     presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6     blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8     compatibility/2006"
9     xmlns:local="clr-namespace:DateiBetrachter"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12    <Grid>
13        <TextBlock Text="Hello World" />
14    </Grid>
15 </Window>
```

Unsere WPF-App können wir genauso wie unsere anderen Programme auch mit einem Klick auf das bekannte grüne Dreieck in Visual Studio starten.



Abb. 17.1.5 Die „Hello World“-WPF-App

Unsere „Hello World“- WPF-App besteht aus einem Windowsfenster mit dem Titel MainWindow und den klassischen Windows-Schaltflächen, mit denen das Fenster minimiert, maximiert und geschlossen werden kann. Zudem enthält das Fenster den Text „Hello World“.

Im nächsten Unterkapitel werden wir uns näher mit XAML, dem dahinterstehenden Konzept, beschäftigen.

## 17.2 Was ist XAML?

XAML ist ein XML-Vokabular, das in WPF-Apps benutzt wird, um den Aufbau eines grafischen Fensters oder einer grafischen Komponente eines Fensters zu beschreiben. Aber anders als bei HTML wird XAML nicht von einem Webbrower eingelesen und dann interpretiert, sondern XAML wird direkt in C#-Klassen umgesetzt und dann kompiliert, was ihm gegenüber HTML einen signifikanten Geschwindigkeitsvorteil bringt. XAML Elemente entsprechen Klassen. Wenn wir einmal die Datei MainWindow.xaml aus dem vorherigen Kapitel betrachten, sehen wir das folgende Wurzelement:

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3         presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6         blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8         compatibility/2006"
9     xmlns:local="clr-namespace:DateiBetrachter"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12
13 </Window>
```

Der Knoten `<Window>` enthält das Atribut `x:Class= "DateiBetrachter.MainWindow"`. Dadurch wird der XAML-Parser angewiesen, eine Klasse mit dem Namen `MainWindow` im Namensraum `DateiBetrachter` zu erzeugen. Aus dem Namen `Window` des Wurzelements leitet der Parser ab, dass er die Klasse `MainWindow` von der Klasse `Window` ableiten soll. Auf die `xmlns`-Attribute wollen wir hier nicht näher eingehen. Es handelt sich hier um sogenannte xml-Namespaces, die uns erlauben, XML-Knoten und Attribute zu verwenden, die wo anders definiert sind. Im Prinzip sind sie bei XAML so etwas wie die `using`-Anweisungen in C#. Der Wurzelknoten `<Window>` weist außerdem noch den Attributen `Title`, `Height` und `Width` Werte zu. Das Attribut `Title` erhält einen Titel, der im Header unseres Fensters angezeigt wird. Die Attribute `Height` und `Width` erhalten Werte in Pixel für die Höhe und die Breite, die unser Hauptfenster nach dem Start der WPF-App haben wird. `Title`, `Height` und `Width` sind Properties, die die Klasse `MainWindow` von der Klasse `Window` erbt. Die Klasse `Window` vererbt auch die Property `Content` vom Typ `object`. Der Property `Content` kann ein Objekt zugewiesen werden, das den Inhalt des Fensters darstellt. Die Property `Content` ist die sogenannte Default-Property der Klasse `Window` und damit von allen Klassen, die von `Window` abgeleitet sind. Das heißt, wenn wir in XAML einen XML-Knoten als Kind Knoten in den Knoten `Window` setzen, wird die Klasse, die diesem Kind Knoten entspricht, automatisch der Property `Content` zugewiesen. In der Datei `MainWindow.xaml` unserer „Hello World“-WPF-App enthält der Knoten `<Window>` den Kind Knoten `<Grid>`. Die Klasse `Grid` erlaubt uns grafische Elemente innerhalb einer gitterartigen Struktur zu

platzieren. Auf die Verwendung der Klasse Grid werden wir später noch detaillierter eingehen. In unserem ersten Beispiel entspricht die Klasse Grid lediglich einem Gitter mit einer Zeile und einer Spalte. Da ein Gitter mehrere Elemente enthalten kann, hat die Klasse Grid keine Content-Property. Aber sie besitzt die Property Children vom Typ `UIElementCollection`. Hierbei handelt es sich um einen Auflistungstyp, der Elemente vom Typ `UIElement` enthalten kann. Alle grafischen Komponenten, die in einer WPF-App dargestellt werden können, leiten sich entweder direkt oder indirekt von `UIElement` ab. Das heißt, sie können der Children-Property eines Grid-Objekts hinzugefügt werden. Die Children-Property ist die Default-Property der Klasse `Grid`. Daher können in einen `<Grid>`-Knoten in XAML ein oder mehrere grafische Elemente, die von `UIElement` abgeleitet sind, als Kind-Knoten eingefügt werden. In unserem „Hello World“-Beispiel enthält der `<Grid>`-Knoten einen `<TextBlock>`-Knoten. Die Klasse `TextBlock` ist ein einfaches grafisches Element, das einen Text ausgibt. Der auszugebende Text wird dabei der Property `Text` bzw. dem Attribut `Text` des `<TextBlock>`-Knotens zugewiesen. Wenn Visual Studio eine XAML-Datei erstellt, wie zum Beispiel die Datei `MainWindow.xaml`, dann wird immer eine mit der XAML-Datei verbundene C#-Klasse mit erstellt. Diese Klasse ist die sogenannte Code Behind Klasse. Sie erhält den Namen der XAML-Datei mit der zusätzlichen Endung `.cs`, also zum Beispiel `MainWindow.xaml.cs`. Diese Code Behind Datei ist auf den ersten Blick in Visual Studio nicht sichtbar. Wenn wir im Projektmappen-Explorer auf das kleine Dreieck links neben der Datei `MainWindow.xaml` klicken, wird unterhalb der XAML-Datei die Datei `MainWindow.xaml.cs` angezeigt.

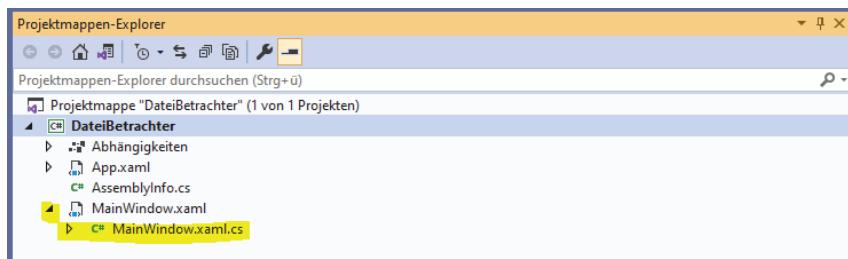


Abb. 17.2.1 Die Code Behind Datei von `MainWindow.xaml`

Diese C#-Klasse kann wie jede andere C#-Klasse mit einem Doppelklick geöffnet werden:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows;
7  using System.Windows.Controls;
8  using System.Windows.Data;
```

## 17 Grafische Benutzeroberflächen mit WPF

```
9  using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace DateiBetrachter
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26         }
27     }
28 }
```

Die Klasse in der Code Behind-Datei befindet sich im Namensraum `DateiBetrachter`, heißt `MainWindow` und ist von der Klasse `Window` abgeleitet, also genauso wie in der XAML-Datei definiert. Auffällig ist, dass vor dem Schlüsselwort `class` das Schlüsselwort `partial` steht und dass im Konstruktor die Methode `InitializeComponent()` aufgerufen wird, obwohl `InitializeComponent()` in der Klasse nicht definiert ist. Wenn Visual Studio eine WPF-App kompiliert, wird aus einer XAML-Datei eine C#-Klasse erzeugt, die genauso heißt, wie in der XAML-Datei angegeben. In unserem Fall heißt die Klasse `MainWindow`. Diese generierte Klasse ist nicht die Code Behind-Datei. Sie ist in Visual Studio auch nicht sichtbar. Diese Klasse ist ebenfalls mit dem Schlüsselwort `partial` deklariert. Wenn eine Klasse als `partial` deklariert wird, kann der Programmcode der Klasse auf mehrere Dateien verteilt werden. Das heißt, die Klasse kann mehrfach mit dem gleichen Namen deklariert werden. Bei jeder Deklaration muss das Schlüsselwort `partial` angegeben werden. Der Compiler erzeugt dann eine Klasse, die alle Properties und alle Methoden aller zusammengehörenden Klassendeklarationen enthält. Mit der Code Behind-Datei können wir die Klasse, die aus der XAML-Datei generiert wird, erweitern. In der Code Behind-Datei können wir auch die einzelnen Komponenten der XAML-Datei zugreifen. Um das an einem Trivialbeispiel zu demonstrieren, ändern wir die Datei `MainWindow.xaml` wie folgt ab.

```
1  <Window x:Class="DateiBetrachter.MainWindow"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3      presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:d="http://schemas.microsoft.com/expression/
6      blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-
8      compatibility/2006"
```

```
9      xmlns:local="clr-namespace:DateiBetrachter"
10     mc:Ignorable="d"
11     Title="MainWindow" Height="450" Width="800">
12     <Grid>
13         <TextBlock x:Name="HalloText" />
14     </Grid>
15 </Window>
```

Wir haben dem Knoten `<TextBlock>` die Zuweisung des Textes „Hello World“ zum Attribut `Text` entfernt und stattdessen haben wir dem Attribut `x:Name` den Namen „HalloText“ zugewiesen. Durch diese Zuweisung erzeugen wir eine Property vom Typ `TextBlock` mit dem Namen `HalloText`, auf die wir in der Code Behind-Datei zugreifen können.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows;
7  using System.Windows.Controls;
8  using System.Windows.Data;
9  using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace DateiBetrachter
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26
27             HalloText.Text = "Hallo Welt";
28         }
29     }
30 }
```

Im Konstruktor weisen wir der Property `Text` von `HalloText` den Wert „Hallo Welt“ zu. Wenn wir unsere WPF-App starten, können wir den Effekt unserer Änderungen überprüfen.



Abb. 17.2.2 Die „Hello World“-WPF-App mit Code Behind-Datei

### 17.3 Wir programmieren einen einfachen Textdateibetrachter

In diesem Unterkapitel wollen wir eine etwas sinnvollere Anwendung als eine „Hello World“-WPF-App vorstellen und dabei weitere Funktionalitäten des WPF-Frameworks kennenlernen. Wir wollen eine WPF-App entwickeln, mit der wir Text-Dateien anzeigen können. Doch zuerst möchte ich noch einen neuen Fachbegriff einführen. Wenn wir bisher über die xml-Knoten einer XAML-Datei gesprochen haben, so haben wir die Begriffe XAML-Knoten oder XAML-Element verwendet. Der eigentliche Fachbegriff ist XAML-Control oder einfach Control. Die Datei MainWindow.xaml unserer „Hello World“-WPF-App enthält ein Window-Control, das ein Grid-Control enthält. Das Grid-Control enthält wiederum ein TextBlock-Control.

Für unseren Dateibetrachter werden wir die WPF-App, die wir im vorherigen Unterkapitel erstellt haben, verwenden und löschen die Zeile:

```
HalloText.Text = "Hallo Welt";
```

aus der Code Behind-Klasse der Datei MainWindow.xaml. Danach ändern wir MainWindow.xaml wie folgt ab:

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3   presentation"
4   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

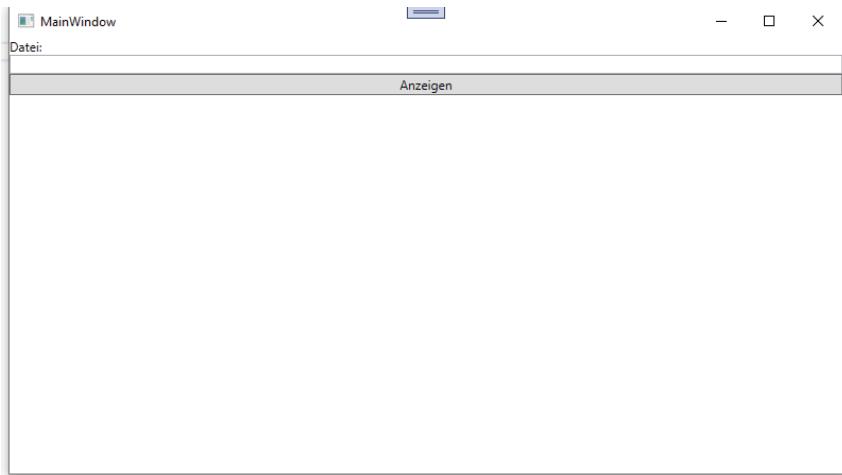
**17.3 Wir programmieren einen einfachen Textdateibetrachter**

```
5      xmlns:d="http://schemas.microsoft.com/expression/
6      blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-
8      compatibility/2006"
9      xmlns:local="clr-namespace:DateiBetrachter"
10     mc:Ignorable="d"
11     Title="MainWindow" Height="450" Width="800">
12     <StackPanel>
13         <TextBlock Text="Datei:"/>
14         <TextBox x:Name="Datei"/>
15         <Button Content="Anzeigen"/>
16         <TextBlock x:Name="Ausgabe"/>
17     </StackPanel>
18 </Window>
```

Unser Hauptfenster besteht aus einem Window-Control, das ein StackPanel-Control enthält. Das StackPanel-Control gehört zu den sogenannten Layout-Controls. Alle Layout Controls dienen als Container für weitere Controls, die vom Layout-Control nach bestimmten Regeln am Bildschirm angezeigt werden. Die Anwendung dieser Regeln kann der Programmierer sowohl durch Setzen von Attributen des Layout-Controls als auch durch Setzen von Attributen der Controls, die sich innerhalb des Layout-Controls befinden, beeinflussen.

Unser StackPanel-Control enthält ein TextBlock-Control, dessen Attribut `Text` auf den Wert „Datei:“ gesetzt wird. Dadurch geben wir den Text „Datei:“ am Bildschirm aus. Danach folgt ein TextBox-Attribut, dessen Attribut `x:Name` auf „Datei“ gesetzt wird. Das TextBox-Attribut dient zur Eingabe eines Dateinamens mit Pfad für die anzuzeigende Datei. Als nächstes folgt ein Button-Control, dessen Attribut `Content` auf „Anzeigen“ gesetzt wird. Dadurch erzeugen wir eine Schaltfläche mit der Beschriftung „Anzeigen“, die das Anzeigen der im TextBox-Control eingegebenen Datei auslösen wird. Da wir weder für das StackPanel-Control noch für die Control-Elemente innerhalb des StackPanel-Controls Attribute für das Layoutverhalten gesetzt haben, wird unser Layout innerhalb des StackPanel-Controls gemäß dem Standard-Layoutverhalten eines StackPanel-Controls erzeugt. Ein StackPanel-Control stellt standardmäßig alle in ihm enthaltenen Controls senkrecht untereinander dar. Wie das aussieht, sehen wir, wenn wir unsere WPF-App starten.

## 17 Grafische Benutzeroberflächen mit WPF



**Abb. 17.3.1** Erste Version der DateiBetrachter Oberfläche

Unsere WPF-App startet als Fenster mit einer Höhe von 450 Pixeln und einer Breite von 800 Pixeln. Das liegt daran, dass die Properties `Height` beziehungsweise `Width` des `Window`-Controls auf 450 beziehungsweise 800 gesetzt sind. Alle Controls innerhalb des `StackPanel`-Controls nehmen die maximale Breite im Fenster ein und nehmen so viel Höhe ein, wie von ihrem Inhalt benötigt wird. Das `StackPanel`-Control besitzt das Attribut `Orientation`, das den Standardwert „Vertical“ enthält, daher hat das `StackPanel`-Control eine vertikale Ausrichtung, wenn wir das Attribut `Orientation` nicht setzen. Als nächstes setzen wir das Attribut `Orientation` für unser `StackPanel`-Control auf `Horizontal`:

```
1 <StackPanel Orientation="Horizontal">
```

und starten die WPF-App.

## 17.3 Wir programmieren einen einfachen Textdateibetrachter

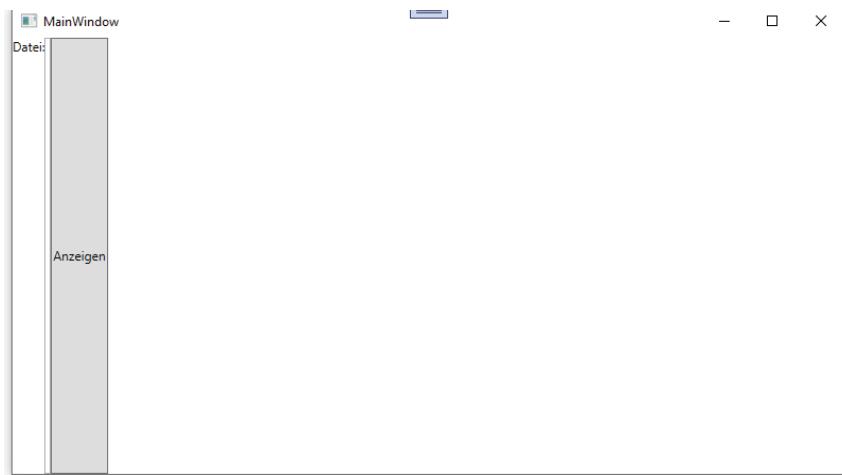


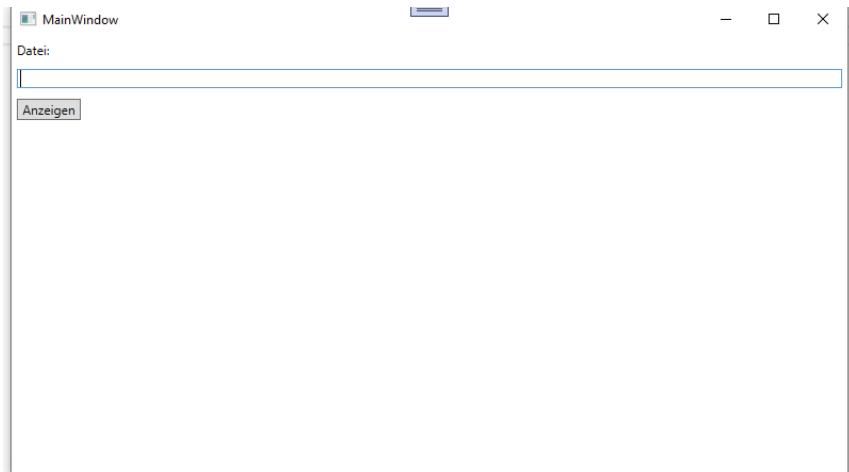
Abb. 17.3.2 Die Dateibetrachter-Oberfläche in horizontaler Ausrichtung

Jetzt nehmen alle Controls innerhalb des StackPanel-Controls die maximale Höhe ein. Sie sind aber nur so breit, wie es für ihren Inhalt notwendig ist. Da diese Variante für unseren Dateibetrachter wenig Sinn macht, entfernen wir das Attribut Orientation wieder aus dem StackPanel-Control. Zusätzlich wollen wir das Design der Benutzeroberfläche unserer WPF-App noch etwas aufhübschen. Dadurch ändert sich die Datei MainWindow.xaml wie folgt:

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3         presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6         blend/2008"
7     xmlns:mcs="http://schemas.openxmlformats.org/markup-
8         compatibility/2006"
9     xmlns:local="clr-namespace:DateiBetrachter"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12    <StackPanel>
13        <TextBlock Text="Datei:" Margin="5"/>
14        <TextBox x:Name="Datei" Margin="5"/>
15        <Button Content="Anzeigen" Width="60"
16            HorizontalAlignment="Left" Margin="5"/>
17        <TextBlock x:Name="Ausgabe" Margin="5"/>
18    </StackPanel>
19 </Window>
```

## 17 Grafische Benutzeroberflächen mit WPF

Alle Controls im StackPanel haben das Attribut `Margin="5"` erhalten. Dadurch erzeugt WPF ein Abstand von 5 Pixel um jedes Control. Die Schreibweise `Margin="5"` ist eine Abkürzung für `Margin="5, 5, 5, 5"`. Damit kann jeder Seite eines Controls ein eigener Abstand zugewiesen werden. Dabei ist die Reihenfolge: links, oben, rechts und unten. Dem Button-Control haben wir mit dem Attribut `Width="60"` eine Breite von 60 Pixeln zugewiesen. Und das Attribut `HorizontalAlignment="Left"` sorgt dafür, dass die Schaltfläche links ausgerichtet ist, ansonsten würde sie in der Mitte des Fensters stehen. Wenn wir unsere WPF-App starten, sehen wir das Ergebnis unserer Änderungen.



**Abb. 17.3.3** Die Dateibetrachter-Oberfläche mit etwas Design

Als nächstes hängen wir an das Button-Control einen sogenannten Click-Eventhandler. Dazu fügen wir dem Button-Control das Attribut `Click` hinzu. Wir tippen dazu nur `Click=` und Visual Studio unterstützt uns mit einem Vorschlag.



**Abb. 17.3.4** Hinzufügen eines Click-Eventhandlers

Nachdem wir die Tab-Taste drücken, erhält unser Button-Control das Attribut `Click="Button_Click"`. Visual Studio hat nicht nur einen Namen für den Click-

**17.3 Wir programmieren einen einfachen Textdateibetrachter**

Eventhandler ausgewählt, sondern auch eine Methode in der Code Behind-Datei MainWindow.xaml.cs erzeugt.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows;
7  using System.Windows.Controls;
8  using System.Windows.Data;
9  using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace DateiBetrachter
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26         }
27
28         private void Button_Click(object sender, RoutedEventArgs
29         e)
30         {
31
32     }
33 }
34 }
```

Die Methode `Button_Click()` wird von WPF immer dann aufgerufen, wenn der Benutzer mit der linken Maustaste auf die Schaltfläche klickt. Die Methode erhält mit dem Parameter `sender` eine Referenz auf das auslösende Objekt. In unserem Fall ist das ein Objekt vom Typ `Button`, welches dem `Button`-Control in unserer XAML-Datei entspricht. Zudem wird der Parameter `e` vom Typ `RoutedEventArgs` übergeben, welcher noch zusätzliche Informationen liefert. Für unser Beispiel benötigen wir diese beiden Parameter nicht. Außerdem würde es den Rahmen sprengen, wenn wir hier das Konzept der Routed Events näher untersuchen würden.

17

Jetzt benötigen wir noch etwas Programmcode, damit die Methode `Button_Click()` die Datei, die im `TextBox`-Control angegeben wird, einliest und im Fenster darstellt. Zuerst binden wir mit `using System.IO` die .NET-Klassenbibliothek für den Dateizugriff ein und dann ändern wir die Methode `Button_Click()` wie folgt:

## 17 Grafische Benutzeroberflächen mit WPF

```
1 private void Button_Click(object sender, RoutedEventArgs e)
2 {
3     try
4     {
5         Ausgabe.Text = File.ReadAllText(Datei.Text);
6     }
7     catch (Exception ex)
8     {
9         Ausgabe.Text = $"Fehler: {ex.Message}";
10    }
11 }
```

Die Methode `Button_Click()` ruft in einem try-Block die statische Methode `ReadAllText()` der Klasse `File` auf und übergibt ihr das Objekt `Datei.Text`. Dabei handelt es sich um die Property `Text` des `TextBox`-Controls, dem wir den Namen `Datei` gegeben haben. Damit übergeben wir als Dateinamen den vom Benutzer eingegebenen Dateinamen. Die Methode `ReadAllText()` gibt den Inhalt der eingelesenen Textdatei zurück und weist ihn an `Ausgabe.Text` zu. Das ist die `Text`-Property des letzten `TextBlock`-Controls in der Datei `MainWindow.xaml`, dem wir den Namen `Ausgabe` gegeben haben. Falls die Methode `ReadAllText()` die angegebene Datei nicht finden kann, wirft sie eine `Exception` aus, die wir in einem catch-Block fangen und dort eine Fehlermeldung an `Ausgabe.Text` zuweisen. Zum Schluss starten wir unsere WPF-App und überprüfen die Funktionalität.

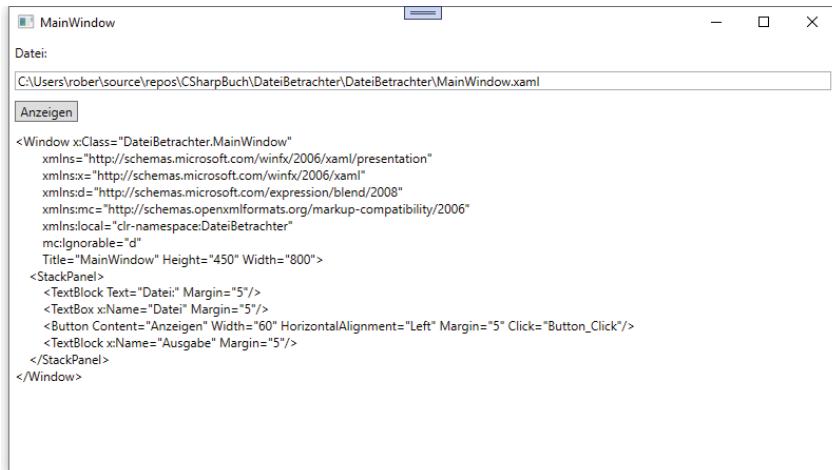


Abb. 17.3.5 Die Datei `MainWindow.xaml` im Dateibetrachter

## 17.4 Organisation der Benutzeroberfläche mit dem Grid-Steuerelement

In diesem Unterkapitel beschäftigen wir uns mit dem Grid-Control. Das Grid-Control ist wie das StackPanel-Control ein Layout-Control. Es ordnet die Controls, die es enthält, in einem tabellenartigen Layout mit Zeilen und Spalten an. Wir hatten das Grid-Control schon in unserer Hello-World WPF-App verwendet. Allerdings hatten wir keinerlei Attribute gesetzt, um das Grid-Control zu konfigurieren. Daher wurden für alle Parameter des Grid-Controls Standardwerte verwendet. Für ein Grid-Control bedeutet das, dass alle Controls innerhalb des Grid-Controls in einer einzigen Zelle angeordnet werden, da ein Grid-Control mit Standardwerten nur eine Zeile und ein Spalte hat. Mit folgenden XAML können wir die Benutzeroberfläche unseres Dateibetrachters mit einem Grid-Control etwas sinnvoller organisieren.

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3     presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6     blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8     compatibility/2006"
9     xmlns:local="clr-namespace:DateiBetrachter"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12    <Grid ShowGridLines="True">
13        <Grid.RowDefinitions>
14            <RowDefinition Height="Auto"/>
15            <RowDefinition Height="*"/>
16        </Grid.RowDefinitions>
17        <Grid.ColumnDefinitions>
18            <ColumnDefinition/>
19            <ColumnDefinition/>
20            <ColumnDefinition/>
21        </Grid.ColumnDefinitions>
22
23        <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:" Margin="5" />
24        <TextBox Grid.Row="0" Grid.Column="1" x:Name="Datei" Margin="5" />
25        <Button Grid.Row="0" Grid.Column="2" Content="Anzeigen" Width="60" HorizontalAlignment="Left" Margin="5" Click="Button_Click"/>
26        <TextBlock Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" x:Name="Ausgabe" Margin="5" />
27    </Grid>
28 </Window>
```

Das Grid-Control erhält das Attribut `ShowGridLines="True"`. Damit sorgen wir dafür, dass die Gitterlinien des Grid-Controls angezeigt werden. Das dient weniger der optischen Gestaltung des Grid-Controls, sondern vielmehr hilft es dem Programmierer, besser zu verstehen, wie sich eine Änderung der Parameter des Grid-Controls auf

## 17 Grafische Benutzeroberflächen mit WPF

das Erscheinungsbild der Benutzeroberfläche auswirkt. Vor der Auslieferung einer WPF-App an die Benutzer sollte dieses Attribut wieder entfernt werden.

Das Grid-Control hat eine Property mit Namen `RowDefinitions` vom Typ `RowDefinitionCollection`. Da dieser Property eine Liste von nicht trivialen Objekten zugewiesen wird, können wir diese Property nicht mit der Attributschreibweise `RowDefinitions="..."` setzen. Wir setzen sie daher mit einer anderen Schreibweise. Wir erstellen innerhalb des Grid.Controls einen XML-Knoten, der den Namen des Controls gefolgt von einem Punkt und dem Namen der Property erhält.

```
1 <Grid.RowDefinitions>
2   <RowDefinition Height="Auto"/>
3   <RowDefinition Height="*"/>
4 </Grid.RowDefinitions>
```

Innerhalb des Knotens `<Grid.RowDefinitions>` stehen die Objekte, die der Property zugewiesen werden. In unserem Fall zwei RowDefinition-Objekte. Damit legen wir fest, dass unser Grid-Control seinen Inhalt in zwei Zeilen organisieren soll. RowDefinition-Objekte haben eine Property namens `Height`. Wir weisen ihr für das erste RowDefinition-Objekt den Wert `Auto` zu und für das zweite RowDefinition-Objekt den Wert `*`. Der Wert `Auto` bedeutet, dass die Höhe der betreffenden Zeile genauso hoch gewählt wird, wie es nötig ist, um den Inhalt der Zeile vollständig darzustellen. Der Wert Stern bedeutet, dass die Höhe der betreffenden Zeile den ganzen restlichen zur Verfügung stehenden Platz erhält. Alternativ könnten wir der `Height`-Property auch einen konkreten Zahlenwert zuweisen, der dann die Höhe der Zeile in Pixel angibt.

Auf die Property `<Grid.RowDefinitions>` folgt die Property `<Grid.ColumnDefinitions>`. Diese Property enthält drei ColumnDefinition-Objekte. Damit legen wir fest, dass unser Grid-Control seinen Inhalt in drei Spalten organisiert. Die ColumnDefinition-Objekte haben eine Property namens `Width`. Diese Property ist für die Breite der jeweiligen Spalte des Grid-Controls zuständig. Sie kann genauso wie die `Height`-Property den Wert `Auto`, `*` oder eine konkrete Pixelangabe erhalten. In unserem Beispiel haben wir die Property `Width` für die drei ColumnDefinition-Objekte nicht gesetzt. Dadurch hat die Property den Standardwert. Dieser ist für die `Width`-Property eines ColumnDefinition-Objekts der Wert `*`. Das heißt, alle drei Spalten erhalten die maximale Breite. Das ist ein Drittel der gesamten für alle drei Spalten zu Verfügung stehenden Breite.

Nach den Properties `<Grid.RowDefinitions>` und `<Grid.ColumnDefinitions>` folgen die Controls, die im Grid-Control dargestellt werden sollen. Alle diese Controls haben die Properties `Grid.Row` und `Grid.Column` gesetzt. Diese Properties sind sogenannte Attached Properties, das heißt Sie sind ursprünglich durch das Grid-Control definiert, können aber auch in anderen Controls wie zum Beispiel `TextBlock`, `TextBox` oder `Button` verwendet werden. Die Proper-

## 17.4 Organisation der Benutzeroberfläche mit dem Grid-Steuerelement

Die Property `Grid.Row` gibt an, in welcher Zeile das betreffende Control platziert wird, wobei die Zählung bei null beginnt. Die Property `Grid.Column` gibt an, in welcher Spalte das Control platziert wird. Auch hier beginnt die Zählung bei null. Das letzte TextBlock-Control im `GridControl` wird in die Zeile 1 und die Spalte 0 gesetzt. Es besitzt eine weitere Attached Property mit Namen `Grid.ColumnSpan`, die auf den Wert 3 gesetzt ist. Dadurch legen wir fest, dass dieses TextBlock-Control in Spalte 0 beginnt und sich über 3 Spalten erstreckt.

Wenn wir die WPF-App starten, sehen wir, wie sich die Benutzeroberfläche durch das Verwenden eines Grid-Controls verändert.

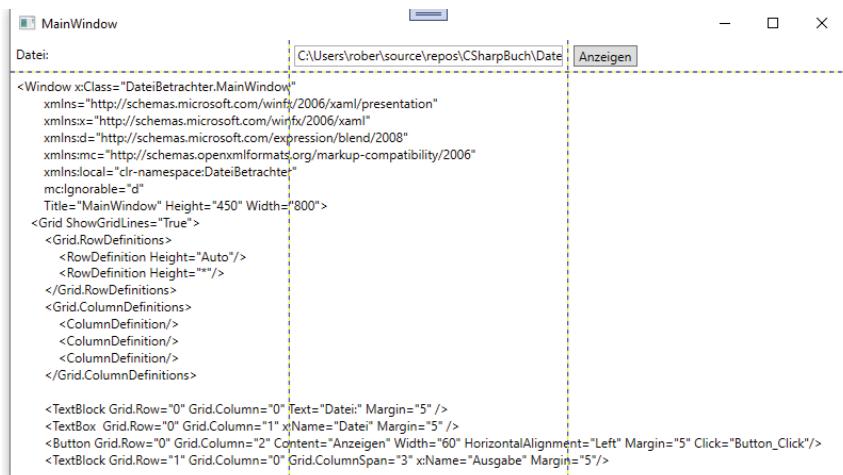


Abb. 17.4.1 Die Dateibetrachter Oberfläche mit Grid-Control

Das Design dieser Benutzeroberfläche hat noch Verbesserungspotential. Wir ändern die Datei `MainWindow.xaml` wie folgt ab, um die Benutzeroberfläche etwas ansehnlicher zu gestalten:

```
1  <Window x:Class="DateiBetrachter.MainWindow"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3          presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:d="http://schemas.microsoft.com/expression/
6          blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-
8          compatibility/2006"
9      xmlns:local="clr-namespace:DateiBetrachter"
10     mc:Ignorable="d"
11     Title="MainWindow" Height="450" Width="800">
12     <Grid>
13         <Grid.RowDefinitions>
```

## 17 Grafische Benutzeroberflächen mit WPF

```
14             <RowDefinition Height="Auto"/>
15             <RowDefinition Height="*"/>
16         </Grid.RowDefinitions>
17         <Grid.ColumnDefinitions>
18             <ColumnDefinition Width="Auto"/>
19             <ColumnDefinition Width="*"/>
20             <ColumnDefinition Width="Auto"/>
21         </Grid.ColumnDefinitions>
22
23         <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:" Margin="5" />
24         <TextBox Grid.Row="0" Grid.Column="1" x:Name="Datei" Margin="5" />
25         <Button Grid.Row="0" Grid.Column="2" Content="Anzeigen" Width="60"
26             HorizontalAlignment="Left" Margin="5"
27             Click="Button_Click"/>
28         <TextBox Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" x:Name="Ausgabe"
29             VerticalScrollBarVisibility="Auto"
30             HorizontalScrollBarVisibility="Auto"
31             IsReadOnly="True"
32             Margin="5" />
33     </Grid>
34 </Window>
```

Zuerst entfernen wir beim Grid-Control das Attribut `ShowGridLines="True"`. Beim ersten und dritten ColumnDefinition-Objekt setzen wir die Property `Width` auf `Auto` und beim zweiten ColumnDefinition-Objekt setzen wir `Width` auf `*`. Dadurch wird die erste Spalte des Grid-Controls, in der sich das TextBlock-Control, das die Beschriftung der Datei anzeigt, befindet, nur so breit, wie es nötig ist, den Text anzuzeigen. Das gleiche gilt für die dritte Spalte, die die Schaltfläche „Anzeigen“ enthält. Die mittlere Spalte, die das TextBox-Control zur Eingabe von Pfad und Dateinamen enthält, erhält den gesamten frei Platz. Das TexBlock-Control zur Anzeige der geladenen Datei ersetzen wir durch das leistungsfähigere TexBox-Control. Dabei übernehmen wir alle Properties des alten TextBlock-Controls. Zusätzlich fügen wir dem TextBox-Control die beiden Attribute `VerticalScrollBarVisibility="Auto"` und `HorizontalScrollBarVisibility="Auto"` hinzu. Dadurch erreichen wir, dass für die TextBox automatisch ein vertikaler und ein horizontaler Scroll-Balken eingeblendet wird, falls die geladene Datei mehr Platz zur Anzeige benötigt als vorhanden ist. Da wir die geladene Datei aber nur anzeigen und nicht verändern wollen, benötigen wir für das TextBox-Control noch das Attribut `IsReadOnly="True"`. Damit verhindern wir, dass der Benutzer die geladene Datei im TextBox-Control verändern kann.

Zum Schluss starten wir unsere WPF-App, um das Ergebnis unserer Änderungen zu überprüfen.

## 17.5 Auswahl einer Datei mit einem FileDialog-Objekt

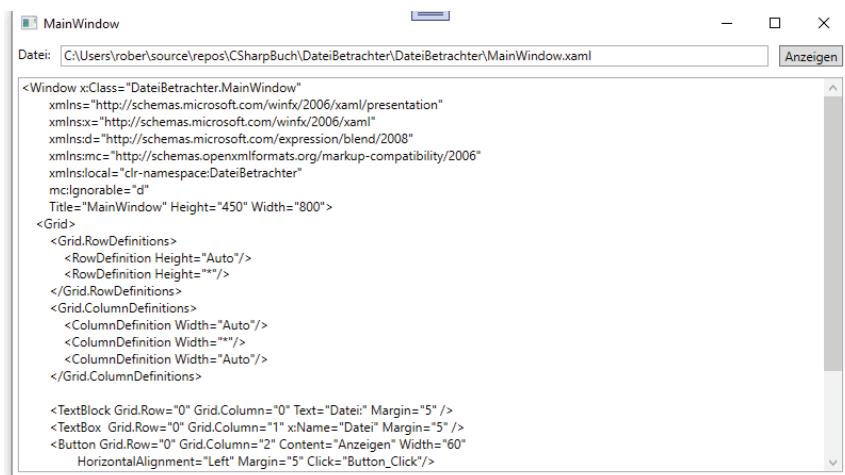


Abb. 17.4.2 Die Dateibetrachter-Oberfläche mit Grid-Control und etwas Design

## 17.5 Auswahl einer Datei mit einem FileDialog-Objekt

Die Benutzeroberfläche unseres Dateibetrachters erwartet vom Benutzer, den Namen und den Pfad einer Datei in einem TextBox-Control einzugeben, um eine Datei betrachten zu können, - für eine Windows-Anwendung ist das aber nicht zeitgemäß. Eine Windows-Anwendung bietet normalerweise einen grafischen Dialog zur Dateiauswahl an. In diesem Unterkapitel werden wir unsere Benutzeroberfläche auch mit einem Dateiauswahldialog ausstatten. Aber zuerst passen wir das Design in MainWindow.xaml wie folgt an:

```

1  <Window x:Class="DateiBetrachter.MainWindow"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3          presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:d="http://schemas.microsoft.com/expression/
6          blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-
8          compatibility/2006"
9      xmlns:local="clr-namespace:DateiBetrachter"
10     mc:Ignorable="d"
11     Title="MainWindow" Height="450" Width="800">
12     <Grid>
13         <Grid.RowDefinitions>
14             <RowDefinition Height="Auto"/>
15             <RowDefinition Height="*"/>
16         </Grid.RowDefinitions>
17         <Grid.ColumnDefinitions>
18             <ColumnDefinition Width="Auto"/>

```

## 17 Grafische Benutzeroberflächen mit WPF

```

19             <ColumnDefinition Width="*"/>
20             <ColumnDefinition Width="Auto"/>
21         </Grid.ColumnDefinitions>
22
23         <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:" Margin="5"/>
24
25         <TextBox Grid.Row="0" Grid.Column="1" x:Name="Datei" Margin="5" IsReadOnly="True"/>
26
27         <Button Grid.Row="0" Grid.Column="2" Content="..." Width="20"
28                 Margin="5" Click="Button_Click"/>
29
30         <TextBox Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" x:Name="Ausgabe" Margin="5"/>
31                 VerticalScrollBarVisibility="Auto"
32                 HorizontalScrollBarVisibility="Auto"
33                 IsReadOnly="True"
34                 Margin="5"/>
35
36     </Grid>
37 </Window>

```

Die TextBox mit dem Namen Datei erhält das Attribut `IsReadOnly="True"`, da wir den Dateinamen nicht mehr selbst tippen wollen, sondern ihn nur noch mit einem Dateiauswahldialog bestimmen wollen. Dass wir die TextBox nicht durch einen TextBlock ersetzen, hat lediglich optische Gründe. Das Content-Attribut des Button-Controls ändern wir zu `Content="..."`. Die Beschriftung mit drei Punkten für eine Schaltfläche, die einen Dateiauswahldialog öffnet, ist unter Windows fast schon ein Quasi-Standard. Eine passende Breite für das Button-Control setzen wir mit `Width="20"`.

In der Code Behind-Datei MainWindow.xaml.cs binden wir die Klassenbibliothek Microsoft.Win32 über eine using-Anweisung ein und ändern die Methode `Button_Click()` wie folgt:

```

1 private void Button_Click(object sender, RoutedEventArgs e)
2 {
3     var fileDialog = new OpenFileDialog();
4
5     if (fileDialog.ShowDialog() == true)
6     {
7         Datei.Text = fileDialog.FileName;
8
9         try
10         {
11             Ausgabe.Text = File.ReadAllText(Datei.Text);
12         }
13         catch (Exception ex)
14         {
15             Ausgabe.Text = $"Fehler: {ex.Message}";
16         }
17     }
18 }

```

## 17.5 Auswahl einer Datei mit einem OpenFileDialog-Objekt

Zuerst erzeugen wir eine neue Instanz der Klasse `OpenFileDialog` und weisen sie der Variablen `fileDialog` zu. Danach rufen wir die Methode `ShowDialog()` von `fileDialog` auf und überprüfen in einer `If`-Verzweigung, ob die Methode `true` zurückgibt. `ShowDialog()` öffnet einen Dateiauswahl dialog. Wird der Dialog mit der Schaltfläche „Öffnen“ beendet, gibt `ShowDialog()` den Wert `true` zurück. Wird der Dialog mit der Schaltfläche „Abbrechen“ beendet, gibt `ShowDialog()` den Wert `false` zurück. Wenn `ShowDialog()` mit der Dialogschaltfläche „Schließen“ (Das kleine X rechts oben im Dialog) beendet wird, gibt `ShowDialog()` den Wert `null` zurück. Wird der Wert `true` zurückgegeben, weisen wir der Property `Text` des TextBox-Controls `Datei` den Wert der Property `fileDialog.FileName` zu. Diese Property enthält den Namen und Pfad der im Dialog gewählten Datei. Dann versuchen wir in einem `try`-Block mit der Anweisung `File.ReadAllText(Datei.Text)` die ausgewählte Datei einzulesen und den Dateiinhalt der Property `Text` der TextBox Ausgabe zuzuweisen. Damit wird die eingelesene Datei im TextBlock Ausgabe angezeigt, sollte beim Einlesen der Datei ein Fehler passieren, so wird eine `Exception` geworfen, die wir in einem Catch-Block fangen, um dann eine Fehlermeldung in den TextBlock Ausgabe zu schreiben.

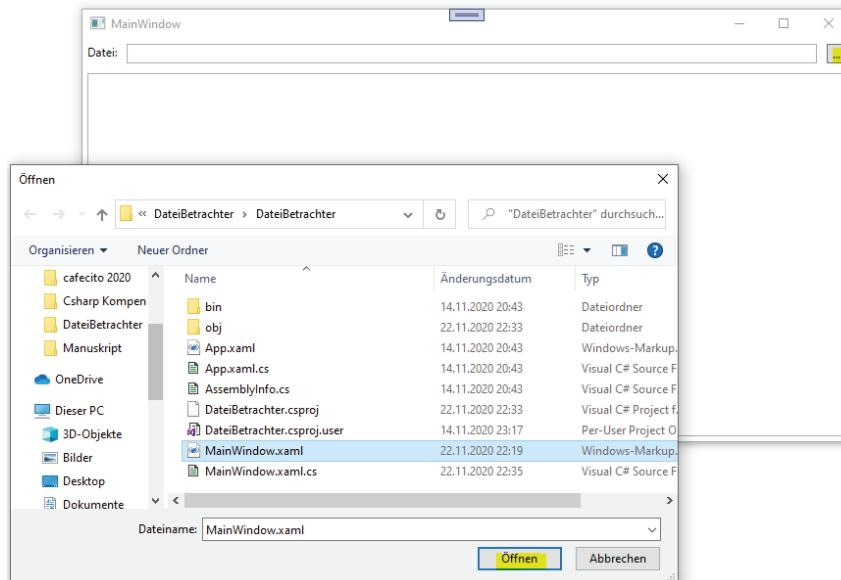


Abb. 17.5.1 Der Dateibetrachter mit Dateiauswahl dialog

## 17.6 Auflistung der Dateien mit einem ListBox-Steuerelement

Nachdem wir die Benutzeroberfläche unseres Dateibetrachters mit Hilfe der Klasse `FileDialog` modernisiert haben, betrachten wir in diesem Unterkapitel ein weiteres WPF-Control, nämlich die `ListBox`. Mit einem `ListBox`-Control kann man gleichstrukturierte Elemente in einer Liste darstellen. Der Benutzer kann dann eines oder mehrere dieser Elemente auswählen und der Programmierer kann auf diese Auswahl reagieren. Die Elemente einer `ListBox` können im einfachsten Fall vom Typ `string` sein. Sie können aber auch von jedem beliebigen anderen C#-Typ sein. Zudem erlaubt uns das `ListBox`-Control die WPF-Konzepte „Data Templates“ und „Data Binding“ einzuführen. Dazu später mehr. Wir wollen in unserem Dateibetrachter nicht mehr eine Datei auswählen und anzeigen, sondern nur ein Dateiverzeichnis. Die Dateien des ausgewählten Verzeichnisses sollen in einem `ListBox`-Control links neben dem `TextBox`-Control für den Dateinhalt dargestellt werden. Klickt der Benutzer auf eine Datei im `ListBox`-Control, soll der Inhalt der Datei im `TextBlock`-Control „Ausgabe“ dargestellt werden. Betrachten wir zunächst die Datei `MainWindow.xaml` für die modifizierte Benutzeroberfläche.

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3         presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6         blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8         compatibility/2006"
9     xmlns:local="clr-namespace:DateiBetrachter"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12    <Grid>
13        <Grid.RowDefinitions>
14            <RowDefinition Height="Auto"/>
15            <RowDefinition Height="*"/>
16        </Grid.RowDefinitions>
17        <Grid.ColumnDefinitions>
18            <ColumnDefinition Width="Auto"/>
19            <ColumnDefinition Width="*"/>
20            <ColumnDefinition Width="Auto"/>
21        </Grid.ColumnDefinitions>
22
23        <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:" Margin="5"/>
24        <TextBox Grid.Row="0" Grid.Column="1" x:Name="Datei" Margin="5" IsReadOnly="True"/>
25        <Button Grid.Row="0" Grid.Column="2" Content="..." Width="20" Margin="5" Click="Button_Click"/>
26
27        <Grid Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" >
28            <Grid.ColumnDefinitions>
29                <ColumnDefinition Width="Auto"/>
```

## 17.6 Auflistung der Dateien mit einem ListBox-Steuerelement

```
34             <ColumnDefinition Width="*"/>
35         </Grid.ColumnDefinitions>
36
37         <ListBox Grid.Row="0" Grid.Column="0"
38             x:Name="Dateiliste"
39                 SelectionChanged="Dateiliste_
40                     SelectionChanged"
41                     Margin="5">
42             <ListBox.ItemTemplate>
43                 <DataTemplate>
44                     <TextBlock Text="{Binding Name}"/>
45                 </DataTemplate>
46             </ListBox.ItemTemplate>
47         </ListBox>
48         <TextBox Grid.Row="0" Grid.Column="1"
49             x:Name="Ausgabe"
50                 VerticalScrollBarVisibility="Auto"
51                 HorizontalScrollBarVisibility="Auto"
52                 IsReadOnly="True"
53                 Margin="5"/>
54
55     </Grid>
56
57 </Grid>
58 </Window>
```

Das erste Grid-Control innerhalb des Window-Controls belassen wir unverändert.

Auch beim TextBlock-Control, beim TextBox-Control und beim Button-Control in der Zeile „erste Zeile“ (`Grid.Row="0"`) des Grid-Controls gibt es keine Änderungen.

In der zweiten Zeile (`Grid.Row="1"`) des Grids platzieren wir in der ersten Spalte (`Grid.Column="0"`) ein weiteres Grid-Control. Mit dem Attribut `Grid.ColumnSpan="3"` sorgen wir dafür, dass sich das innere Grid-Control über drei Spalten erstreckt. Beim inneren Grid-Control setzen wir keine `Grid.RowDefinitions` Property. Damit legen wir fest, dass das innere Grid-Control nur eine Zeile besitzt. Zusätzlich konfigurieren wir für das innere Grid-Control zwei Spalten, wobei die Breite der ersten Spalte auf `Auto` und die Breite der zweiten Spalte wird auf `*` gesetzt wird. In der ersten Spalte des inneren Grid-Controls platzieren wir ein ListBox-Control. Das ListBox-Control erhält den Namen `Dateiliste`. Die Property `Margin` der ListBox wird auf `5` gesetzt, was nur der Optik der Benutzeroberfläche dient. Wenn der Benutzer ein Element der `ListBox` auswählt wird, ähnlich wie beim Click-Eventhandler eines Button-Controls, ein sogenannter `SelectionChanged`-Eventhandler aufgerufen. Diesen Eventhandler legen wir genauso an wie den Click-Eventhandler des Button-Controls. Nur verwenden wir anstelle des Attributs `Click` das Attribut `SelectionChanged`.

Das ListBox-Control hat die Property `ItemTemplate` vom Typ `DataTemplate`. Mit dem XAML-Fragment:

## 17 Grafische Benutzeroberflächen mit WPF

```
1 <ListBox.ItemTemplate>
2     <DataTemplate>
3         <TextBlock Text="{Binding Name}" />
4     </DataTemplate>
5 </ListBox.ItemTemplate>
```

weisen wir der Property `ItemTemplate` ein `DataTemplate`-Objekt zu. Dieses Template wird für jedes Element der `ListBox` verwendet. Innerhalb des `DataTemplate`-Controls platzieren wir ein `TextBlock`-Control. Der Property `Text` des `TextBlock`Controls weisen wir den Ausdruck `{Binding Name}` zu. Damit legen wir fest, dass wir von einem Objekt, das der `ListBox` hinzugefügt wird, erwarten, dass es eine Property hat, die `Name` heißt. Dabei kann die Property `Name` des Objekts einen beliebigen Typ haben, da bei der Darstellung des `TextBlock`-Controls die `ToString()`-Methode aufgerufen wird. Warum wir dieses `DataTemplate` verwenden und nicht einfach String-Objekte der `ListBox` zuweisen, sehen wir später, wenn wir die Änderungen für die Code Behind-Datei `MainWindow.xaml.cs` betrachten.

Mit der bisher verwendeten Klasse `OpenFileDialog` können wir nur Dateien, aber keine Verzeichnisse auswählen. Windows stellt für die Auswahl eines Verzeichnisses einen Dialog zur Verfügung, der `FolderBrowser` heißt. Allerdings suchen wir so einen Dialog in den Klassenbibliotheken von WPF vergeblich. Aber in Windows Forms, einer älteren Technologie, um grafische Benutzeroberflächen zu erstellen, gibt es die Klasse `FolderBrowserDialog`. Diese Klasse kann auch in WPF verwendet werden. Dazu müssen wir unsere Projektdatei etwas umkonfigurieren. Bisher habe wir Konfigurationen in der Projektdatei nur indirekt vorgenommen, indem wir mit der rechten Maustaste auf das Projekt geklickt haben und das Kontextmenü `Eigenschaften` ausgewählt haben. Danach konnten wir mit einer grafischen Benutzeroberfläche Projektkonfigurationen vornehmen. Um Windows Forms Elemente verwenden zu können, müssen wir die Projektdatei direkt editieren. Dazu klicken wir das Projekt mit der linken Maustaste an.

## 17.6 Auflistung der Dateien mit einem ListBox-Steuerelement

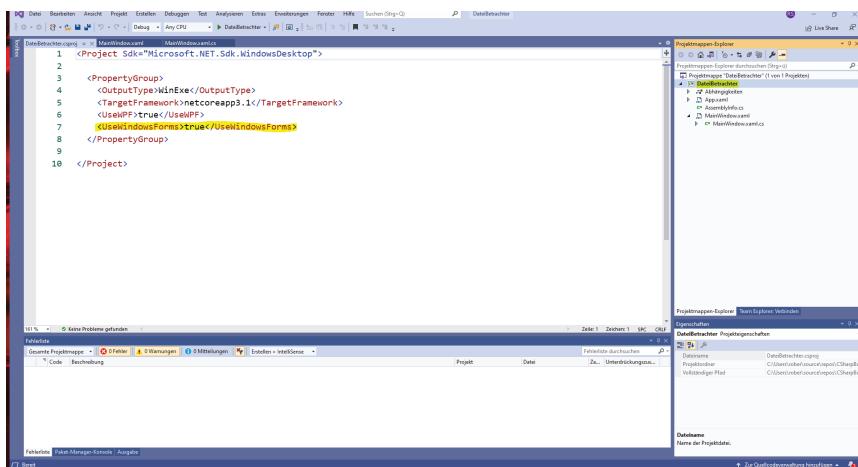


Abb. 17.6.1 Die Projektdatei im Editor

Wie in Abbildung 17.6.1 markiert, fügen wir die Zeile `<UseWindowsForms>true</UseWindowsForms>` in die Projektdatei ein. Dadurch werden die Klassenbibliotheken von Windows Forms in unser Projekt eingebunden und wir können die Klasse `FolderBrowserDialog` verwenden.

Betrachten wir jetzt die Code Behind-Datei `MainWindow.xaml.cs` für unsere modifizierte Dateibetrachter-Anwendung.

```

1  using System;
2  using System.IO;
3  using System.Windows;
4  using System.Windows.Controls;
5  using System.Windows.Forms;
6
7  namespace Dateibetrachter
8  {
9      /// <summary>
10     /// Interaction logic for MainWindow.xaml
11     /// </summary>
12     public partial class MainWindow : Window
13     {
14         public MainWindow()
15         {
16             InitializeComponent();
17         }
18
19         private void Button_Click(object sender, RoutedEventArgs
20         e)
21         {
22             var folderBrowser = new FolderBrowserDialog();

```

## 17 Grafische Benutzeroberflächen mit WPF

```
23
24         var result = folderBrowser.ShowDialog();
25
26         if(result == System.Windows.Forms.DialogResult.OK)
27         {
28             var ordnerInfo = new DirectoryInfo(folderBrowser.
29             SelectedPath);
30             if (ordnerInfo.Exists)
31             {
32                 Dateiliste.Items.Clear();
33                 foreach(var dateiInfo in ordnerInfo.
34                 GetFiles())
35                 {
36                     Dateiliste.Items.Add(dateiInfo);
37                 }
38             }
39         }
40     }
41
42     private void Dateiliste_SelectionChanged(object sender,
43     SelectionChangedEventArgs e)
44     {
45         try
46         {
47             Ausgabe.Text = File.ReadAllText(Dateiliste.
48             SelectedItem.ToString());
49             Datei.Text = Dateiliste.SelectedItem.ToString();
50         }
51         catch(Exception ex)
52         {
53             Ausgabe.Text = ex.Message;
54         }
55     }
56 }
57 }
```

Um die Klasse FolderBrowserDialog verwenden zu können, binden wir mit der Anweisung `using System.Windows.Forms;` die Windows Forms-Klassenbibliothek ein. Die Methode `Button_Click()` wird aufgerufen, wenn der Benutzer auf die Schaltfläche mit den drei Punkten in der Benutzeroberfläche klickt. In der Methode erstellen wir zuerst eine neue Instanz der Klasse `FolderBrowserDialog` und weisen sie der Variablen `folderBrowser` zu. Danach rufen wir die Methode `folderBrowser.ShowDialog()` auf. Die Methode `ShowDialog()` öffnet einen Verzeichnisauswahldialog. Wenn der Benutzer den Dialog beendet, entweder durch Auswahl einer Datei oder mit der Schaltfläche „Abbrechen“ oder durch Schließen des Dialogfensters, endet die Methode `ShowDialog()` und diese gibt ein Ergebnis vom Typ `System.Windows.Forms.DialogResult` zurück, welches wir in der Variablen `result` speichern. `System.Windows.Forms.DialogResult` ist ein enum-Typ, den wir mit seinem vollqualifizierten Namen ansprechen müssen, obwohl wir die Klassenbibliothek `System.Windows.Forms` eingebunden haben. Das liegt daran, dass die Methode `Button_Click()` zu einer Klasse gehört, die von der

## 17.6 Auflistung der Dateien mit einem ListBox-Steuerelement

WPF-Klasse Window abgeleitet ist. Die Klasse Window besitzt eine Property vom Type `bool?`, die ebenfalls DialogResult heißt. Somit würde der Compiler davon ausgehen, dass wir die Property DialogResult der Klasse Window meinen, wenn wir nur DialogResult schreiben würden. Wenn der Benutzer ein Verzeichnis ausgewählt hat, gibt ShowDialog() den Wert `System.Windows.Forms.DialogResult.OK` zurück. Ist dies der Fall, erstellen wir eine neue Instanz der Klasse DirectoryInfo und legen sie in der Variablen `ordnerInfo` ab. Dem Konstruktor der Klasse DirectoryInfo übergeben wir den vollständigen Pfad des ausgewählten Verzeichnisses, welchen wir in `folderBrowser.SelectedPath` finden. Wir überprüfen zunächst, ob das Verzeichnis existiert. Das ist nicht unbedingt nötig, aber falls wir ein Netzwerkverzeichnis auswählen, könnte es theoretisch sein, dass ein anderer Benutzer das Verzeichnis löscht, nachdem wir es ausgewählt haben. Wenn das Verzeichnis existiert, löschen wir mit der Anweisung `Dateiliste.Items.Clear()` zuerst alle Elemente der ListBox, damit die Dateiliste nach einem Wechsel des Verzeichnisses wieder neu erstellt wird. Danach laufen wir durch alle FileInfo-Objekte, die uns die Methode `GetFiles()` der Klasse DirectoryInfo liefern. Innerhalb der Schleife fügen wir jedes FileInfo-Objekt der Struktur `Dateiliste.Items` hinzu, indem wir es der Methode `Add()` übergeben. Die Property Dateiliste enthält das ListBox-Control aus der Datei MainWindow.xaml. Damit fügen wir dem ListBox-Control für jede Datei im ausgewählten Verzeichnis ein FileInfo-Objekt hinzu. Der Property `ItemTemplate` des ListBox-Controls haben wir ein DataTemplate zugewiesen, das ein TextBlock-Control enthält, dessen Text-Property an die Name Property des entsprechenden Listenelements gebunden ist. Ein Listenelement ist jetzt ein FileInfo-Objekt, das über eine Name-Property verfügt, welche den Dateinamen ohne Pfad enthält. Somit enthält unser ListBox-Control FileInfo-Objekte, zeigt aber nur die Dateinamen an.

Um zu verstehen, warum wir das tun, betrachten wir jetzt die Methode: `Dateiliste_SelectionChanged()`.

Diese Methode wird ausgeführt, wenn der Benutzer auf eine Datei in der Liste klickt. In der Methode rufen wir in einem try-Block die statische Methode `File.ReadAllText()` auf. Diese Methode erwartet einen Dateinamen mit Pfadangabe als Übergabeparameter. Wir übergeben ihr `Dateiliste.SelectedItem.ToString()`. Die Property `SelectedItem` des ListBox-Controls enthält das ausgewählte Listenelement und ist vom Typ `FileInfo`. Daher wird die Methode `ToString()` der Klasse `FileInfo` aufgerufen und diese gibt den vollständigen Namen der Datei mit Pfadangabe zurück. Damit schließt sich der Kreis. Das ListBox-Control enthält FileInfo-Objekte. Über ein DataTemplate und einen Binding-Ausdruck wird die Property Name der Klasse `FileInfo` zur Anzeige des Dateinamens verwendet. Über die Methode `ToString()` der Klasse `FileInfo` erhalten wir den vollständigen Namen der Datei, den wir zum Einlesen der Datei benötigen. Die eingelesene Datei weisen wir zur Anzeige der Property `Ausgabe.Text` zu. Zudem weisen wir den vollständigen Namen der Datei mit Pfadangabe der Pro-

## 17 Grafische Benutzeroberflächen mit WPF

roperty Text des TextBox-Controls Datei zu, um ihn in unserer Benutzeroberfläche anzuzeigen. Sollte beim Lesen der Datei ein Fehler auftreten, fangen wir die ausgelöste Exception in einem catch-Block und weisen die Fehlermeldung Ausgabe.Text zu.

Zum Überprüfen der neuen Funktionalität starten wir den Dateibetrachter.

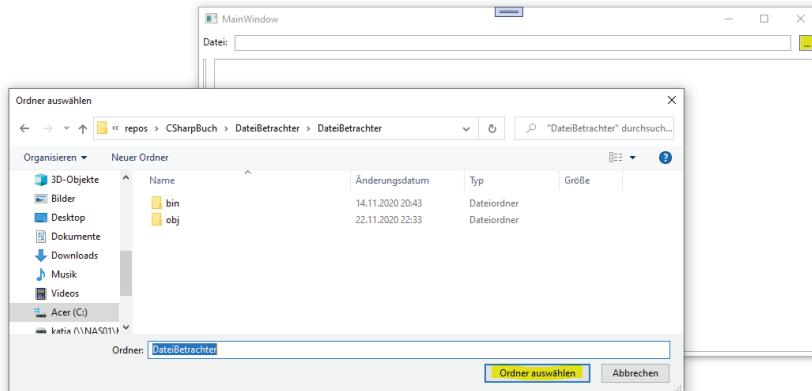


Abb. 17.6.2 Der Dateibetrachter mit Verzeichnisauswahl dialog



Abb. 17.6.3 Der Dateibetrachter mit Dateiauswahl liste

## 17.7 Styling mit Ressourcen

In diesem Unterkapitel beschäftigen wir uns mit dem sogenannten Styling von WPF-Applikationen. Unter Styling versteht man das Beeinflussen des Aussehens von XAML-Controls durch Setzen von Farben, Schriftarten, Schriftgrößen und anderen Werten, die für die Optik eines Controls verantwortlich sind. Des Weiteren betrachten wir Techniken, mit denen man diese Einstellungen so organisieren kann, dass eine Änderung der Optik einer Benutzeroberfläche mit möglichst geringem Aufwand erfolgen kann.

Jedes XAML-Control hat Properties, die gesetzt werden können, um seine optische Erscheinung zu verändern. Zum Beispiel haben die meisten Controls die Property `Background`. Mit `Background="Black"` können wir die Hintergrundfarbe eines Controls auf schwarz setzen. Bei einer komplexen grafischen Benutzeroberfläche mit vielen Controls in vielen Fenstern und wechselnden Fensterinhalten, bei der alle Controls einzeln auf die gleiche Hintergrundfarbe eingestellt sind, ist das Ändern dieser Hintergrundfarbe eine abendfüllende Beschäftigung. WPF stellt uns daher einen eleganteren Weg für das Styling einer Benutzeroberfläche zur Verfügung. Um dieses WPF-Styling zu untersuchen, werden wir Stück für Stück für unseren Dateibetrachter, aus den vorhergehenden Unterkapiteln, eine moderne dunkle Oberfläche entwickeln. Zuerst fügen wir in das Window-Control das folgende XAML-Fragment ein:

```
1 <Window.Resources>
2   <SolidColorBrush x:Key="StandardHintergrund" Color="Black"/>
3   <Style TargetType="Grid">
4     <Setter Property="Background" Value="{StaticResource
5       StandardHintergrund}"/>
6   </Style>
7 </Window.Resources>
```

Das Window-Control hat die Property `Resources`. Diese Property implementiert das Interface `IDictionary` und kann mehrere XAML-Controls enthalten. Die XAML-Controls sind für alle Controls innerhalb des Window-Controls verfügbar. Das erste Control innerhalb von `<Window.Resources>` ist ein `SolidColorBrush`-Control. Diesem Control weisen wir über das Attribut `x:Key` den Bezeichner `StandardHintergrund` zu. Über das Attribut `Color="Black"` stellen wir die Farbe Schwarz ein. Dieses Control können wir uns wie einen Pinsel mit schwarzer Farbe vorstellen. Wenn wir diesen Pinsel einer Property eines Controls zuweisen, die für eine Farbe zuständig ist, stellen wir damit die Farbe ein. Das zweite Control ist ein `Style`-Control. Ein `Style`-Control ist ein Objekt, das Werte für verschiedene Properties bündelt und wenn das `Style`-Control einem anderen Control zugewiesen wird, werden diesem Control alle im `Style`-Control gebündelten Properties auf einmal zugewiesen. Unser `Style`-Control hat die Property `TargetType`, die wir auf den Wert `Grid` setzen. Da wir diesem `Style` für das Attribut `x:Key` keinen Wert zuweisen, gilt dieser `Style` für alle `Grid`-Controls innerhalb des `Window`-Controls, ohne dass wir explicit ein `Style`-Control zuweisen müssen.

## 17 Grafische Benutzeroberflächen mit WPF

Für jede Property, die mit einem Style-Control gesetzt werden soll, enthält das Style-Control ein Setter-Objekt. Ein Setter Objekt hat die beiden Properties `Property` und `Value`. `Property` enthält den Namen der zu setzenden Property und `Value` enthält den zu setzenden Wert. In unserem Fall setzen wir für die Property `Background` den Wert `{StaticResource StandardHintergrund}`. Mit dem Schlüsselwort `StaticResource` können wir auf alle Controls in `<Window.Resources>` zugreifen, die über das Attribut `x:Key` einen Bezeichner haben. In unserem Beispiel verwenden wir das `SolidColorBrush`-Control mit dem Namen `Standardhintergrund` und der Farbe Schwarz. Welchen Effekt dieses Styling auf unsere Benutzeroberfläche hat, sehen wir in der folgenden Abbildung.

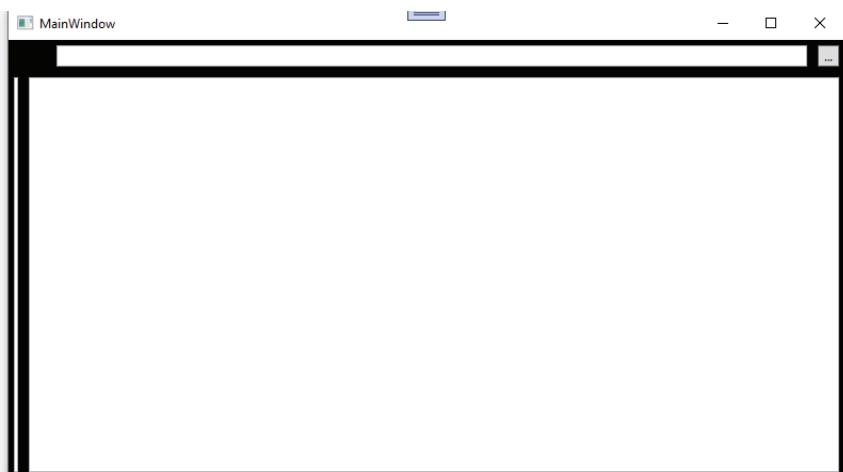


Abb. 17.7.1 Der Dateibetrachter mit schwarzem Hintergrund

Jetzt haben wir zwar einen schwarzen Hintergrund, aber zu einem dunklen Oberflächendesign fehlt noch einiges. Die `TextBox`-Controls und das `ListBox`-Control haben standardmäßig einen weißen Hintergrund. Die Standardhintergrundfarbe für die `TextBlock`-Controls ist transparent, daher hat der Text „Datei:“ die Farbe schwarz vor schwarzem Hintergrund und ist somit unsichtbar. Mit ein paar weiteren Style-Controls können wir unserer Benutzeroberfläche ein dunkles Design geben.

```
1  <Window x:Class="DateiBetrachter.MainWindow"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3          presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:d="http://schemas.microsoft.com/expression/
6          blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-
8          compatibility/2006"
9      xmlns:sys="clr-namespace:System;assembly=System.Runtime"
```

```
10      xmlns:local="clr-namespace:DateiBetrachter"
11      mc:Ignorable="d"
12      Title="MainWindow" Height="450" Width="800">
13  <Window.Resources>
14      <SolidColorBrush x:Key="StandardHintergrund"
15          Color="Black"/>
16      <SolidColorBrush x:Key="StandardSchriftfarbe"
17          Color="#FAFAFA"/>
18      <SolidColorBrush x:Key="ControlHintergrund"
19          Color="#333333" />
20      <SolidColorBrush x:Key="StandardAkzentfarbe"
21          Color="LightGray"/>
22      <FontFamily x:Key="StandardSchrift">Segoe UI</
23      FontFamily>
24      <sys:Double x:Key="NormaleSchrift">12</sys:Double>
25      <sys:Double x:Key="GrosseSchrift">14</sys:Double>
26
27      <Style TargetType="Grid">
28          <Setter Property="Background" Value="{StaticResource
29              StandardHintergrund}"/>
30      </Style>
31
32      <Style TargetType="TextBlock" x:Key="StandardTextBlock">
33          <Setter Property="Foreground" Value="{StaticResource
34              StandardSchriftfarbe}"/>
35          <Setter Property="FontFamily" Value="{StaticResource
36              StandardSchrift}"/>
37          <Setter Property="FontSize" Value="{StaticResource
38              NormaleSchrift}"/>
39      </Style>
40      <Style TargetType="TextBlock" BasedOn="{StaticResource
41          StandardTextBlock}"/>
42      <Style TargetType="TextBlock" x:Key="ListenElement"
43          BasedOn="{StaticResource StandardTextBlock}">
44          <Setter Property="FontSize" Value="{StaticResource
45              GrosseSchrift}"/>
46          <Setter Property="FontWeight" Value="Bold"/>
47      </Style>
48
49      <Style TargetType="TextBox" x:Key="StandartTextBox">
50          <Setter Property="Foreground" Value="{StaticResource
51              StandardSchriftfarbe}"/>
52          <Setter Property="FontFamily" Value="{StaticResource
53              StandardSchrift}"/>
54          <Setter Property="FontSize" Value="{StaticResource
55              NormaleSchrift}"/>
56          <Setter Property="Background" Value="{StaticResource
57              ControlHintergrund}"/>
58          <Setter Property="BorderBrush" Value="{StaticResource
59              StandardAkzentfarbe}"/>
60          <Setter Property="BorderThickness" Value="2"/>
61      </Style>
62      <Style TargetType="TextBox" BasedOn="{StaticResource
63          StandartTextBox}"/>
64      <Style TargetType="TextBox" x:Key="DateiAnzeige"
65          BasedOn="{StaticResource StandartTextBox}">
```

## 17 Grafische Benutzeroberflächen mit WPF

```
66         <Setter Property="FontFamily" Value="Consolas"/>
67     </Style>
68
69     <Style TargetType="ListBox">
70         <Setter Property="Foreground" Value="{StaticResource
71             StandardSchriftfarbe}"/>
72         <Setter Property="FontFamily" Value="{StaticResource
73             StandardSchrift}"/>
74         <Setter Property="FontSize" Value="{StaticResource
75             NormaleSchrift}"/>
76         <Setter Property="Background" Value="{StaticResource
77             ControlHintergrund}"/>
78         <Setter Property="BorderBrush" Value="{StaticResource
79             StandardAkzentfarbe}"/>
80         <Setter Property="BorderThickness" Value="2"/>
81     </Style>
82
83     <Style TargetType="Button">
84         <Setter Property="FontFamily" Value="{StaticResource
85             StandardSchrift}"/>
86         <Setter Property="FontSize" Value="{StaticResource
87             NormaleSchrift}"/>
88         <Setter Property="FontWeight" Value="Bold"/>
89         <Setter Property="BorderBrush" Value="{StaticResource
90             StandardAkzentfarbe}"/>
91         <Setter Property="BorderThickness" Value="2"/>
92     </Style>
93 </Window.Resources>
94 <Grid>
95     <Grid.RowDefinitions>
96         <RowDefinition Height="Auto"/>
97         <RowDefinition Height="*"/>
98     </Grid.RowDefinitions>
99     <Grid.ColumnDefinitions>
100        <ColumnDefinition Width="Auto"/>
101        <ColumnDefinition Width="*"/>
102        <ColumnDefinition Width="Auto"/>
103    </Grid.ColumnDefinitions>
104
105    <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:"
106    Margin="5" />
107    <TextBox Grid.Row="0" Grid.Column="1" x:Name="Datei"
108    Margin="5" IsReadOnly="True"/>
109    <Button Grid.Row="0" Grid.Column="2" Content="..." 
110    Width="20"
111        Margin="5" Click="Button_Click"/>
112
113    <Grid Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" >
114        <Grid.ColumnDefinitions>
115            <ColumnDefinition Width="Auto"/>
116            <ColumnDefinition Width="*"/>
117        </Grid.ColumnDefinitions>
118
119        <ListBox Grid.Row="0" Grid.Column="0"
120            x:Name="Dateiliste"
121            SelectionChanged="Dateiliste_
```

```
122             SelectionChanged"
123             Margin="5">
124         <ListBox.ItemTemplate>
125             <DataTemplate>
126                 <TextBlock Text="{Binding Name}" Style="
127                     {StaticResource ListenElement}"/>
128             </DataTemplate>
129         </ListBox.ItemTemplate>
130     </ListBox>
131     <TextBox Grid.Row="0" Grid.Column="1" x:Name=
132         "Ausgabe" Style="{StaticResource DateiAnzeige}"
133         VerticalScrollBarVisibility="Auto"
134         HorizontalScrollBarVisibility="Auto"
135         IsReadOnly="True"
136         Margin="5"/>
137
138     </Grid>
139
140 </Grid>
141 </Window>
```

Wir fügen `<Window.Resources>` drei weitere Farben mit den `x:Key`-Attributten `StandartSchriftfarbe`, `ControlHintergrund` und `StandartAkzentFarbe` hinzu. Um die Farben für `StandartHintergrund` und `StandartAkzentFarbe` festzulegen, verwenden wir die benannten Farben `Black` und `LightGrey`. Für die Farbe `StandartControlHintergrund` möchten wir ein sehr dunkles Grau und für `StandartSchriftfarbe` ein sehr helles Grau festlegen. Diese beiden Farbtöne finden wir nicht in den benannten Farben. Daher machen wir es wie in einem Farbenfachgeschäft und mischen sie uns selbst. Das sehr dunkle Grau geben wir mit dem Farbcode `#333333` an und für das sehr helle Grau verwenden wir den Farbcode `#FAFAFA`. WPF arbeitet mit einem sogenannten RGB-Farbraum. Dabei stehen die Buchstaben `RGB` für die Farben Rot, Grün und Blau. Ein Farbcode beginnt mit dem Zeichen `#` dem sechs weitere Zeichen folgen. Zwei Zeichen für Rot, zwei Zeichen für Grün und zwei Zeichen für Blau. Die Zeichen stellen Zahlen im Hexadezimalsystem dar, also Werte von `00` bis `FF`, was Zahlen von `0-255` entspricht. Je höher ein Wert, desto höher ist der Anteil der betreffenden Farbe. Wenn wir für alle drei Farben den höchstmöglichen Wert verwenden, ergibt sich der Farbcode `FFFFFF`, der der Farbe Weiß entspricht. Wenn wir für alle drei Farben die kleinste Zahl verwenden, ergibt sich der Farbcode `#000000`, welcher der Farbe Schwarz entspricht. Wenn wir für alle drei Farben den gleichen Wert verwenden, ergeben sich Grautöne. Damit entspricht unser Farbcode `#333333`, wie gewünscht, einem sehr dunklen Grau und der Farbcode `#FAFAFA` ist ein sehr helles Grau.

Mit einem `FontFamily`-Objekt definieren wir uns eine Schriftart und vergeben das `x:Key`-Attribut `StandartSchriftart` und legen diese auf die Schriftart `Segoe UI` fest. Mit `sys:Double`-Objekten legen wir die Schriftgrößen `12` und `14` fest und vergeben dafür die `x:Key`-Attribute `NormaleSchrift` und `GrosseSchrift`. Der Typ `sys:Double` entspricht dem ganz normalen C#-Typ `Double`. Damit wir ihn in XAML verwenden können, müssen wir

## 17 Grafische Benutzeroberflächen mit WPF

den entsprechenden Namensraum deklarieren. Als Bezeichnung für den Namensraum verwenden wir sys. Im Window-Control deklarieren wir den Namensraum mit `xmlns:sys="clr-namespace:System;assembly=System.Runtime"`.

Als nächstes folgt das bekannte Style-Control, mit dem wir für alle Grid-Controls die Hintergrundfarbe auf schwarz setzen. Aber jetzt verwenden wir nicht mehr direkt die benannte Farbe `Black`, sondern die Ressource `StandartHintergrund`.

Danach folgen drei Style-Controls, um das Aussehen der TextBlock-Controls in unserer Benutzeroberfläche festzulegen. Der erste Style für die TextBlock-Controls erhält über das `x:Key`-Attribut die Bezeichnung `StandartTextBlock`. Wir legen für ihn die Attribute `Foreground`, `FontFamily` und `FontSize` fest, indem wir sie auf die bereits definierten Ressourcen `StandartSchriftfarbe`, `StandartSchrift`, und `NormaleSchrift` setzen. Als nächstes folgt das Style-Control `<Style TargetType="TextBlock" BasedOn="{StaticResource StandardTextBlock}" />`. Mit dem Attribut `BasedOn` legen wir fest, dass dieses Style-Control alle Einstellungen des Style-Controls `StandardTextBlock` übernimmt und da wir für dieses Style-Control kein `x:Key`-Attribut vergeben, gelten somit die im Style-Control `StandardTextBlock` festgelegten Einstellungen für alle TextBlock-Controls, denen kein expliziter Style zugewiesen wird. Das letzte der Style-Controls für die TextBlock-Controls erhält den Bezeichner `ListenElement`. Dieses Style-Control weisen wir dem TextBlock-Control im `DataTemplate` für die ListBox-Elemente zu. Über das `BasedOn`-Attribut erhält dieses Style-Control alle Einstellungen von `StandartTextBlock`. Zusätzlich überschreiben wir die `FontSize`-Property und setzen sie auf die Ressource `GrosseSchrift`. Des Weiteren setzen wir die Property `FontWeight` auf den Wert `Bold`, um die Dateinamen in der ListBox fett darzustellen.

Für die TextBox-Controls folgen drei Style-Controls analog zu den Style-Controls für die TextBlock-Control. Dabei deklarieren wir für die TextBox, die den Dateiinhalt darstellt. Sie ist ein eigenes Style-Control, bei dem wir das `x:Key`-Attribut auf `DateiAnzeige` setzen. Bei diesem Style-Control überschreiben wir die Property `FontFamily` und setzen sie auf die Schrift Consolas. Das ist die gleiche Schrift, die Visual Studio zum Anzeigen von Programmcode verwendet.

Als nächstes folgt ein Style Control, mit dem wir die optischen Eigenschaften für das ListBox-Control festlegen, und zum Schluss benötigen wir noch einen Style für das Button-Control.

Beim TextBlock-Control im DataTemplate des ListBox-Control setzen wir die Property `Style` auf den Wert `{StaticResource ListenElement}`, damit hier der spezielle Style `ListenElement` und nicht der allgemeine Style für alle TextBlock-Controls

verwendet wird. Die Property Style für die TextBox Ausgabe setzen wir explicit auf {StaticResource DateiAnzeige}.

Das Ergebnis dieses Stylings sehen wir in der folgenden Abbildung.

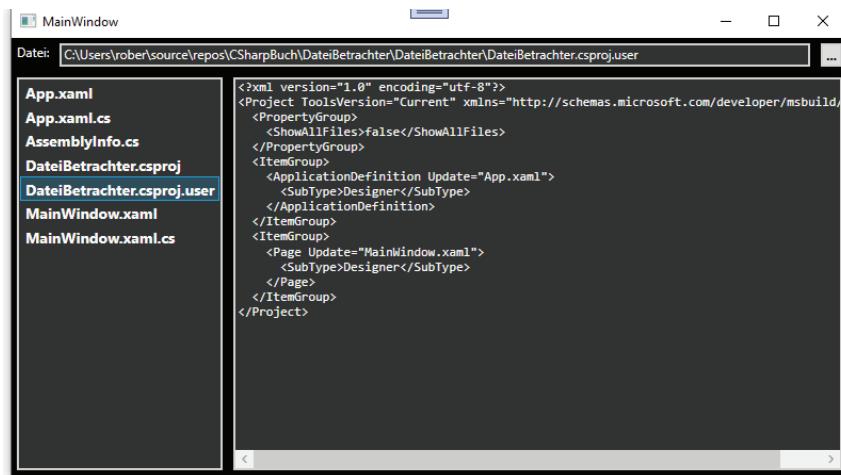
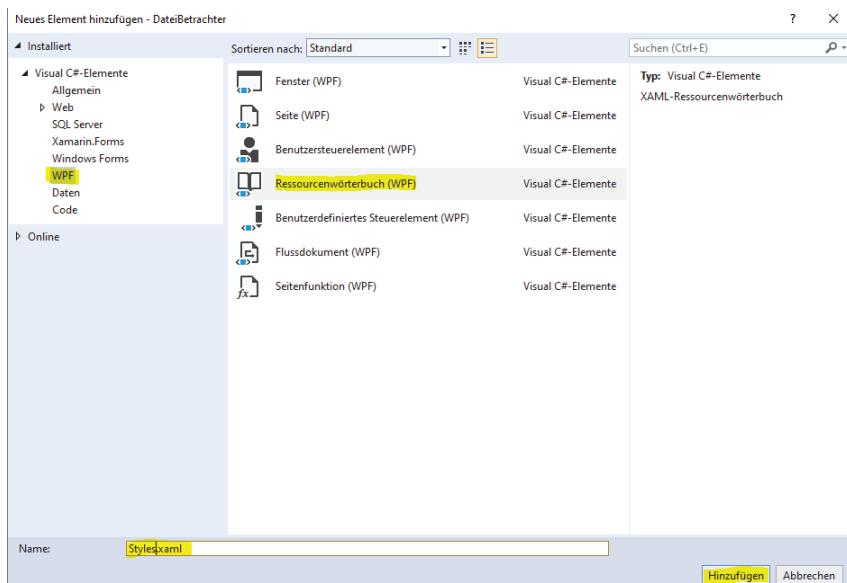


Abb. 17.7.2 Der Dateibetrachter mit dunklem Design

Auf den ersten Blick sieht unser dunkles Design nicht schlecht aus, aber auf den zweiten Blick erkennt man, dass Controls, wenn sie den Fokus erhalten oder wenn man die Maus über sie bewegt, einen blauen Rahmen erhalten, welcher nicht wirklich zu einem dunklen Design passt. WPF bietet zwar über sogenannte Control-Templates die Möglichkeit, auch diese Effekte noch anzupassen, allerdings würde dieses Thema den Rahmen dieses Buches sprengen.

Bei größeren WPF-Projekten kann eine XAML-Datei sehr schnell unübersichtlich werden, daher betrachten wir zum Abschluss dieses Unterkapitels noch eine Möglichkeit, wie wir die Controls in <Window.Resources> in eine eigene Datei auslagern können. Dazu klicken wir mit der rechten Maustaste auf unser Projekt und wählen „Hinzufügen\Neues Element“ aus dem Kontextmenü aus.

## 17 Grafische Benutzeroberflächen mit WPF



**Abb. 17.7.3** Ein Ressourcenwörterbuch zum Projekt hinzufügen

Auf der rechten Seite des Dialogs wählen wir WPF aus und als Element wählen wir Ressourcenwörterbuch (WPF) aus. Als Dateinamen vergeben wir Styles.xaml. Nachdem wir auf die Schaltfläche „Hinzufügen“ geklickt haben, erstellt Visual Studio für uns die Datei Styles.xaml.

```

1 <ResourceDictionary xmlns="http://schemas.microsoft.com/
2 wifx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/
4 wifx/2006/xaml"
5     xmlns:local="clr-namespace:DateiBetrachter">
6
7 </ResourceDictionary>
```

Die Datei enthält ein XAML-Element mit dem Namen `<ResourceDictionary>`. Dieses Element dient uns als Container für alle Controls, die wir von `<Window.Resources>` aus der Datei MainWindow.xaml auslagern wollen. Dann kopieren wir alle Controls aus `<Window.Resources>` in das Element `<ResourceDictionary>` in der Datei Styles.xaml. Danach müssen wir noch den XML-Namespace `xmlns:sys="clr-namespace:System;assembly=System.Runtime"` dem Element `<ResourceDictionary>` hinzufügen. Die vollständige Datei Styles.xaml sieht dann wie folgt aus:

```
1 <ResourceDictionary xmlns="http://schemas.microsoft.com/
2   winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/
4       winfx/2006/xaml"
5     xmlns:sys="clr-namespace: System;
6       assembly=System.Runtime"
7     xmlns:local="clr-namespace:DateiBetrachter">
8   <SolidColorBrush x:Key="StandardHintergrund" Color="Black"/>
9   <SolidColorBrush x:Key="StandardSchriftfarbe"
10    Color="#FAFAFA"/>
11   <SolidColorBrush x:Key="ControlHintergrund" Color="#333333"
12   />
13   <SolidColorBrush x:Key="StandardAkzentfarbe"
14    Color="LightGray"/>
15   <FontFamily x:Key="StandardSchrift">Segoe UI</FontFamily>
16   <sys:Double x:Key="NormaleSchrift">12</sys:Double>
17   <sys:Double x:Key="GrosseSchrift">14</sys:Double>
18
19   <Style TargetType="Grid">
20     <Setter Property="Background" Value="{StaticResource
21       StandardHintergrund}"/>
22   </Style>
23
24   <Style TargetType="TextBlock" x:Key="StandardTextBlock">
25     <Setter Property="Foreground" Value="{StaticResource
26       StandardSchriftfarbe}"/>
27     <Setter Property="FontFamily" Value="{StaticResource
28       StandardSchrift}"/>
29     <Setter Property="FontSize" Value="{StaticResource
30       NormaleSchrift}"/>
31   </Style>
32   <Style TargetType="TextBlock" BasedOn="{StaticResource
33     StandardTextBlock}"/>
34   <Style TargetType="TextBlock" x:Key="ListenElement"
35     BasedOn="{StaticResource StandardTextBlock}">
36     <Setter Property="FontSize" Value="{StaticResource
37       GrosseSchrift}"/>
38     <Setter Property="FontWeight" Value="Bold"/>
39   </Style>
40
41   <Style TargetType="TextBox" x:Key="StandartTextBox">
42     <Setter Property="Foreground" Value="{StaticResource
43       StandardSchriftfarbe}"/>
44     <Setter Property="FontFamily" Value="{StaticResource
45       StandardSchrift}"/>
46     <Setter Property="FontSize" Value="{StaticResource
47       NormaleSchrift}"/>
48     <Setter Property="Background" Value="{StaticResource
49       ControlHintergrund}"/>
50     <Setter Property="BorderBrush" Value="{StaticResource
51       StandardAkzentfarbe}"/>
52     <Setter Property="BorderThickness" Value="2"/>
53   </Style>
54   <Style TargetType="TextBox" BasedOn="{StaticResource
55     StandartTextBox}"/>
56   <Style TargetType="TextBox" x:Key="DateiAnzeige">
```

## 17 Grafische Benutzeroberflächen mit WPF

```

57     BasedOn="{StaticResource StandartTextBox}">
58         <Setter Property="FontFamily" Value="Consolas"/>
59     </Style>
60
61     <Style TargetType="ListBox">
62         <Setter Property="Foreground" Value="{StaticResource
63             StandardSchriftfarbe}"/>
64         <Setter Property="FontFamily" Value="{StaticResource
65             StandardSchrift}"/>
66         <Setter Property="FontSize" Value="{StaticResource
67             NormaleSchrift}"/>
68         <Setter Property="Background" Value="{StaticResource
69             ControlHintergrund}"/>
70         <Setter Property="BorderBrush" Value="{StaticResource
71             StandardAkzentfarbe}" />
72         <Setter Property="BorderThickness" Value="2"/>
73     </Style>
74
75     <Style TargetType="Button">
76         <Setter Property="FontFamily" Value="{StaticResource
77             Standardschrift}"/>
78         <Setter Property="FontSize" Value="{StaticResource
79             NormaleSchrift}"/>
80         <Setter Property="FontWeight" Value="Bold"/>
81         <Setter Property="BorderBrush" Value="{StaticResource
82             StandardAkzentfarbe}" />
83         <Setter Property="BorderThickness" Value="2"/>
84     </Style>
85 </ResourceDictionary>
```

Das Element `<Window.Resources>` in der Datei `MainWindow.xaml` müssen wir natürlich samt Inhalt löschen. Falls wir jetzt versuchen sollten, unsere WPF-App zu starten, erhalten wir einen Fehler, da wir unser Ressourcenwörterbuch bisher nur erstellt, aber noch nicht in unsere Anwendung eingebunden haben. Damit unsere WPF-App das Ressourcenwörterbuch kennt, ändern wir die Datei `App.xaml` wie folgt ab:

```

1 <Application x:Class="DateiBetrachter.App"
2     xmlns="http://schemas.microsoft.com/winfx/2006/
3         xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/
5         xaml"
6     xmlns:local="clr-namespace:DateiBetrachter"
7     StartupUri="MainWindow.xaml">
8
9     <Application.Resources>
10        <ResourceDictionary>
11            <ResourceDictionary.MergedDictionaries>
12                <ResourceDictionary Source="Styles.xaml"/>
13            </ResourceDictionary.MergedDictionaries>
14        </ResourceDictionary>
15    </Application.Resources>
16 </Application>
```

In der Datei `App.Xaml` befindet sich das Element `<Application>`, welches unsere WPF-App repräsentiert. Genauso wie das Control `<Window>` in der Datei

MainWindow.xaml hat es eine Resources-Property. Dieser Property weisen wir ein ResourceDictionary-Element zu. Das ResourceDictionary-Element besitzt eine Property mit Namen MergedDictionaries. Diese Property kann mehrere ResourceDictionary-Elemente enthalten. Ein ResourceDictionary-Element kann auf zwei Arten definiert werden. Entweder man gibt seinen Inhalt direkt an, so wie wir das in der Datei Styles.xaml getan haben, oder aber man weist der Property Source des ResourceDictionary-Elements den Namen und Pfad einer XAML-Datei zu, die das gewünschte ResourceDictionary enthält. Dabei wird der Pfad relativ zum Projektverzeichnis erwartet. In unserem Fall genügt die Angabe Styles.xaml.

## 17.8 Das MVVM-Entwurfsmuster

In den bisherigen Unterkapiteln haben wir gesehen, wie eine WPF-App aufgebaut ist und wie man mit XAML eine Benutzeroberfläche erstellen und stylen kann. Bei umfangreicherer WPF-Apps wird der Programmcode aber sehr schnell sehr unübersichtlich. Daher hat Microsoft ein Entwurfsmuster für WPF-Apps entwickelt, das eng am MVC-Entwurfsmuster, das wir aus dem Kapitel 16 ASP.NET MVC bereits kennen, angelehnt ist. Das Entwurfsmuster nennt sich MVVM. Dabei steht MVVM für Model View ViewModel. Dahinter steckt die Idee, die Code Behind-Datei hinter einer XAML-Datei so gut wie überhaupt nicht zu verwenden. Stattdessen erstellt man für jede XAML-Datei eine sogenannte ViewModel-Klasse. Diese ViewModel-Klasse enthält Properties für alle Eigenschaften der Benutzeroberfläche, die sich irgendwie ändern können. Außerdem enthält das ViewModel alle Methoden, die durch eine Benutzeraktion ausgelöst werden können. In der zugehörigen XAML-Datei werden alle Properties und Methoden für Benutzeraktionen an die Benutzeroberfläche gebunden. Durch das MVVM-Entwurfsmuster erreicht man eine starke Trennung zwischen der Benutzeroberfläche und den Daten einer Anwendung - samt ihrer Verarbeitung. Nach der kurzen theoretischen Einführung betrachten zunächst ein triviales MVVM „Hello World“-Beispiel.

Wir erstellen uns eine neue WPF-App und geben ihr den Namen HelloMVVM. Dann erstellen wir im Projektverzeichnis die Klasse MainWindowViewModel.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace HelloMVVM
6  {
7      public class MainWindowViewModel
8      {
9          public string Begrüßung { get; set; } = "Hello World";
10     }
11 }
```

## 17 Grafische Benutzeroberflächen mit WPF

Die Klasse MainWindowViewModel ist sehr einfach aufgebaut, sie enthält nur die string-Property Begruesung mit dem Standartwert „Hello World“.

Als nächstes ändern wir die Datei MainWindow.xaml wie folgt:

```
1 <Window x:Class="HelloMVVM.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3     presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6     blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8     compatibility/2006"
9     xmlns:local="clr-namespace:HelloMVVM"
10    mc:Ignorable="d"
11    Title="MainWindow" Height="450" Width="800">
12    <StackPanel>
13        <TextBlock Text="{Binding Begruebung}" />
14        <Button Click="Button_Click">Auf Deutsch</Button>
15    </StackPanel>
16 </Window>
```

Auch die Benutzeroberfläche für unser „Hello World“-Beispiel ist sehr einfach aufgebaut. In einem StackPanel befinden sich ein TextBlock- und ein Button-Control. Die Text-Property des TextBlock-Controls ist an die Property Begruebung gebunden. Und der Click-Event des Button-Controls ist an die Methode Button\_Click() gebunden. Dass WPF die Methode Button\_Click() in der zugehörigen Code Behind Datei erwartet, kennen wir bereits. Aber wo wird die Property Begruebung erwartet? Wenn der Binding-Ausdruck keine Information enthält, wo die Property, an die gebunden werden soll, zu finden ist, dann sucht WPF in der Property DataContext des jeweiligen Controls. Jedes Xaml-Control hat eine DataContext-Property vom Typ object. In unserem Beispiel sucht WPF also in der DataContext-Property des TextBlock-Controls. Wenn WPF das TextBlock-Control zur Laufzeit erstellt, und keine explizite Zuweisung für seine DataContext-Property existiert, dann setzt WPF die DataContext-Property auf den Wert der DataContext-Property des Eltern-Controls. In unserem Beispiel also auf die DataContext-Property des StackPanel-Controls. Da bei unserem StackPanel-Control die DataContext-Property auch nicht explizit gesetzt wird, wird sie auf den Wert der DataContext-Property des Window-Controls gesetzt. Damit die DataContext-Property des Window-Controls gesetzt wird, ändern wir die Code Behind-Datei MainWindow.xaml.cs wie folgt ab:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using System.Windows;
7 using System.Windows.Controls;
```

```
8  using System.Windows.Data;
9  using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace HelloMVVM
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26             DataContext = new MainWindowViewModel();
27         }
28     }
29 }
```

Im Konstruktor nach dem Aufruf der Methode `InitializeComponent()` weisen wir der `DataContext`-Property eine neu erzeugte Instanz der Klasse `MainWindowViewModel` zu. Somit befindet sich in der `DataContext`-Property des `TextBlock`-Controls ein Objekt mit einer Property, die `Begrüßung` heißt, an die gebunden werden kann. Damit wir unser Beispiel-Programm ausführen können, fügen wir der Code Behind-Datei noch die Methode `Button_Click()` hinzu.

```
1  private void Button_Click(object sender, RoutedEventArgs e)
2  {
3      var viewModel = (MainWindowViewModel)DataContext;
4      viewModel.Begrüßung = "Hallo Welt";
5  }
```

In der Methode `Button_Click()` führen wir einen Type Cast auf die Property `DataContext` aus und weisen die so erhaltenen Referenz vom Typ `MainWindowViewModel` der Variablen `viewModel` zu. Danach weisen wir der Property `Begrüßung` unseres ViewModels den Wert „Hallo Welt“ zu. Jetzt können wir unser Beispielprogramm starten.

## 17 Grafische Benutzeroberflächen mit WPF

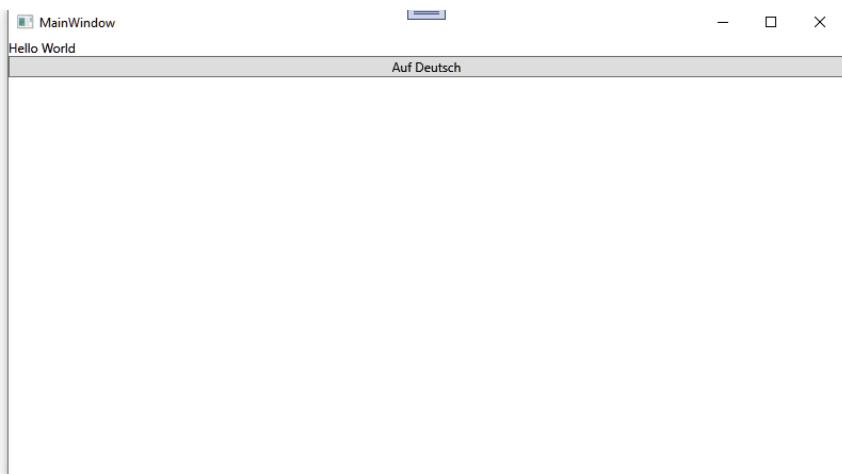


Abb. 17.8.1 Ein TextBlock, der an ein ViewModel gebunden ist

Nach dem Start der WPF-App sehen wir den Effekt der Bindung: Das TextBlock-Control enthält den Inhalt der Property `Begruessung`. Allerdings funktioniert unsere Bindung nur zum Teil. Wenn wir auf das Button Control klicken, rufen wir die Methode `Button_Click()` auf und weisen somit der Property `Begruessung` den Wert „Hallo Welt“ zu, aber das TextBlock-Control ändert sich nicht, obwohl es an die Property `Begruessung` gebunden ist. Das liegt daran, dass bei der Änderung der Property `Begruessung` die Benutzeroberfläche nicht über diese Änderung informiert wird. Damit die Benutzeroberfläche auf die Änderung gebundener Properties reagieren kann, muss unser `ViewModel` das Interface `INotifyPropertyChanged` implementieren, welches von WPF zur Verfügung gestellt wird.

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6
7  namespace HelloMVVM
8  {
9      public class MainWindowViewModel : INotifyPropertyChanged
10     {
11         public event PropertyChangedEventHandler
12             PropertyChanged;
13
14         private string begruessung = "Hello World";
15         public string Begruessung
16         {
17             get
18             {
```

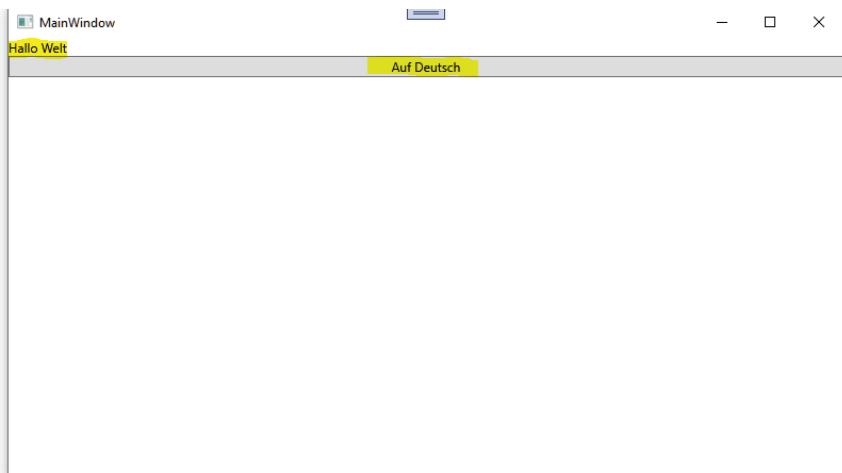
```
19             return begruessung;
20         }
21     set
22     {
23         begruessung = value;
24         OnPropertyChanged();
25     }
26 }
27
28 private void OnPropertyChanged([CallerMemberName] string
29 propertyName = "")
30 {
31     if(PropertyChanged != null)
32     {
33         PropertyChanged(this, new
34         PropertyChangedEventArgs(propertyName));
35     }
36 }
37 }
38 }
```

Die Klasse MainWindowViewModel bindet zunächst die Klassenbibliotheken System.ComponentModel und System.Runtime.CompilerServices ein und implementiert das Interface INotifyPropertyChanged. Dieses Interface verlangt die Deklaration eines sog. Eventhandlers. Der Eventhandler muss den Namen PropertyChanged und den Typ PropertyChangedEventHandler haben. Diese Deklaration erledigen wir mit der Zeile: public event PropertyChangedEventHandler PropertyChanged; Ein Eventhandler ist ein Objekt, bei dem mehrere Methoden, die einer vom Eventhandler vorgegebenen Signaturen entsprechen müssen, registriert werden können. Immer wenn ein bestimmtes Ereignis auftritt, werden alle beim Eventhandler registrierten Methoden ausgeführt. In unserem Anwendungsfall ist das auslösende Ereignis die Änderung einer Property unserer ViewModels. Durch die Bindung einer Property eines Xaml-Controls an eine Property unseres ViewModels, wird eine Methode bei unserem Eventhandler registriert, die bei ihrem Aufruf die gebundene Property des ViewModels aktualisiert. Das Registrieren dieser Methode erledigt die Bindung automatisch. Wir müssen uns aber als Programmierer darum kümmern, dass das Ereignis auch ausgelöst wird. Daher deklarieren wir die Property Begruessung als voll qualifizierte Property so, dass bei der Zuweisung eines Wertes zu dieser Property der Programmcode ausgeführt wird. Im Setter der Property Begruessung rufen wir die private-Methode OnPropertyChanged() auf. Diese Methode überprüft mit dem Ausdruck PropertyChanged!=null, ob beim Eventhandler PropertyChangedMethoden registriert sind. Falls ja, wird mit dem Aufruf PropertyChanged(this, new PropertyChangedEventArgs(propertyName)); der EventHandler ausgelöst. Das Auslösen ist ein Methoden-Aufruf mit dem Namen des EventHandlers, also PropertyChanged, und der Signatur des Eventhandlers. Die Signatur des EventHandlers PropertyChanged erwartet zwei Übergabeparameter. Zuerst einen Parameter vom Typ object.

## 17 Grafische Benutzeroberflächen mit WPF

An dieser Stelle wird das Objekt erwartet, das den EventHandler auslöst. In unserem Fall ist das das ViewModel. Daher übergeben wir das Schlüsselwort this, was dem aktuellen Objekt entspricht. Der zweite Parameter muss ein Objekt vom Typ PropertyChangedEventArgs sein. Das Objekt erstellen wir mit new und übergeben dem Konstruktor einen String mit dem Namen der Property, die den Eventhandler auslöst. Dieser wird vorher der Methode OnPropertyChanged() übergeben. Der Parameter propertyName der Methode OnPropertyChanged() ist optional deklariert und mit dem Attribut [CallerMemberName] versehen. Dieses Attribut sorgt dafür, dass bei einem Aufruf von OnPropertyChanged(), bei dem kein Parameter übergeben wird, der Parameter auf den Namen der Property gesetzt wird, in der OnPropertyChanged() aufgerufen wird. Damit genügt es im Setter von jeder Property des ViewModels einfach OnPropertyChanged() ohne Parameter aufzurufen und der Eventhandler PropertyChanged wird mit dem richtigen Property-Namen aufgerufen.

Wenn wir jetzt unsere WPF-App starten, ändert sich auch das TextBlock-Control, wenn wir auf das Button-Control klicken.



**Abb. 17.8.2** Ein TextBlock reagiert auf eine Änderung im ViewModel

Bis jetzt haben wir nur einen Teil des MVVM-Entwurfsmusters implementiert. Um es vollständig zu implementieren, müssen wir noch die Methode Button\_Click() in unser ViewModel verschieben. Dazu ändern wir das ViewModel wie folgt ab:

```
1 using System;  
2 using System.Collections.Generic;
```

```
3  using System.ComponentModel;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using System.Windows;
7
8  namespace HelloMVVM
9  {
10     public class MainWindowViewModel : INotifyPropertyChanged
11     {
12         public event PropertyChangedEventHandler
13             PropertyChanged;
14
15         private string begruessung = "Hello World";
16         public string Begruebung
17         {
18             get
19             {
20                 return begruessung;
21             }
22             set
23             {
24                 begruessung = value;
25                 OnPropertyChanged();
26             }
27         }
28
29         public void OnButtonClick(object sender, RoutedEventArgs
30 e)
31         {
32             Begruebung = "Hallo Welt";
33         }
34
35         private void OnPropertyChanged([CallerMemberName] string
36 propertyName = "")
37         {
38             if(PropertyChanged != null)
39             {
40                 PropertyChanged(this, new
41                     PropertyChangedEventArgs(propertyName));
42             }
43         }
44     }
45 }
```

Wir haben die public-Methode `OnButtonClick()` in das ViewModel aufgenommen und die Klassenbibliothek `System.Windows` eingebunden. Die Klassenbibliothek benötigen wir damit der Compiler den Typ `RoutedEventArgs` für den Übergabeparameter `e` der Methode `OnButtonClick()` kennt. Die Methode `OnButtonClick()` hat die gleiche Signatur wie die frühere Methode `Button_Click()`. Der Inhalt von `OnButtonClick()` gestaltet sich nun einfacher, da `OnButtonClick()` jetzt Bestandteil des ViewModels ist. Wir weisen lediglich der Property `Begruebung` den Wert „Hallo Welt“ zu. Als nächstes löschen wir die Methode `Button_Click()` aus der Code Behind-Datei und entfernen das Attribut `Click="Button_Click"` von

## 17 Grafische Benutzeroberflächen mit WPF

unserem Button-Control in der Datei MainWindow.xaml. Wir haben jetzt zwar die Funktionalität der Methode `Button_Click()` in das ViewModel verschoben, allerdings müssen wir noch den Aufruf der Methode `OnButtonClick()` beim Click auf das Button-Control implementieren. Dazu müssen wir via NuGet das Paket Microsoft.Xaml.Behaviors.Wpf installieren. Danach ändern wir die Datei MainWindow.xaml wie folgt:

```
1 <Window x:Class="HelloMVVM.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3     presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6     blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-
8     compatibility/2006"
9     xmlns:i="http://schemas.microsoft.com/xaml/behaviors"
10    xmlns:local="clr-namespace:HelloMVVM"
11    mc:Ignorable="d"
12    Title="MainWindow" Height="450" Width="800">
13    <StackPanel>
14        <TextBlock Text="{Binding Begruessung}"/>
15        <Button Content="Auf Deutsch">
16            <i:Interaction.Triggers>
17                <i:EventTrigger EventName="Click">
18                    <i:CallMethodAction
19                        MethodName="OnButtonClick"
20                        TargetObject="{Binding}"/>
21                </i:EventTrigger>
22            </i:Interaction.Triggers>
23        </Button>
24    </StackPanel>
25 </Window>
```

Um das NuGet-Paket `Microsoft.Xaml.Behaviors.Wpf` verwenden zu können, fügen wir dem Window-Control den XML-Namespace `xmlns:i="http://schemas.mircosoft.com/xaml/behaviors"` hinzu. Die Beschriftung des Button-Controls verlagern wir in das Attribut `Content`. Zudem fügen wir dem Button Control die Attached Property `Interaction.Triggers` hinzu. `Interaction.Triggers` ist eine Listenstruktur, die aus Elementen vom Typ `EventTrigger` besteht. In diese Liste nehmen wir ein `EventTrigger`-Element auf. Über das Attribut `EventName="Click"` legen wir fest, dass das `EventTrigger`-Element mit dem Click-Event des Button-Controls verbunden ist. Innerhalb des `EventTrigger`-Elements erstellen wir ein `CallMethodAction`-Element, mit dem wir festlegen, welche Methode gerufen werden soll, wenn das Click-Event des Button-Controls ausgelöst wird. Dem Attribut `MethodName` der `CallMethodAction` weisen wir den Methodennamen `OnButtonClick` zu und dem Attribut `TargetObject` weisen wir den Ausdruck `{Binding}` zu. Im Attribut `TargetObject` wird das Objekt erwartet, das über die Methode verfügt, die im Attribut `MethodName` festgelegt wird. Das heißt an `TargetObject` müssen wir unser ViewModel zuweisen. Der Aus-

druck {Binding} - ohne weitere Parameter - bindet an die Property `DataContext` des betreffenden Elements. In unserem Fall an die Property `DataContext` des `CallMethodAction`-Elements. Da aber die `DataContext`-Property weder für das `CallMethodAction`-Element noch für irgendwelche Elternelemente von `CallMethodAction` gesetzt ist. Wird an die `DataContext`-Property des Window-Controls gebunden, welche in der Code Behind Datei auf das ViewModel gesetzt wird. Damit haben wir die gesamte Funktionalität auf das ViewModel und auf XAML verlagert. Wir weisen jetzt nur noch das ViewModel in der Code Behind-Datei der `DataContext`-Property des Window-Controls zu. Aber selbst diese Zuweisung können wir nach `MainWindow.xaml` übertragen. Dazu löschen wir diese Zuweisung zunächst aus der Code Behind-Datei und fügen in das Window-Element der Datei `MainWindow.xaml` folgendes XAML-Fragment ein:

```
1 <Window.DataContext>
2   <local:MainWindowViewModel/>
3 </Window.DataContext>
```

Das Element `<Window.DataContext>` entspricht der Property `DataContext` des Window-Controls. Visual Studio hat beim Erstellen der Datei `MainWindow.xaml` den XML-Namespace `xmlns:local="clr-namespace:HelloMVVM"` mit erstellt. Da unser ViewModel im Namensraum `HelloMVVM` liegt, erstellen wir mit dem XAML-Element `<local:MainWindowViewModel>` eine Instanz unseres ViewModels und weisen es dem `DataContext` des Window-Controls zu. Damit haben wir keinen eigenen Programmcode mehr in der Code Behind-Datei und unsere Benutzeroberfläche ist nur lose über Bindungen an unser ViewModel gekoppelt. Zum Überprüfen der Funktionalität starten wir unsere WPF-App.

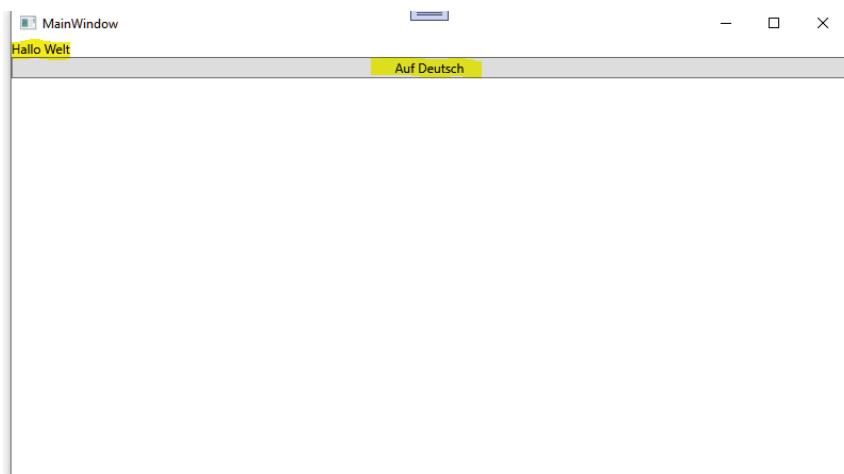


Abb. 17.8.3 Die WPF-App mit vollständigem MVVM-Entwurfsmuster

## 17.9 Übungsaufgabe: Eine WPF-Anwendung auf MVVM umstellen

In dieser Übung werden wir den Dateibetrachter, den wir in den Unterkapiteln 17.3-17.7 entwickelt haben, auf das MVVM-Entwurfsmuster umstellen.

### Teilaufgabe 1:

Erstellen sie die Klasse `ViewModelBase`. Da in komplexeren Anwendungen viele verschiedene ViewModels benötigt werden, die alle das Interface `INotifyPropertyChanged` implementieren müssen, wollen wir diese Aufgabe an eine Basisklasse mit dem Namen `ViewModelBase` delegieren. Diese Klasse implementiert das Interface und stellt die `protected`-Methode  `OnPropertyChanged()` zur Verfügung.

### Teilaufgabe 2:

Erstellen sie die Klasse `DateiListenElementViewModel`. Für die Objekte, die wir dem `ListBox`-Control für die Liste der Dateien hinzufügen, wollen wir ein eigenes ViewModel erstellen. Erstellen Sie dieses ViewModel, das von der in Teilaufgabe 1 erstellten Klasse `ViewModelBase` abgeleitet wird und über die beiden public-Properties `Name` und `VollerName` verfügt. Die Property `Name` soll den Namen einer Datei aufnehmen und die Property `VollerName` soll den vollständigen Namen mit Pfadangabe einer Datei aufnehmen.

### Teilaufgabe 3:

Erstellen Sie die Klasse `MainWindowViewModel`. Diese Klasse verwenden wir als ViewModel für unser Hauptfenster `MainWindow.xaml`. Die Klasse erhält die beiden Properties `Datei` und `Ausgabe`. Die Property `Datei` enthält den vollständigen Namen mit Pfadangabe der ausgewählten Datei und wird **später** an das `TextBox`-Control gebunden, das diesen Namen anzeigt. Die Property `Ausgabe` enthält den Inhalt der ausgewählten Datei und wird **später** an das `TextBlock`-Control zur Anzeige der ausgewählten Datei gebunden.

### Teilaufgabe 4:

Erweitern Sie die Klasse `MainWindowViewModel`. Fügen sie Properties `DateiListe` und `AusgewahlteDatei` hinzu. Die Property `DateiListe` ist vom Typ `ObservableCollection<DateiListenElementViewModel>`. Dieser Typ ist ein generischer Listentyp aus dem Namensraum `System.Collections.ObjectModel`. Er implementiert das Interface `INotifyPropertyChanged` bereits. Zusätzlich implementiert er das Interface `INotifyCollectionChanged`. Dadurch wird ein XAML-Control auch über eine Änderung informiert, wenn ein Element der Liste hinzugefügt oder von der Liste ent-

## 17.9 Übungsaufgabe: Eine WPF-Anwendung auf MVVM umstellen

fernt wird. Daher kann DateiListe als einfache Property, ohne Aufruf der Methode OnPropertyChanged() im Setter, implementiert werden.

Initialisieren Sie die Property DateiListe bereits bei der Deklaration mit einer leeren Listenstruktur.

Die Property AusgewählteDatei ist vom Typ DateiListenElementViewModel, den wir in der Teilaufgabe 2 erstellt haben. Im Setter lesen Sie die ausgewählte Datei ein und weisen deren Inhalt der Property Ausgabe zu. Achten Sie dabei auf die Fehlerbehandlung und weisen Sie der Property Ausgabe eine Fehlermeldung zu, falls das Einlesen der Datei zu einem Fehler führt.

### Teilaufgabe 5:

Erweitern Sie die Klasse MainWindowViewModel um die Methode VerzeichnisAuswahlClick(). Die Methode soll die folgende Signatur haben:

```
public void VerzeichnisAuswahlClick(object sender,  
RoutedEventArgs e)
```

Diese Methode wird **später** an das Click-Ereignis des Button-Controls zum Auswählen eines Verzeichnisses gebunden. Die Methode soll einen FolderBrowserDialog öffnen und nach erfolgter Verzeichnisauswahl mit Hilfe der Methode Clear() von ObservableCollection<T> die Listenstruktur DateiListe leeren und dann mit einer Schleife die Dateien im ausgewählten Verzeichnis durchlaufen und für jede Datei ein Objekt vom Typ DateiListenElementViewModel erzeugen und der Listenstruktur Dateiliste hinzufügen.

### Teilaufgabe 6:

In dieser Teilaufgabe wird stellen Sie die WPF-App DateiBetrachter nun vollständig auf das MVVM-Entwurfsmuster um. Dazu binden Sie die Properties Datei und Ausgabe an das Text-Attribut der zugehörigen TextBox-Controls. Die Property DateiListe wird an das ItemsSource-Attribut des ListBox-Controls gebunden. Die Property AusgewählteDatei binden Sie an das SelectedItem-Attribut des ListBox-Controls. Die Methode VerzeichnisAuswahlClick() binden Sie an das Click-Ereignis des Button-Controls, mit dem ein Verzeichnis ausgewählt wird.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6
7  namespace DateiBetrachter
8  {
9      public class ViewModelBase : INotifyPropertyChanged
10     {
11         public event PropertyChangedEventHandler
12             PropertyChanged;
13
14         protected void OnPropertyChanged([CallerMemberName]
15             string propertyName = "")
16         {
17             if(PropertyChanged != null)
18             {
19                 PropertyChanged(this, new
20                     PropertyChangedEventArgs(propertyName));
21             }
22         }
23     }
24 }
```

**Musterlösung für Teilaufgabe 2:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace DateiBetrachter
6  {
7      public class DateiListenElementViewModel : ViewModelBase
8      {
9          private string name;
10
11         public string Name
12         {
13             get { return name; }
14             set
15             {
16                 name = value;
17                 OnPropertyChanged();
18             }
19         }
20
21         private string vollerName;
22
23         public string VollerName
24         {
25             get { return vollerName; }
26             set
27             {
28                 vollerName = value;
29                 OnPropertyChanged();
30             }
31         }
32     }
33 }
```

**Musterlösung für Teilaufgabe 3:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Windows;
5  using System.Windows.Forms;
6
7  namespace DateiBetrachter
8  {
9      public class MainWindowViewModel : ViewModelBase
10     {
11         private string datei;
12         private string ausgabe;
13
14         public string Datei
15         {
16             get { return datei; }
17             set
18             {
19                 datei = value;
20                 OnPropertyChanged();
21             }
22         }
23
24         public string Ausgabe
25         {
26             get { return ausgabe; }
27             set
28             {
29                 ausgabe = value;
30                 OnPropertyChanged();
31             }
32         }
33     }
34 }
```

**Musterlösung für Teilaufgabe 4:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.IO;
5  using System.Text;
6  using System.Windows;
7  using System.Windows.Forms;
8
9  namespace DateiBetrachter
10 {
11     public class MainWindowViewModel : ViewModelBase
12     {
13         private string datei;
14         private string ausgabe;
15         private DateiListenElementViewModel ausgewahlteDatei;
16
17         public string Datei
18         {
19             get { return datei; }
20             set
21             {
22                 datei = value;
23                 OnPropertyChanged();
24             }
25         }
26
27         public string Ausgabe
28         {
29             get { return ausgabe; }
30             set
31             {
32                 ausgabe = value;
33                 OnPropertyChanged();
34             }
35         }
36
37         public ObservableCollection<DateiListenElementViewModel>
38         DateiListe { get; set; } = new ObservableCollection<
39         DateiListenElementViewModel>();
40
41         public DateiListenElementViewModel AusgewahlteDatei
42         {
43             get { return ausgewahlteDatei; }
44             set
45             {
46                 ausgewahlteDatei = value;
47                 try
48                 {
49                     Ausgabe = File.ReadAllText(AusgewahlteDatei.
50                     VollerName);
51                     Datei = AusgewahlteDatei.VollerName;
52                 }
53                 catch (Exception ex)
54                 {
```

## 17 Grafische Benutzeroberflächen mit WPF

```
55             Ausgabe = ex.Message;
56         }
57     OnPropertyChanged();
58 }
59 }
60 }
61 }
```

**Musterlösung für Teilaufgabe 5:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.IO;
5  using System.Text;
6  using System.Windows;
7  using System.Windows.Forms;
8
9  namespace DateiBetrachter
10 {
11     public class MainWindowViewModel : ViewModelBase
12     {
13         private string datei;
14         private string ausgabe;
15         private DateiListenElementViewModel ausgewahlteDatei;
16
17         public string Datei
18         {
19             get { return datei; }
20             set
21             {
22                 datei = value;
23                 OnPropertyChanged();
24             }
25         }
26
27         public string Ausgabe
28         {
29             get { return ausgabe; }
30             set
31             {
32                 ausgabe = value;
33                 OnPropertyChanged();
34             }
35         }
36
37         public ObservableCollection<DateiListenElementViewModel>
38         DateiListe { get; set; } = new ObservableCollection<
39         DateiListenElementViewModel>();
40
41         public DateiListenElementViewModel AusgewahlteDatei
42         {
43             get { return ausgewahlteDatei; }
44             set
45             {
46                 ausgewahlteDatei = value;
47                 try
48                 {
49                     Ausgabe = File.ReadAllText(AusgewahlteDatei.
50                     VollerName);
51                     Datei = AusgewahlteDatei.VollerName;
52                 }
53             }
54         }
55     }
56 }
```

## 17 Grafische Benutzeroberflächen mit WPF

```
53             catch (Exception ex)
54             {
55                 Ausgabe = ex.Message;
56             }
57
58             OnPropertyChanged();
59         }
60     }
61
62     public void VerzeichnisAuswahlClick(object sender,
63                                         RoutedEventArgs e)
64     {
65         var folderBrowser = new FolderBrowserDialog();
66
67         var result = folderBrowser.ShowDialog();
68
69         if (result == System.Windows.Forms.DialogResult.OK)
70         {
71
72             var ordnerInfo = new DirectoryInfo(folderBrowser.
73 SelectedPath);
74             if (ordnerInfo.Exists)
75             {
76                 DateiListe.Clear();
77                 foreach (var dateiInfo in ordnerInfo.
78 GetFiles())
79                 {
80                     var dateiListenElement = new
81 DateiListenElementViewModel
82                     {
83                         Name = dateiInfo.Name,
84                         VollerName = dateiInfo.ToString()
85                     };
86
87                     DateiListe.Add(dateiListenElement);
88                 }
89             }
90         }
91     }
92 }
93 }
```

**Musterlösung für Teilaufgabe 6:**

```
1 <Window x:Class="DateiBetrachter.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
3         presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:d="http://schemas.microsoft.com/expression/
6         blend/2008"
7     xmlns:mcs="http://schemas.openxmlformats.org/markup-
8         compatibility/2006"
9     xmlns:i="http://schemas.microsoft.com/xaml/behaviors"
10    xmlns:sys="clr-namespace:System;assembly=System.Runtime"
11    xmlns:local="clr-namespace:DateiBetrachter"
12    mc:Ignorable="d"
13    Title="MainWindow" Height="450" Width="800">
14    <Window.DataContext>
15        <local:MainWindowViewModel/>
16    </Window.DataContext>
17    <Grid>
18        <Grid.RowDefinitions>
19            <RowDefinition Height="Auto"/>
20            <RowDefinition Height="*"/>
21        </Grid.RowDefinitions>
22        <Grid.ColumnDefinitions>
23            <ColumnDefinition Width="Auto"/>
24            <ColumnDefinition Width="*"/>
25            <ColumnDefinition Width="Auto"/>
26        </Grid.ColumnDefinitions>
27
28        <TextBlock Grid.Row="0" Grid.Column="0" Text="Datei:" Margin="5" />
29        <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Datei}" Margin="5" IsReadOnly="True"/>
30        <Button Grid.Row="0" Grid.Column="2" Content="..." Width="20" Margin="5">
31            <i:Interaction.Triggers>
32                <i:EventTrigger EventName="Click">
33                    <i:CallMethodAction TargetObject="{Binding}" MethodName="VerzeichnisAuswahlClick"/>
34                </i:EventTrigger>
35            </i:Interaction.Triggers>
36        </Button>
37
38        <Grid Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" >
39            <Grid.ColumnDefinitions>
40                <ColumnDefinition Width="Auto"/>
41                <ColumnDefinition Width="*"/>
42            </Grid.ColumnDefinitions>
43
44            <ListBox Grid.Row="0" Grid.Column="0" ItemsSource="{Binding DateiListe}" SelectedItem="{Binding AusgewahlteDatei}" Margin="5">
45                <ListBox.ItemTemplate>
```

## 17 Grafische Benutzeroberflächen mit WPF

```
55      <DataTemplate>
56          <TextBlock Text="{Binding Name}"
57              Style="{StaticResource ListenElement}"/>
58      </DataTemplate>
59  </ListBox.ItemTemplate>
60 </ListBox>
61 <TextBox Grid.Row="0" Grid.Column="1"
62     Text="{Binding Ausgabe}"
63     Style="{StaticResource DateiAnzeige}"
64     VerticalScrollBarVisibility="Auto"
65     HorizontalScrollBarVisibility="Auto"
66     IsReadOnly="True"
67     Margin="5"/>
68
69 </Grid>
70
71 </Grid>
72 </Window>
```

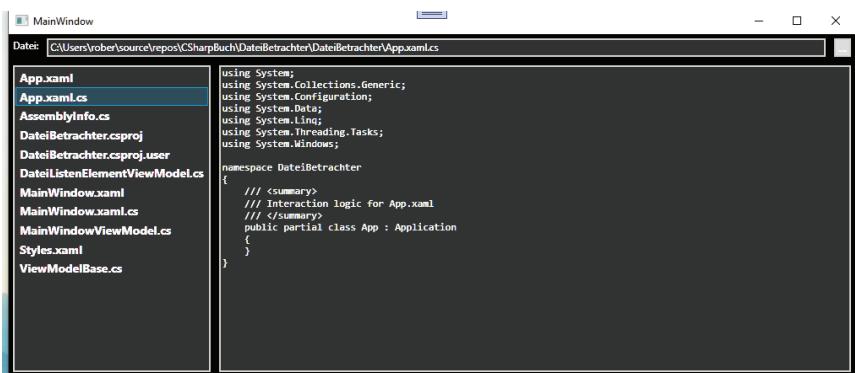


Abb. 17.9.1 Der Dateibetrachter mit MVVM-Entwurfsmuster

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/books/cs-kompendium/>



Sie erhalten die eBook-Ausgabe zum Buch  
kostenlos auf unserer Website:



<https://bmu-verlag.de/books/cs-kompendium/>

**Downloadcode:** siehe Kapitel 18

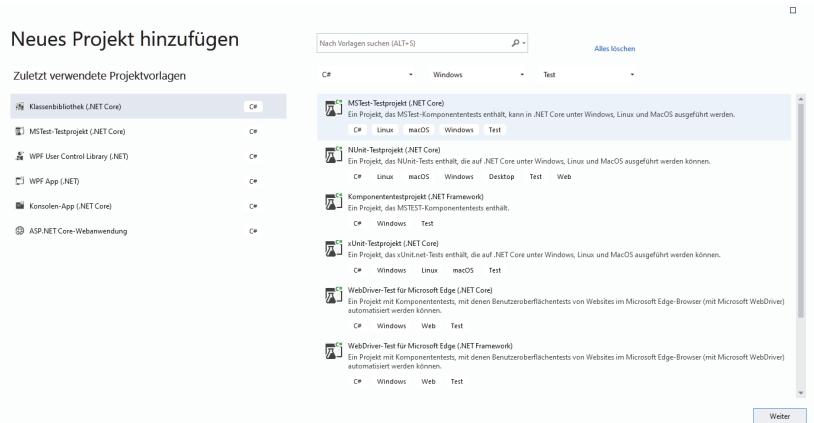
## Kapitel 18

# Testautomatisierung mit MS-Test

In den bisherigen Kapiteln haben wir oft eine Klasse mit einer bestimmten Funktionalität geschrieben und dann in der Klasse `Program` Code geschrieben, der diese Klasse getestet hat. Wir haben die Klasse `Program` auch oft als Testprogramm bezeichnet. Visual Studio enthält ein ausgereiftes Test-Framework, mit dem wir Testprogramme für unsere Klassen schreiben können. Das Test-Framework hat den Namen MS-Test. Des Weiteren enthält MS-Test auch Funktionalitäten zum Verwalten und Ausführen von unseren Tests. Diese Tests werden wir im weiteren Unit-Tests oder Komponententests nennen.

### 18.1 Ein Testprojekt anlegen

Um einen Unit-Test zu schreiben, benötigen wir zuerst eine zu testende Klasse und ein Test-Projekt. Als zu testende Klasse verwenden wir die Klasse `MainWindowViewModel` aus unserem DateiBetrachter-Programm. Ein Test-Projekt erhalten wir, indem wir der Projektmappe `DateiBetrachter` ein weiteres Projekt hinzufügen. Dazu klicken wir im Projektmappen-Explorer mit der rechten Maustaste auf die Projektmappe `DateiBetrachter` und wählen „`Hinzufügen\Neues Projekt...`“ aus dem Kontextmenü aus. Im Dialog „`Neues Projekt hinzufügen`“ stellen wir in der Auswahlliste Sprache C#, in der Auswahlliste Plattform Windows und als Projekttyp „Test“ ein. Wir wählen die Projektvorlage „`MSTest-Testprojekt (.NET Core)`“ aus und klicken auf die Schaltfläche „Weiter“.



**Abb. 18.1.1** Ein Testprojekt hinzufügen

Als Namen für unser Testprojekt vergeben wir DateiBetrachter.Test und klicken auf die Schaltfläche erstellen. Visual Studio erstellt uns dann ein neues Testprojekt in der Projektmappe DateiBetrachter.

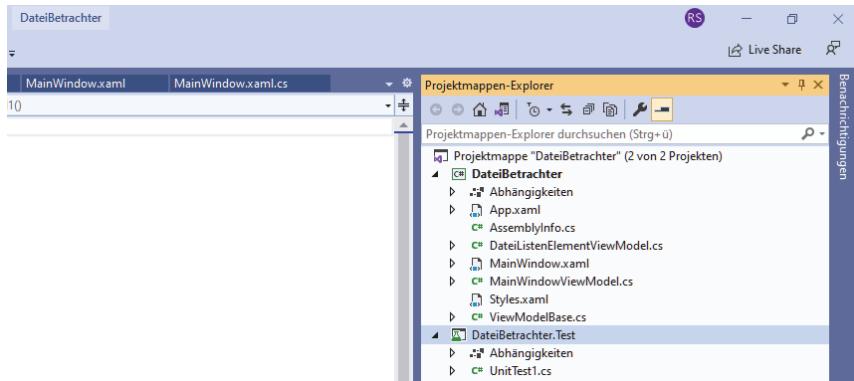


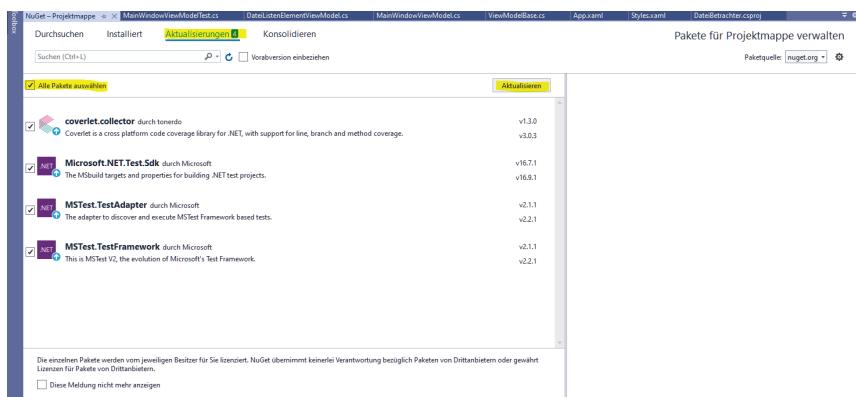
Abb. 18.1.2 Das neue Testprojekt

Um Testprogramme zu erstellen, müssen wir jetzt keinen Code mehr in unserem eigentlichen Programm DateiBetrachter erstellen und später eventuell wieder bereinigen. Sämtliche Testprogramme für Klassen des Projekts DateiBetrachter können wir jetzt im Projekt DateiBetrachter.Test schreiben.

Visual Studio hat auch gleich die erste Testklasse erstellt. Allerdings ist der Name `UnitTest1.cs` nicht wirklich sinnvoll für unsere Testklasse, daher benennen wir sie um in `MainWindowViewModelTest.cs`. Die Frage, ob wir die zur Datei gehörige Klasse auch mit umbenennen wollen, bestätigen wir mit Ja.

Bevor wir unsere erste Testklasse näher betrachten, müssen wir allerdings noch das MS-Testframework aktualisieren. Dazu klicken wir mit der rechten Maustaste auf die Projektmappe und wählen „NuGet-Pakete für Projektmappe verwalten...“ aus dem Kontextmenü aus.

## 18 Testautomatisierung mit MS-Test



**Abb. 18.1.3** Aktualisieren des MSTest-Frameworks

Wir wählen den Reiter Aktualisierungen aus, selektieren „Checkbox Alle Pakete“ und klicken auf die Schaltfläche „Aktualisieren“.

Da Microsoft das Paket MS-Test unabhängig von den .NET-Aktualisierungen und Visual Studio-Aktualisierungen weiterentwickelt und die neuesten Versionen über NuGet zur Verfügung stellt, ist es immer eine gute Idee, MS-Test auf dem neuesten Stand zu halten. Nachdem wir MS-Test aktualisiert haben, werden wir im nächsten Unterkapitel unseren ersten Komponententest erstellen.

## 18.2 Erstellen eines Komponententests

In diesem Unterkapitel erstellen unseren ersten Komponententest. Dazu betrachten wir die Testklasse, die Visual Studio für uns erstellt hat und die wir im letzten Unterkapitel in `MainWindowViewModelTest.cs` umbenannt haben.

```

1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2
3  namespace DateiBetrachter.Test
4  {
5      [TestClass]
6      public class MainWindowViewModelTest
7      {
8          [TestMethod]
9          public void TestMethod1()
10         {
11         }
12     }
13 }
```

Unsere Testklasse bindet die Klassenbibliothek Microsoft.VisualStudio.TestTools.UnitTesting ein. Diese benötigen wir, um die einzelnen Funktionalitäten des Test-Frameworks verwenden zu können. Die Klasse selbst erhält das Klassenattribut [TestClass] und die Methode TestMethod1() erhält das Methodenattribut [TestMethod]. Dadurch weiß der Test-Framework, welche Testmethoden er aufrufen kann.

Um einen Test für die Klasse MainWindowViewModel zu schreiben, müssen wir erst einen Projektverweis auf das Projekt DateiBetrachter erstellen. Mit unserem Test wollen wir unter anderem das Einlesen einer Datei testen, deshalb benötigen wir in unserem Testprojekt auch eine Testdatei. Dazu erstellen wir in unserem Testprojekt das Verzeichnis TestDaten und im Verzeichnis TestDaten erstellen wir eine Textdatei mit dem Namen Test.txt. Die Datei Test.txt erhält den Inhalt test. In den Eigenschaften der Datei setzen wir die Eigenschaft „In Ausgabeverzeichnis kopieren“ auf den Wert „Immer kopieren“.

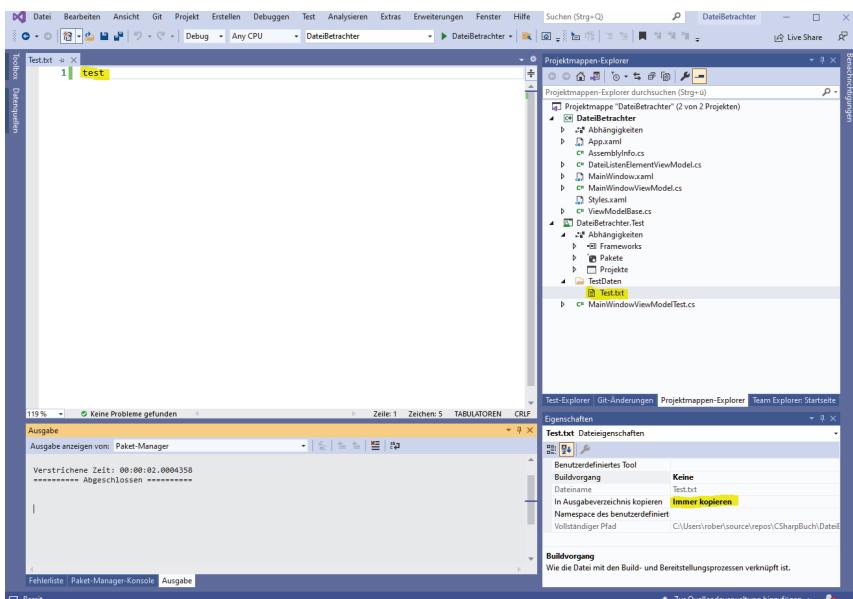


Abb. 18.2.1 Testdatenverzeichnis mit Testdatei

Die Klasse MainWindowViewModelTest ändern wir wie folgt ab:

```

1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using System.IO;
3
4  namespace DateiBetrachter.Test

```

## 18 Testautomatisierung mit MS-Test

```

5   {
6       [TestClass]
7       public class MainWindowViewModelTest
8   {
9       [TestMethod]
10      public void TestMainWindowViewModel()
11   {
12         var viewModel = new MainWindowViewModel();
13
14         var dateiListenElement = new
15         DateiListenElementViewModel();
16         dateiListenElement.VollerName = @"TestDaten\Test.
17         txt";
18         dateiListenElement.Name = "Test.txt";
19         viewModel.AusgewaehlteDatei = dateiListenElement;
20
21         Assert.AreEqual("test", viewModel.Ausgabe);
22         Assert.AreEqual(@"TestDaten\Test.txt", viewModel.
23         Datei);
24     }
25   }
26 }
```

Der Methode `TestMethod1()` geben wir den aussagekräftigeren Namen `TestMainWindowViewModel()`. In dieser Methode befindet sich nun unser ganzes Testprogramm. Zuerst erstellen wir mit `new` eine neue Instanz der zu testenden Klasse `MainWindowViewModel` und legen sie in der Variablen `viewModel` ab. Danach erstellen wir eine neue Instanz der Klasse `DateiListenElementViewModel` und speichern sie in der Variablen `dateiListenElement`. Den Properties `dateiListenElement.VollerName` und `dateiListenElement.Name` weisen wir die Texte „`TestDaten\Test.txt`“ und „`Test.txt`“ zu. Dann setzen wir die Property `viewModel.AusgewaehlteDatei` auf `dateiListenElement`.

Wenn wir uns an die Programmierung der App `DateiBetrachter` im Kapitel „Grafische Benutzeroberflächen mit WPF“ erinnern, passiert genau das, wenn wir auf eine Datei in der Dateiliste klicken. Beim Setzen der Property `viewModel.AusgewaehlteDatei` wird der Setter dieser Property aufgerufen. Den wollen wir uns hier nochmal genauer ansehen.

```

1 set
2 {
3     ausgewaehlteDatei = value;
4     try
5     {
6         Ausgabe = File.ReadAllText(AusgewaehlteDatei.
7         VollerName);
8         Datei = AusgewaehlteDatei.VollerName;
9     }
10    catch (Exception ex)
11    {
12        Ausgabe = ex.Message;
```

```
13     }
14
15     OnPropertyChanged();
16 }
```

In diesem Setter wird die Datei, deren Name und Pfad in der Property `AusgewaehlteDatei.VollerName` gespeichert ist, von der Festplatte eingelesen und der Inhalt der Datei in der Property Ausgabe gespeichert. Dann wird der vollständige Name der Datei in die Property Datei geschrieben.

Mit den letzten beiden Zeilen unserer Testmethode überprüfen wir, ob die Properties `Datei` und `Ausgabe` so gesetzt wurden, wie es vom Programm vorgesehen wurde. Dazu verwenden wir die statische Methode `AreEqual()` der Klasse `Assert` des Microsoft-Testframeworks. Die Methode `AreEqual()` bekommt zwei Ausdrücke übergeben und vergleicht die beiden, wenn die beiden Ausdrücke nicht gleich sind, wird eine `Exception` geworfen. In unserem Beispiel vergleichen wir die zu erwartenden Werte „`test`“ und „`TestDaten\Test.txt`“ mit den Properties `viewModel.Ausgabe` und `viewModel.Datei`. Der Test gilt als bestanden, wenn er ausgeführt werden kann, ohne eine `Exception` zu werfen.

In einem richtigen Projekt müssten wir noch einige Tests mehr schreiben, um die Funktionalität unseres Codes möglichst vollständig zu überprüfen, aber für unsere Beispiel begnügen wir uns mit dieser Testmethode.

Als nächstes wollen wir uns damit beschäftigen, wie wir unser Testprogramm ausführen können.

### 18.3 Verwendung des Test-Explorers

Unser Testprojekt ist kein richtiges Programm, das wir wie eine Konsolen-App oder eine WPF-App einfach starten können. Wenn wir das Testprojekt kompilieren, erhalten wir eine Klassenbibliothek, die man nicht so einfach ausführen kann. Um Testmethoden aus einer Klassenbibliothek auszuführen, verfügt Visual Studio über eine spezielle Komponente: den sogenannten Test-Explorer. Um den Test-Explorer anzuzeigen, wählen wir `Test/Test-Explorer` aus dem Menü von Visual Studio aus. Der Test-Explorer erscheint im gleichen Tab-Control, in dem sich auch der Projektmappen-Explorer befindet.

## 18 Testautomatisierung mit MS-Test



**Abb. 18.3.1** Der Testexplorer

Im Testexplorer sehen wir alle Testmethoden einer Solution in einer Baumstruktur angeordnet. Die Hierarchie des Baums ist Projekt/Namespace/Klasse/Methode. In unserem Beispiel gibt es ein Testprojekt mit dem Namen DateiBetrachter.Test. Das Testprojekt enthält einen Namespace mit dem Namen DateiBetrachter.Test. Der Namespace enthält eine Testklasse mit dem Namen MainWindowViewModelTest und diese Testklasse enthält eine Testmethode mit dem Namen TestMainWindowViewModel().

In der Toolbar des Testexplorers befindet sich eine Testzusammenfassung mit vier Icons. Das Erste Icon (Reagenzglas-Piktogramm) gibt die Anzahl der Test wieder, das zweite Icon (grüner Kreis mit weißem Haken) zeigt die Anzahl der bestandenen Tests an, das dritte Icon (roter Kreis mit weißem X) zeigt an, viele Tests ausgeführt und gescheitert sind und das vierte Icon (blaue Raute mit weißem Ausrufezeichen) gibt die Anzahl der noch nicht ausgeführten Tests an. In unserem Beispiel ist unser Test mit einer blauen Raute dekoriert, da er noch nicht ausgeführt wurde. Wenn wir jetzt mit der rechten Maustaste auf unsere Testmethode klicken und im Kontextmenü den Menüpunkt „Ausführen“ auswählen, wird unser Test ausgeführt und, wenn keine Exception geworfen wird, mit einem grünen Icon dekoriert.



**Abb. 18.3** Ein erfolgreich ausgeführter Test

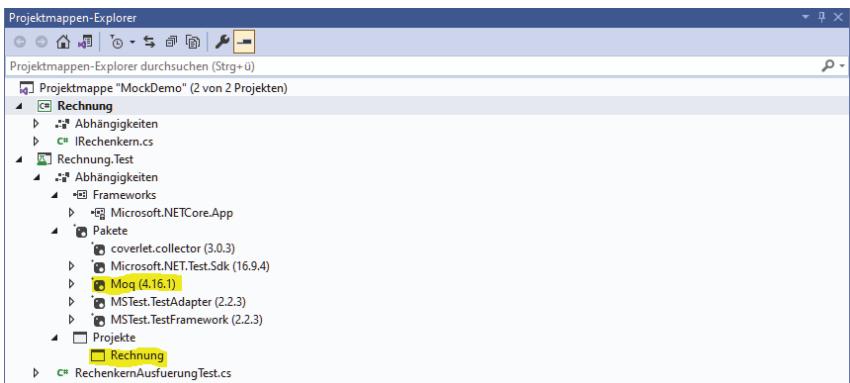
## 18.4 Abhängigkeiten reduzieren mit Mocks

Nachdem wir nun wissen, wie wir einen Komponenten-Test erstellen und ausführen können, werden wir uns mit dem Thema Mock beschäftigen. Das englische Wort Mock

übersetzt man im Deutschen am besten mit dem Fremdwort „Attrappe“. Also etwas, das nur so tut als ob. Wozu benötigen wir in der Programmierung also Attrappen? Stellen wir uns mal vor, wir haben eine Klasse, die wir Rechenkern nennen. Diese Klasse speichert in einer Listenstruktur Objekte, die einige Zahlen enthalten. Des Weiteren hat unsere Klassen zwei Methoden, die wir Rechne1 () und Rechne2 () nennen wollen, damit diese Methoden vernünftig arbeiten können, benötigen sie in der oben genannten Listenstruktur mindestens eine Million Objekte. Bei weniger als einer Million Objekte werfen diese Methoden eine Exception mit der Meldung, dass die Menge der Eingangsdaten nicht ausreichend ist. Die Typische Laufzeit für jede dieser Methoden beträgt zehn Sekunden. Ein Komponententest für diese Klasse muss sich einmal darum kümmern, dass genügend Eingangsdaten geladen werden, üblicherweise aus einer Datei oder Datenbank, und dann die beiden Methoden ausgeführt werden. Wenn wir jetzt davon ausgehen, dass das Laden der Eingangsdaten fünf Sekunden dauert, dann haben wir einen Komponententest der 25 Sekunden dauert. Als nächstes betrachten wir eine weitere Klasse. Diese nennen wir RechenkernAusfuehrung. Diese Klasse erstellt eine Instanz der Klasse Rechenkern und stellt einige Methoden bereit, die nach einer vorgegebenen Logik die Methoden Rechne1() und Rechne2() von Rechenkern mehrmals aufrufen. Wenn wir jetzt einen Komponententest für die Klasse RechenkernAusfuehrung erstellen, müssen wir uns wieder darum kümmern, dass wir eine Million Objekte für die Eingangsdaten der Klasse Rechenkern bereitstellen. Zudem wird dieser Komponententest sehr lange laufen, da wir für das Laden der Eingangsdaten fünf Sekunden und für jeden Aufruf von Rechne1() und Rechne2() jeweils zehn Sekunden veranschlagen müssen. Zudem ist es wahrscheinlich, dass wir verschiedene Zusammenstellungen von Eingangsdaten bereitstellen müssen, um alle Kombinationen der vorgegebenen Logik der Klasse RechenkernAusfuehrung abzudecken. Im Folgenden möchte ich zeigen, wie wir dieses Problem mit einer geschickten Softwarearchitektur und mit Hilfe von Mocks elegant umschiffen können. Zudem werden wir die freie Klassenbibliothek Moq kennenlernen, die uns beim Erstellen von Mocks gute Dienste leisten wird.

Für das oben beschriebene Beispiel legen wir uns eine Projektmappe mit zwei Projekten an. Die Projektmappe nennen wir MockDemo. Das erste Projekt ist eine Klassenbibliothek, die wir Rechnung nennen. Das zweite Projekt ist ein Komponententestprojekt und bekommt den Namen Rechnung . Test. Die Klasse Class1.cs im Projekt Rechnung benennen wir in IRechenkern.cs um und die Klasse UnitTest1.cs im Projekt Rechnung . Test nennen wir RechenkernAusfuehrungTest.cs. Danach fügen wir mit Hilfe von NuGet das Packet Moq dem Projekt Rechnung . Test hinzu und aktualisieren ebenfalls über NuGet den MS-Test Framework. Abschließend erstellen wir im Projekt Rechnung . Test einen Projektverweis auf das Projekt Rechnung. Danach sieht unsere Projektstruktur wie folgt aus:

## 18 Testautomatisierung mit MS-Test



**Abb. 18.4.1** Die Struktur der Projektmappe MockDemo

Rufen wir uns noch einmal unser aktuelles Ziel ins Gedächtnis. Wir wollen einen Komponententest. Für die Klasse `RechkernAusfuehrung` schreiben. Die Klasse `RechenkernAusfuehrung` benötigt eine Instanz der Klasse `Rechenkern`, die aufwendig zu erstellen ist. Zudem haben die Methoden der Klasse `Rechenkern` lange Laufzeiten. Unser Komponententest für die Klasse `RechenkernAusfuehrung` soll aber einfach zu erstellen sein und eine kurze Laufzeit haben. Um das zu erreichen, benötigen wir das Interface `IRechenkern`, das wie folgt aussieht:

```

1  using System;
2
3  namespace Rechnung
4  {
5      public interface IRechenkern
6      {
7          int Rechne1();
8
9          int Rechne2();
10     }
11 }
```

Das Interface definiert lediglich die beiden Methoden `Rechne1()` und `Rechne2()`, welche jeweils einen Integer-Wert zurückgeben. In einem richtigen Projekt muss es auch eine Klasse `Rechenkern` geben, welche das Interface `IRechenkern` implementiert. Aber in unserem Beispiel wollen wir uns auf den Komponententest der Klasse `RechenkernAusfuehrung` konzentrieren und können daher auf das Schreiben der Klasse `Rechenkern` verzichten.

Als nächstes legen wir im Projekt `Rechnung` die Klasse `RechenkernAusfuehrung` mit dem folgenden Programmcode an.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Rechnung
6  {
7      public class RechenkernAusfuehrung
8      {
9          private IRechenkern rechenkern;
10
11         public RechenkernAusfuehrung(IRechenkern rechenkern)
12         {
13             this.rechenkern = rechenkern;
14         }
15
16         public int Ausfuerung1()
17         {
18             var ergebnis = rechenkern.Rechne1();
19             if (ergebnis < 0)
20             {
21                 return rechenkern.Rechne2();
22             }
23             else
24             {
25                 return ergebnis;
26             }
27         }
28
29         public int Ausfuerung2()
30         {
31             var ergebnis = Ausfuerung1();
32             if (ergebnis < 10 && ergebnis >= 0)
33             {
34                 return rechenkern.Rechne1() + rechenkern.
35                 Rechne2();
36             }
37             else
38             {
39                 return ergebnis;
40             }
41         }
42     }
43 }
```

Der Konstruktor der Klasse `RechenkernAusfuehrung` nimmt ein Objekt vom Typ `IRechenkern` entgegen und legt es in der privaten Membervariablen `rechenkern` ab. Die Methode `Ausfuehrung1()` führt die Methode `Rechne1()` des im Konstruktor übergebenen `IRechenkern`-Objekts aus. Liefert `Rechne1()` einen Wert kleiner 0 zurück, wird die Methode `Rechne2()` ausgeführt und deren Ergebnis zurückgegeben. Ist der Rückgabewert von `Rechne1()` größer oder gleich Null, so wird direkt der Rückgabewert von `Rechne1()` zurückgegeben.

## 18 Testautomatisierung mit MS-Test

Die Methode `Ausfuehrung2()` führt die Methode `Ausfuerung1()` aus. Ist das Ergebnis von `Ausfuehrung1()` kleiner 10 und größer oder gleich 0, geben wir die Summe der Ergebnisse von `Rechne1()` und `Rechne2()` zurück, ansonsten geben wir das Ergebnis von `Ausfuerung1()` zurück.

Um die Logik der Methoden `Ausfuehrung1()` und `Ausfuehrung2()` zu testen, können wir dem Konstruktor der Klasse `RechenkernAusfuehrung` jede beliebige Implementierung des Interfaces `IRechenkern` übergeben. Das heißt, wir können eine Klasse verwenden, die `IRechenkern` implementiert, aber wesentlich einfacher aufgebaut ist als die eigentliche Klasse `Rechenkern`. Mit anderen Worten: Wir verwenden eine Klasse, die so tut, als ob beziehungsweise einen Mock. Zu Testzwecken könnten wir also eine simple Testimplementierung von `IRechenkern` erstellen und müssen nicht die schwergewichtige Implementierung der Applikation verwenden.

Mit Hilfe der Klassenbibliothek Moq können wir uns selbst das Erstellen einer Testimplementierung von `IRechenkern` sparen. Dazu ändern wir die Klasse `RechenkernAusfuehrungTest` wie folgt ab.

```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using Moq;
3
4  namespace Rechnung.Test
5  {
6      [TestClass]
7      public class RechenkernAusfuehrungTest
8      {
9          private Mock<IRechenkern> rechenkernMock=new
10             Mock<IRechenkern>();
11
12          [TestMethod]
13          public void RechenkernAusfuehrung1GibtRechne2zurueck()
14          {
15              //Arrange
16              rechenkernMock.Setup(m => m.Rechne1()).Returns(-100);
17              rechenkernMock.Setup(m => m.Rechne2()).Returns(100);
18              var rechenkernAusfuehrung = new
19                  RechenkernAusfuehrung(rechenkernMock.Object);
20
21              //Act
22              var ergebnis = rechenkernAusfuehrung.Ausfuerung1();
23
24              //Assert
25              Assert.AreEqual(100, ergebnis);
26          }
27
28          [TestMethod]
29          public void RechenkernAusfuehrung2GibtNullZurueck()
30          {
31              //Arrange
32              rechenkernMock.Setup(m => m.Rechne1()).Returns(-5);
33              rechenkernMock.Setup(m => m.Rechne2()).Returns(5);
```

```
34     var rechenkernAusfuehrung = new
35     RechenkernAusfuehrung(rechenkernMock.Object);
36
37     //Act
38     var ergebnis = rechenkernAusfuehrung.Ausfuerung2();
39
40     //Assert
41     Assert.AreEqual(0, ergebnis);
42 }
43 }
44 }
```

In unseren Testmethoden verwenden wir das sogenannte AAA-Entwurfsmuster. AAA steht für arrange, act, assert (Deutsch: einrichten, ausführen, überprüfen). Das heißt, wir richten zunächst unseren Test ein, indem wir alle Objekte und Variablen initialisieren, die wir für den Test benötigen. Danach führen den Test beziehungsweise die zu testende Methode aus. Und zum Schluss überprüfen wir, ob die getestete Methode das erwartete Ergebnis liefert hat.

Die Klasse `RechenkernAusfuehrungTest` definiert die private Membervariable `rechenkernMock` vom Typ `Mock<IRechenkern>`. Damit dieser Typ zur Verfügung steht, binden wir die Klassenbibliothek Moq ein. Die Membervariable `rechenkernMock` initialisieren wir mit dem parameterlosen Konstruktor der Klasse `Mock<IRechenkern>`. In der Testmethode `RechenkernAusfuehrung1GibtRechne2zurueck()` konfigurieren wir unseren Mock `rechenkernMock`, so dass die Methode `Rechne1()` den Wert -100 und die Methode `Rechne2()` den Wert 100 zurückgeben. Wenn wir mit den Konstruktor `Mock<T>()` einen Mock erstellen, legen wir mit dem generischen Typ-Parameter T fest, für welchen Typ wir einen Mock benötigen. In unserem Beispiel für den Typ `IRechenkern`. Mit der Methode `Setup()` der Klasse `Mock<IRechenkern>` können wir das Verhalten einer Methode von `IRechenkern` konfigurieren. Dazu übergeben wir der Methode den Lambda-Ausdruck `m -> m.Rechne1()`. Damit weiß die Methode `Setup()`, welche Methode von `IRechenkern` konfiguriert werden soll. Die Methode `Setup()` gibt ein Objekt vom Typ `ISetup<IRechenkern, int>` zurück, wobei der zweite generische Typparameter int dem Rückgabetyp der Methode `Rechne1()` entspricht. Der Typ `ISetup<IRechenkern>` liefert die Methode `Returns()`, die wir mit `Setup()` verketten. Der Methode `Returns()` übergeben wir den Wert -100. Das ist der Wert, den die Methode `Rechne1()` zurückliefern wird, wenn die Methode für die von `rechenkernMock` erzeugte Instanz von `IRechenkern` gerufen wird. Analog zur `Rechenkern1()` wird `Rechenkern2()` so konfiguriert, dass die Methode den Wert 100 zurückgibt.

Als nächstes erzeugen wir eine Instanz der Klasse `RechenkernAusfuehrung`. Dem Konstruktor übergeben wir `rechenkernMock.Object`. Die Property `Object` von `Mock<IRechenkern>` liefert ein Objekt vom Typ `IRechenkern`, dessen Methoden `Rechne1()` und `Rechne2()` jeweils den Wert -100 bzw. 100 zurückgeben.

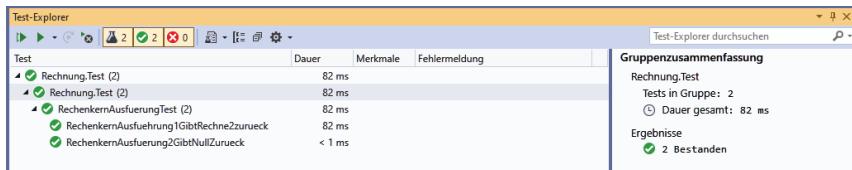
## 18 Testautomatisierung mit MS-Test

Nachdem der Test eingerichtet wurde, führen wir die Methode `Ausfuehrung1()` des erstellten RechenkernAusfuehrung-Objekts aus und speichern die Rückgabe der Methode in der Variablen `ergebnis`.

Zum Schluss überprüfen wir mit `Assert.AreEqual(100, ergebnis)`, ob die Methode `Ausfuehrung1()` das erwartete Ergebnis liefert hat.

Die Testmethode `RechenkernAusfuehrung2GibtNullZurueck()` testet die Methode `Ausfuehrung2()` und ist analog zur Testmethode `RechenkernAusfuerung1GibtRechne1Zurueck()` implementiert.

Jetzt können wir die Klasse `RechenkernAusfuehrung` testen, obwohl die eigentliche Implementierung des Interface `IRechenkern`, die für die Verwendung von `RechenkernAusfuerung` benötigt wird, noch gar nicht geschrieben wurde.



**Abb. 18.4.2** Ausführung der Komponententests für die Klasse `RechenkernAusfuehrung`

In diesem Beispiel haben wir den Mock für das Interface `IRechenkern` vollständig konfiguriert. Das mussten wir aber nur deswegen tun, weil in jeder Testmethode alle Methoden von `IRechenkern` ausgeführt werden. Wenn wir einen Mock für ein Objekt benötigen, das 20 Methoden bereitstellt, aber nur drei Methoden davon in einem Komponententest ausgeführt werden, müssen wir auch nur drei Methoden für diesen Mock konfigurieren.

Die Klassenbibliothek Moq ist sehr umfangreich und bietet wesentlich mehr als nur die Funktionalität, die in diesem Buch vorgestellt wird. Eine tiefere Betrachtung von Moq würde ein eigenes kleines Buch füllen und hier den Rahmen bei weitem sprengen. Eine Einschränkung von Moq sollten sie allerdings noch beachten: Moq kann nur für Interfaces oder für virtuelle Methoden und Properties verwendet werden.

## 18.5 Übungsaufgabe: Komponententests unter Verwendung von Mocks

In dieser Übungsaufgabe schreiben wir eine vereinfachte Version einer Klasse, die so ähnlich in einem Softwarepaket für Lebensversicherungen vorkommen könnte. Bei vielen Lebensversicherungen hat man ein sogenanntes Kapitalwahlrecht. Das heißt,

**18.5 Übungsaufgabe: Komponententests unter Verwendung von Mocks**

man kann bei Ablauf der Versicherung wählen, ob man einmalig eine größere Summe ausgezahlt bekommt oder ob man eine monatliche Rente erhält. Die monatliche Rente berechnet sich nachfolgender Formel:

$$\text{Rente} = ((\text{Einmalsumme} / (\text{Lebenserwartung} - \text{Alter})) / 12) * 0,95$$

Die Lebenserwartung ist ein Wert, der durch ein komplexes statistisches Verfahren gewonnen wird. In unserer Übung gehen wir davon aus, dass wir die Lebenserwartung von einer Klasse bekommen, die das folgende Interface implementiert:

```
1 public interface ILebenserwartung
2 {
3     double BerechneLebensErwartung();
4 }
```

**Teilaufgabe1:**

Schreiben Sie die Klasse RentenRechner. Die Klasse bekommt im Konstruktor ein Objekt vom Typ `ILebenserwartung` übergeben, um den Parameter Lebenserwartung für die obige Formel zu erhalten. Die Klasse liefert zudem die Methode `BerechneRente()`, welche die zu erwartenden Rente als `double`-Wert zurückgibt. Die Methode erwartet als Übergabeparameter die zu verrentende `EinmalSumme` vom Typ `double` sowie das Alter des Versicherungsnehmers, ebenfalls vom Typ `double`. Mit diesen Werten und der Lebenserwartung, die von dem im Konstruktor übergebenen `ILebenserwartung`-Objekt geliefert wird, berechnet die Methode `BerechneRente()` die zu erwartende Rente nach obiger Formel. Wenn das Alter über oder gleich der Lebenserwartung ist, soll die Methode eine `InvalidOperationException` mit einer entsprechenden Meldung werfen.

**Teilaufgabe2:**

Schreiben sie eine Testklasse für die Klasse `RentenRechner`. Die Testklasse soll zwei Testmethoden bereitstellen, die für das Interface `ILebenserwartung` jeweils einen Mock mit Hilfe der Klassenbibliothek Moq konfigurieren. Eine Testmethode soll ein sogenannter Positivtest sein. Das heißt, der Parameter Alter soll unter der Lebenserwartung liegen. Die andere Testmethode soll ein sogenannter Negativtest sein. Das heißt, der Parameter Alter soll über der Lebenserwartung liegen.

**Musterlösung für Teilaufgabe 1:**

```
1  using System;
2
3  namespace RentenBerechnung
4  {
5      public class RentenRechner
6      {
7          ILebenserwartung lebenserwartung;
8
9          public RentenRechner(ILebenserwartung lebenserwartung)
10         {
11             this.lebenserwartung = lebenserwartung;
12         }
13
14         public double BerechneRente(double einmalSumme, double
15 alter)
16         {
17             var erwartetesAlter = lebenserwartung.
18             BerechneLebensErwartung();
19             if(alter >= erwartetesAlter)
20             {
21                 throw new InvalidOperationException
22                 ("Rentenberechnung nicht möglich.");
23             }
24
25             return ((einmalSumme / (erwartetesAlter - alter)) /
26             12.0d) * 0.9d;
27         }
28     }
29 }
```

## Musterlösung für Teilaufgabe 2

```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using Moq;
3  using System;
4
5  namespace RentenBerechnung.Test
6  {
7      [TestClass]
8      public class TestRentenRechner
9      {
10          [TestMethod]
11          public void BerechneRentePositivTest()
12          {
13              //Arrange
14              var lebenserwartungMock = new
15              Mock<ILebenserwartung>();
16              lebenserwartungMock.Setup(m =>
17                  m.BerechneLebensErwartung()).Returns(75);
18              var rentenRechner = new
19              RentenRechner(labenserwartungMock.Object);
20
21              //Act
22              var rente = rentenRechner.BerechneRente(100000, 60);
23
24              //Assert
25              Assert.AreEqual(500.0, rente);
26          }
27
28          [TestMethod]
29          public void BerechneRenteNegativTest()
30          {
31              //Arrange
32              var lebenserwartungMock = new
33              Mock<ILebenserwartung>();
34              lebenserwartungMock.Setup(m =>
35                  m.BerechneLebensErwartung()).Returns(75);
36              var rentenRechner = new
37              RentenRechner(labenserwartungMock.Object);
38
39              try
40              {
41                  //Act
42                  var rente = rentenRechner.BerechneRente(100000,
43                      80);
44              }
45              catch(InvalidOperationException ex)
46              {
47                  //Assert
48                  Assert.AreEqual("Rentenberechnung nicht
49                      möglich.", ex.Message);
50              }
51          }
52      }
53  }
```

## 18 Testautomatisierung mit MS-Test

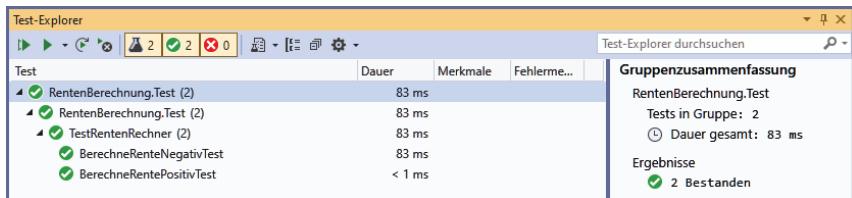


Abb. 18.5.1 Ausführung der Testklasse TestRentenRechner

Anleitung zum Download:

1. Klicken Sie auf den Link zum Buch
2. Klicken Sie rechts auf den Button “Kostenfreie eBook Ausgabe”
  3. Melden Sie sich an
  4. Geben Sie den Downloadcode ein



<https://bmu-verlag.de/books/cs-kompendium/>

Downloadcode: c4sch89kp

**Besuchen Sie auch unsere Website:**

Hier finden Sie alle unsere Programmierbücher und können sich Leseproben gratis downloaden:

**[www.bmu-verlag.de](http://www.bmu-verlag.de)**

**Probleme? Fragen? Anregungen?**

Sie können den Autor jederzeit unter [schiefele@bmu-verlag.de](mailto:schiefele@bmu-verlag.de) kontaktieren!

**Hat Ihnen das Buch gefallen?**

Helfen Sie anderen Lesern und bewerten Sie das Buch auf Amazon:

<http://amazon.de/ryp>

## Schlusswort

Zum Schluss möchte ich noch ein paar Dankesworte sprechen. Ein großer Dank gilt meiner Frau Katia und meinen Kindern Daniel und Alex, die mir viel Verständnis entgegenbrachten und mir die Zeit ermöglicht haben neben meiner hauptberuflichen Tätigkeit dieses Buch zu schreiben.

Ein weiterer großer Dank gilt meinen Lesern für Ihre Aufmerksamkeit und ihr Durchhaltevermögen. Mit dem Studium dieses Buchs haben Sie eine Menge geleistet. Sie haben eine neue, moderne und zukunftsträchtige Programmiersprache erlernt. Wenn sie jetzt vielleicht C# und Visual Studio in ihrem nächsten beruflichen oder privaten Projekt ausprobieren, werden Sie sich vielleicht, wie ein Klavierschüler nach den ersten drei Monaten Unterricht fühlen. Aber bleiben Sie dran nach drei bis vier Wochen mit einer ernsthaften Programmieraufgabe mit C# werden Sie sich als Klavierspieler fühlen.

## Glossar

### **.NET**

Ist ein von Microsoft entwickeltes plattformunabhängiges Framework für die Anwendungsentwicklung.

### **.NET Core**

Die neueste Evolution des .NET Frameworks, die die Entwicklung von Applikationen für Windows, Linux, Android iOS, MacOS, tvOS und Xbox unterstützt.

### **Android**

Von Google entwickeltes weit verbreitetes Betriebssystem für Smartphones und Tablets. Android gehört zu den .NET Zielplattformen.

### **ASP.NET MVC**

Ein Framework innerhalb des .NET-Frameworks zur Entwicklung von Webanwendungen nach dem MVC-Entwurfsmuster.

### **C#**

Eine Programmiersprache die von Anders Hejlsberg im Auftrag von Microsoft als Ersatz für Java im .Net-Framework entwickelt wurde.

### **CIL**

Steht für Common Intermediate Language. Dabei handelt es sich um eine Pseudomaschinensprache, die bei jedem Kompilervorgang einer .NET Sprache erzeugt wird.

### **CLI**

Steht für Common Language Infrastructure. Das ist eine Laufzeitumgebung die CIL code verarbeiten kann.

### **CLR**

Steht für Common Language Runtime. Konkrete Implementierung einer CLI für eine bestimmte Computer-Plattform.

### **Compiler**

Ein Programm das von Menschen (Programmierern) lesbaren Programmcode in von Computern verarbeitbaren Maschinencode übersetzt.

### **CSS**

Steht für Cascaded Style Sheets. Hierbei handelt es sich um eine BeschreibungsSprache um das optische Design einer Webseite zu definieren.

### **Datenbanktabelle**

Eine Datenbanktabelle organisiert Daten in einer tabellarischen Struktur wobei die Zellen einer Spalte alle den gleichen Datentyp aufweisen.

### **Debugger**

Computerprogramm das von Programmierern zur Fehlersuche in Computerprogrammen benutzt wird.

### **Design Pattern**

Siehe Entwurfsmuster.

### **Editor**

Computerprogramm zur Erstellung von Textdateien. Wird von Programmierern verwendet, um Programmcode zu schreiben.

### **Entität**

(Fremdwort aus dem lateinischen: das seiende)

Abstrakter Überbegriff für Ding, Gegenstand, Zustand oder einfach für alles das irgendwie existieren kann. In der Objekt-orientierten Programmierung versteht man unter Entität eine Instanz einer Klasse. In der Datenbankprogrammierung ist eine Entität ein Datensatz in einer Tabelle.

### **Entity Framework**

Ein Datenbank-unabhängiger Framework der Firma Microsoft, um Datenbankanwendungen Objekt orientiert zu programmieren.

### **Entity Relationship Model**

Ein Datenmodell für eine Datenbank, gemäß der von Edgar F. Codd 1970 vorgestellten Theorie der Relationalen Datenbanken.

### **Entwurfsmuster**

sind bewährte standardisierte Lösungsansätze für immer wiederkehrende Problemstellungen in der Softwareentwicklung.

### **Exception-System**

(Englisch für Ausnahmensystem)

Ein vom .NET Framework und anderen Programmierframeworks verwendetes System zur Fehlerbehandlung in Programmen. Exceptions werden in Fehlersituationen geworfen. Und können dann im Aufrufenden Programmteil behandelt werden.

### **Fremdschlüssel**

Ein Schlüssel in einer Datenbanktabelle der einen Datensatz einem anderen Datensatz in einer anderen Tabelle zuordnet. Der Fremdschlüssel ist der Primärschlüssel der anderen Tabelle.

### **Garbage Collector**

(zu Deutsch: Müllsampler)

Ein niedrig priorisierte Prozess im .NET Framework, der ständig im Hintergrund läuft und nicht mehr benötigten Speicher wieder freigibt. Das entlastet .NET Programmierer von der Speicherverwaltung.

### **Hochsprache**

Siehe Höhere Programmiersprache.

### **Höhere Programmiersprache**

Im Gegensatz zur Maschinesprache, deren Vokabular aus Zahlen besteht, besteht das Vokabular einer höheren Programmiersprache aus Vokabeln die einer menschlichen Sprache (meist Englisch) und Formelzeichen, die der Mathematik entlehnt sind.

### **HTML**

Steht für Hyper Text Markup Language. Eine Seitenbeschreibungssprache für die Gestaltung von Internetseiten.

### **IDE**

Siehe Integrierte Entwicklungsumgebung.

### **Integrierte Entwicklungsumgebung**

Softwarepaket zur Entwicklung von Software, das Editor, Compiler, Debugger und weitere Dienstprogramme unter einer Benutzeroberfläche vereinigt.

### **Intellisense**

Von Microsoft entwickelte Technologie, die dem Programmierer Vorschläge für den Programmcode schon während der Programmierung macht.

**Interpreter**

Ein Computerprogramm das ein, in einer Hochsprache geschriebenes, Computerprogramm ausführt.

**iOS**

Betriebssystem für Smartphones und Tablets der Firma Apple. iOS gehört zu den .NET Zielplattformen.

**Java**

Die zweitbeste Programmiersprache der Welt. (Private Meinung des Autors)

**JavaScript**

Eine interpretierte Skriptsprache, die von Webbrowsern verstanden wird, um dynamische und interaktive Webanwendungen zu ermöglichen.

**JIT-Compiler**

Steht für Just In Time Compiler. Ein Compiler der CIL code bei der Ausführung in echten Maschinencode umsetzt.

**Klassenbibliothek**

Eine Objekt-orientierte Sammlung von Klassen, die in anderen Programmen wiederverwendet werden kann.

**Komponententest**

Siehe Unittest.

**Konsolen-App**

Textbasierte Computeranwendung die in einer Textkonsole von Windows, Linux oder MacOS ausgeführt werden kann.

**Lambda-Ausdruck**

Ein Ausdruck, der ein Literal für eine funktionale Variable darstellt und einer funktionalen Variablen zugewiesen werden kann.

**Lazy Loading**

Ein Konzept in Datenbank-Frameworks, das dafür sorgt, dass Daten erst dann geladen werden, wenn sie vom Benutzer auch wirklich benötigt werden.

**Linq**

Steht für Language Integrated Query und ist eine von Microsoft entwickelte Objekt orientierte Abfragesprache, zur Abfrage beliebiger Objektstrukturen.

**Linux**

Beliebtes quelloffenes Betriebssystem für Desktop-Computer, Notebooks und Tablets. Linux gehört zu den .NET Zielplattformen.

**MacOS**

Betriebssystem der Firma Apple für Desktop-Computer und Notebooks dieses Herstellers. MacOS gehört zu den .NET Zielplattformen.

**Maschinensprache**

Eine Sprache zum Erstellen von Computerprogrammen. Maschinensprache hängt von dem in einem Computer verwendeten Prozessor ab, kann von Menschen nur schwer gelesen werden, kann aber von Computern direkt ausgeführt werden.

**Microsoft**

Ein 1975 gegründeter amerikanischer Softwarehersteller, der unter anderem für die Produkte Windows, SQL Server, Visual Studio und .Net verantwortlich ist.

**MS-Test**

Ein von Microsoft entwickeltes in Visual Studio integriertes Testframework zum

Erstellen, Verwalten und Ausführen von Unitests.

### **MSIL**

Steht für Microsoft Intermediate Language. Synonym für CIL (siehe CIL).

### **MVVM**

Steht für Model-View-ViewModel und bezeichnet ein Entwurfsmuster, das von Microsoft speziell zur Entwicklung von grafischer Benutzeroberflächen entworfen wurde. Es beruht auf dem MVC Entwurfsmuster.

### **MVC**

Steht für Model View Controller und bezeichnet ein Entwurfsmuster das 1979 zum ersten Mal für die Programmiersprache SmallTalk beschrieben wurde. Es teilt den Code in 3 Schichten auf wobei Model für das Datenmodell und View für die Benutzeroberfläche steht. Die Controller-Schicht vermittelt zwischen Benutzeroberfläche und Datenmodell.

### **Primärschlüssel**

Ein Identifikationskennzeichen, das einen Datensatz in einer Datenbanktabelle eindeutig identifiziert. D. h. ein Primärschlüssel darf in einer Datenbanktabelle nur einmal vorkommen.

### **Projektmappe**

Eine Organisationsstruktur innerhalb von Visual Studio, die mehrere zusammengehörige Projekte gruppiert.

### **Projektmappen-Explorer**

Komponente in Visual Studio die, den Inhalt einer Projektmappe in einer Baumstruktur darstellt.

### **Razor-Engine**

Technologie in ASP.NET MVP mit der mit einer Mischung aus C#- und HTML-code Webanwendungen entwickelt werden können.

### **Referenzielle Integrität**

Ein Begriff aus der Datenbankprogrammierung. Die Referenzielle Integrität ist dann erfüllt, wenn zu allen abhängigen Entitäten eine Entität existiert, von der die jeweilige abhängige Entität abhängig ist. Wenn eine Mitarbeiter-Entität einer Firmen-Entität mit dem Schlüssel x zugeordnet ist, dann muss auch eine Firmen-Entität mit dem Schlüssel x in der Datenbank existieren.

### **Source Code Highlighting**

Ein Verfahren, das durch Farbliche Hervorhebung die Lesbarkeit von Computerprogrammen in einem Editor verbessert.

### **SQL Server**

Ein Server basiertes Relationales Datenbank Management System (RDBMS) der Firma Microsoft.

### **Tabelle**

Siehe Datenbanktabelle.

### **Test-Explorer**

Komponente des Test-Frameworks MSTest, mit der in Visual Studio ein Programmierer seine Komponententests organisieren und ausführen kann.

### **tvOS**

Betriebssystem für Apple-TV Geräte. tvOS ist eine .NET Zielplattform.

### **Unitest**

Ein programmiert Test der eine isolierte Funktionalität eines Computerpro-

## Glossar

gramms auf seine korrekte Funktionsweise überprüft.

### **Visual Studio**

Eine Entwicklungsumgebung der Firma Microsoft mit grafischer Benutzeroberfläche zur Entwicklung von Computeranwendungen.

### **Visual Studio Community Edition**

Kostenlose Version von Visual Studio für Schüler, Studenten, Autodidakten, Nicht kommerzielle Projekte und kommerzielle Projekte im kleinen Rahmen.

### **Web-Browser**

Programm zum Darstellen von Webseiten und Ausführen von Webanwendungen aus dem Internet und anderen Netzwerken.

### **Windows**

Von Microsoft entwickeltes und sehr weit verbreitetes Betriebssystem für Desktop-

Computer, Notebooks und Tablets. Windows gehört zu den .NET Zielplattformen.

### **WPF**

Steht für Windows Präsentation Framework. Ist ein Bestanteil des .NET Frameworks mit dem Windows Anwendungen mit einer grafischen Benutzeroberfläche entwickelt werden können.

### **XAML**

Steht für Extensible Application Markup Language. Ein von Microsoft entwickeltes XML-Vokabular zur Beschreibung grafischer Benutzeroberflächen.

### **Xbox**

Beliebte von Microsoft entwickelte Spielkonsole. Die Xbox gehört zu den .NET Zielplattformen.

# Index

## A

- abgeleitete Klassen ..... 203
- absolute Pfadangabe ..... 403
- abstract ..... 240
- abstrakte Klassen ..... 240
- App.xaml ..... 549, 586
- Array 84, 85, 86, 87, 89, 94, 95, 99, 101, 102, 103, 104, 105, 106, 108, 120, 122, 123, 128, 130, 131, 217, 323, 404, 407, 408, 424, 425, 426
- ASP.NET vi, 7, 8, 17, 502, 503, 504, 505, 506, 508, 511, 512, 514, 515, 517, 525, 527, 528, 529, 537, 587
- AsParallel ..... 322, 323, 324
- Aufrufargumente ..... iv, 28, 120
- Aufzählung ..... 60

## B

- Backslash ..... 43
- Baumstrukturen ..... 247, 249, 250, 251, 253, 254, 255, 257, 260
- bedingte Zuweisung ..... iv, 67
- bool ..... 42
- Boolesche Algebra ..... 41
- Boolean ..... 42
- break iv, 63, 64, 65, 66, 73, 87, 90, 91, 122, 145, 146, 190, 191, 193, 195, 197, 199, 200
- Byte ..... 38

## C

- Carriage Return ..... 400
- catch ..... 336
- char ..... 44
- Codenavigation ..... 370
- ConnectionString ..... 459, 527, 528
- continue .... iv, 87, 92, 93, 135, 137, 139, 142, 146, 147, 404
- Controller ..... 519, 525

## D

- Datenbank erstellen ..... 451
- Datenbankserver ..... 435, 451, 453
- Datenstrukturen ..... 101
- Datentyp ..... 38
- Datentypen ..... 101
- DateTime ..... 45, 46, 247, 276, 323, 324, 325, 427, 429, 432, 492, 500
- DbContext ..... 459, 483, 490, 494, 526, 527
- deserialisieren ..... 416
- Debugger ..... 384
- Decimal ..... 39
- Delegaten ..... 277
- Dictionary ..... v, 107, 108, 109, 112, 113, 114, 116, 178, 179, 184, 185, 187, 217, 511, 512, 513, 518, 523
- doppelte obere Anführungszeichen ..... 43
- Double ..... 39, 579, 581, 585
- do-while-Schleife ..... 78

## E

- Einfache Methoden ..... 119
- Entity Framework ..... 453
- Entity-Modell ..... 455
- enum ..... 60
- Erweiterungsmethoden ..... 243
- Escape-Sequenzen ..... 43
- Exception ..... 334
- Exceptions kontrolliert auslösen ..... 340

## F

- false ..... 41
- Fehler vi, 29, 56, 81, 102, 108, 110, 125, 130, 154, 157, 174, 207, 208, 212, 228, 235, 236, 243, 244, 245, 275, 296, 302, 334, 335, 337, 339, 342, 345, 367, 384, 385, 386, 388, 462, 484, 509, 510, 515, 562, 568, 569, 576, 586, 597
- Fehlerbehandlung ..... 334
- Funktionale Programmierung ..... 203

FileDialog-Objekt.....	567
float.....	38, 39, 40, 41
for-Schleife .....	82, 83, 85, 86
funktionale Programmierung .....	277
<b>G</b>	
Garbage Collector v, 217, 218, 219, 222, 223, 224, 225, 226, 227, 228, 229, 405	
Generische Klassen .....	247
Generische Methoden.....	272
Generische Properties.....	275
Grid-Steuerelement.....	563
<b>I</b>	
Instanz.....	160
Int16 .....	38
Int32 .....	38, 46
Int64 .....	38
Integer ..	36, 37, 38, 39, 40, 44, 45, 51, 61, 62, 63, 85, 125, 180, 323, 324, 325, 616
Interface .....	210
<b>K</b>	
Klasse .....	155
Klassenbibliothek erstellen.....	361
Kommentare .....	30
Komponententests.....	608
Konsolenanwendung .....	iv, 26
Konstruktor .....	158
Konvertieren .....	iv, 50, 416
<b>L</b>	
Line Feed .....	400
Linq v, 7, 11, 48, 296, 297, 298, 299, 300, 302, 303, 306, 307, 311, 312, 313, 315, 316, 317, 319, 320, 321, 322, 323, 324, 325, 326, 327, 330, 331, 332, 412, 414, 415, 417, 418, 421, 422, 424, 425, 426, 474, 475, 477, 495, 497, 509, 521, 525, 526, 528, 529, 530, 531, 553, 555, 561, 588	
LINQ to DataSet.....	297
LINQ to Entities .....	297
Linq to Objects .....	v, 7, 296
LINQ to Objects .....	297
LINQ to SharePoint.....	297
LINQ to SQL.....	297
ListBox-Control 570, 571, 575, 578, 582, 596	
ListBox-Steuerelement.....	570
logical and .....	41
logical not .....	41
logical or .....	41
logische Operatoren .....	iv, 56
long.....	38, 64
<b>M</b>	
MainWindow.xaml. 549, 550, 552, 553, 554, 555, 556, 559, 561, 562, 565, 567, 568, 570, 572, 573, 575, 584, 586, 587, 588, 589, 594, 595, 596	
Mehrdimensionale Arrays.....	iv, 103
Mehrfachverzweigung .....	60
Methoden .....	119
Methoden mit Rückgabewerten .....	123
Methoden mit Übergabeparametern....	120
Methoden überladen.....	128
Microsoft.AspNetCore.Mvc ....	509, 521, 528
Microsoft.EntityFrameworkCore..	454, 455, 459, 490, 493, 495, 497, 526, 527
Microsoft.VisualStudio.TestTools.	
UnitTesting.....	610, 611, 618, 623
Microsoft SQL-Server .....	435
Migration-Tool .....	461, 462
Mock .....	614, 618, 619, 620, 621, 623
Moq .....	615, 618, 619, 620, 621, 623
MS-Test.....	vi, 7, 608, 610, 615
MVVM-Entwurfsmuster..	vi, 587, 595, 596, 597, 606
<b>N</b>	
named tuples .....	110
Navigationproperty .....	464
neue Zeile .....	43
NuGet 453, 454, 455, 460, 594, 609, 610, 615	
<b>O</b>	
Objektinitialisierer .	307, 308, 309, 310, 311, 312, 327, 478
Objektorientierte Programmierung.....	152
Operatoren .....	36

Operatoren überladen..... 174

## P

Paket-Manager-Konsole. 460, 461, 463, 465

ParallelQuery<T>..... 322

Polymorphie v. 152, 229, 230, 231, 232, 233, 234, 235

Primärschlüssel 458, 462, 464, 467, 469, 475, 479, 487

private ..... 165

Projektmappe..... 352

Properties ..... 165

public..... 164

## R

rekursiver Methodenaufruf..... 251

relative Pfadangabe ..... 403

Ressourcen..... 577

## S

Sbyte ..... 38

Schleife.....  
iv, 76, 77, 78, 79, 80, 82, 83, 84, 85, 86, 89, 91, 92, 93, 94, 97, 98, 99, 101, 104, 105, 109, 111, 131, 180, 217, 226, 227, 231, 233, 251, 253, 262, 299, 307, 312, 321, 323, 346, 404, 405, 409, 413, 423, 425, 426, 473, 474, 475, 480, 481, 532, 534, 575, 597

Schnittstelle ..... 210

Schriftarten..... 577

Schriftgrößen ..... 577, 581

sealed..... 240

short ..... 38

Single..... 38

SolidColorBrush-Control..... 577

SQL vi, 7, 296, 297, 298, 435, 436, 437, 438, 439, 440, 441, 442, 444, 445, 447, 448, 449, 450, 451, 452, 453, 459, 461, 462, 466, 468, 472, 474, 475, 479, 481, 483, 486, 502

serialisieren ..... 416

StaticResource. 577, 578, 579, 580, 581, 582, 585, 586, 606

string ..... 42

Style-Control..... 577, 582

Switch-Case..... 60

System.Collections.Generic ... 104, 105, 107,

112, 113, 114, 116, 134, 136, 138, 141, 145, 156, 158, 159, 160, 162, 163, 165, 166, 167, 169, 171, 174, 183, 184, 185, 187, 190, 193, 197, 203, 204, 206, 207, 212, 213, 214, 215, 216, 219, 221, 223, 226, 227, 230, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 244, 245, 247, 248, 249, 251, 255, 256, 258, 260, 265, 266, 269, 270, 272, 275, 285, 286, 287, 288, 290, 291, 299, 300, 304, 307, 308, 309, 310, 313, 317, 326, 328, 342, 347, 364, 367, 385, 412, 414, 416, 417, 419, 420, 422, 429, 432, 457, 458, 459, 463, 464, 466, 467, 468, 470, 472, 474, 476, 478, 480, 482, 484, 485, 490, 492, 493, 495, 497, 499, 509, 521, 525, 526, 528, 529, 530, 553, 555, 561, 587, 588, 590, 592, 598, 599, 600, 601, 603, 617

System.ComponentModel..... 467, 468, 490, 492, 493, 525, 590, 591, 593, 598

System.ComponentModel.DataAnnotations 467, 468, 490, 492, 493, 525

System.IO 402, 403, 404, 405, 406, 407, 408, 410, 417, 418, 420, 422, 423, 424, 425, 432, 561, 573, 601, 603, 611

System.Runtime.CompilerServices 590, 591, 593, 598

System.Runtime.Serialization 342, 347, 419, 432

System.Text..... 156, 158, 159, 160, 162, 163, 165, 166, 167, 169, 171, 174, 183, 184, 185, 187, 190, 193, 197, 203, 204, 206, 207, 211, 212, 213, 214, 215, 219, 221, 223, 236, 238, 239, 240, 241, 242, 244, 245, 247, 248, 250, 251, 255, 256, 258, 260, 265, 266, 269, 270, 272, 275, 285, 286, 287, 288, 290, 291, 300, 304, 307, 308, 309, 310, 313, 317, 318, 326, 328, 342, 347, 364, 367, 385, 416, 419, 429, 432, 457, 458, 459, 463, 464, 466, 467, 468, 470, 490, 492, 493, 495, 497, 553, 555, 561, 587, 588, 590, 593, 598, 599, 600, 601, 603, 617

System.Threading	45, 46, 324, 509, 521, 525, 526, 528, 529, 530, 553, 555, 561, 588
System.Windows	..... 561
System.Windows.Controls	.... 553, 555, 561, 573, 588
System.Windows.Data	.... 553, 555, 561, 589
System.Windows.Documents	554, 555, 561, 589
System.Windows.Forms	573, 574, 600, 601, 603, 604
System.Windows.Input	... 554, 555, 561, 589
System.Windows.Media	..... 561
System.Windows.Media.Imaging	.. 554, 555, 561, 589
System.Windows.Navigation	. 554, 555, 561, 589
System.Windows.Shapes	554, 555, 561, 589
System.Xml.Serialization	251, 417, 418, 419, 420, 421, 422, 432
<b>T</b>	
Tabulator	..... 43
Test-Explorer	..... 613
Test-Framework	..... 608, 611
Testprogramm	..... 608
Testprojekt	..... 608
Textdatei	..... 400
Textdateien	vi, 43, 89, 90, 400, 411
Ticks	..... 46
true	..... 41
try	..... 336
Tuple	..... v, 109, 110
Typen	..... 35, 36
Typisierte Listen	iv, 104
<b>U</b>	
uint	..... 38
UInt16	..... 38
UInt32	..... 38
UInt64	..... 38
ulong	..... 38
Unicode	..... 42, 43, 44
Unit-Tests	..... 608
ushort	..... 38
<b>V</b>	
Variablen	..... 36
Vererbung	..... 203
Vergleichsoperatoren	..... iv, 56
Verzeichnis	.... 15, 23, 29, 178, 179, 181, 184, 185, 187, 190, 193, 197, 368, 371, 372, 402, 403, 409, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 511, 521, 525, 531, 535, 575, 597, 611
Verzweigungen	iv, 42, 55, 57, 58, 59, 70, 71, 72
View	..... 519, 531, 533
ViewModel	.... 512, 513, 514, 529, 530, 531, 587, 590, 592, 593, 594, 595, 596
Virtuelle Methoden	..... 229
<b>W</b>	
Wagenrücklauf	..... 43
Webanwendung	. vi, 502, 503, 504, 505, 506, 507, 508, 509, 514, 515, 516, 517, 519, 523, 524, 525, 526, 527, 528, 531, 532, 533, 534, 536, 537, 539
while-Schleife	..... 76, 78, 82
WPF	..... 548
WPF-Styling	..... 577
Wurzelknoten	.. 249, 250, 251, 252, 253, 352, 412, 413, 415, 419, 420, 421, 422, 552
<b>X</b>	
XAML	vi, 549, 550, 551, 552, 554, 556, 561, 563, 577, 581, 583, 584, 587, 595, 596
XML	vi, 296, 297, 400, 411, 412, 413, 414, 415, 416, 417, 419, 420, 421, 422, 427, 428, 466, 552, 564, 584, 594, 595
<b>Z</b>	
Zugriffsmodifizierer	..... 164

**Linux Handbuch für Einsteiger: Der leichte Weg zum Linux-Experten (356 Seiten)**

Dieses Buch erklärt Ihnen alle Details, die für die Nutzung von Linux von Bedeutung ist. Es stellt verschiedene Distributionen vor und geht darauf ein, welche Eigenschaften diese auszeichnen. Sie lernen, wie Sie Linux installieren und wie Sie die grundlegenden Funktionen des Betriebssystems nutzen. Sie lernen sogar, kleine Shell-Scripts zu programmieren, um Aufgaben zu automatisieren. Für Umsteiger von Windows auf Linux das perfekte Buch für den Einstieg. -

**Inhalte**

- ▶ Die Vor- und Nachteile von Linux
- ▶ Ubuntu, Mint und weitere Linux-Distributionen
- ▶ Linux installieren: So gehen Sie vor
- ▶ Die grundlegenden Funktionen des Betriebssystems
- ▶ Shell-Scripts programmieren
- ▶ Aufgaben in Linux automatisieren

**Vorteile**

- ▶ Die Verwendung von Linux verständlich erklärt: ganz ohne Vorkenntnisse
- ▶ Vielfältige Funktionen des Betriebssystems kennenlernen, um dessen Möglichkeiten voll auszunutzen
- ▶ Anschauliche Gestaltung durch zahlreiche Screenshots
- ▶ Eigenständige Übungen vertiefen den Inhalt
- ▶ Zugang zu Gratis-eBook mit dem Buch

**Hier informieren:** <https://bmu-verlag.de/books/linux/>

**Git Handbuch für Einsteiger: Der leichte Weg zum Git-Experten (312 Seiten)**

In diesem Buch lernen Sie mit Git zu arbeiten. Zu Beginn erhalten Sie alle wichtigen Informationen über Git und lernen die Software zu installieren. Danach legen Sie ihr erstes Repository an. Dieses stellt die Grundlage für die Arbeit mit Git dar. Sie lernen, wie Sie einzelne Zweige erstellen und wie Sie die einzelnen Versionen wieder zusammenführen. Darüber hinaus erfahren Sie welche Möglichkeiten es für die Online-Zusammenarbeit gibt. Sie lernen sogar, wie Sie Git auf einem Server einrichten können, um die Zusammenarbeit in einem Unternehmen zu organisieren.

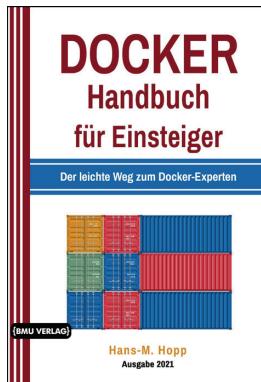
**Inhalte**

- ▶ Was ist ein Versionsverwaltungssystem?
- ▶ Git installieren
- ▶ Ein Repository für die Versionsverwaltung erstellen
- ▶ GitHub als Beispiel für eine Plattform für die Online-Zusammenarbeit
- ▶ Verschiedene Entwicklungszweige erstellen
- ▶ Git auf einem Server einrichten
- ▶ Das Zusammenspiel mit anderen Programmen für die Versionsverwaltung

**Vorteile**

- ▶ Klar strukturierter und verständlicher Aufbau: bestens für Anfänger geeignet
- ▶ Umfassende Einführung in Git – von den Grundfunktionen bis hin zu fortgeschrittenen Anwendungen
- ▶ Zahlreiche Abbildungen gestalten die Beispiele anschaulicher
- ▶ Vielfältige Praxisbeispiele und Übungen
- ▶ Kostenfrei als eBook herunterladen

**Hier informieren:** <https://bmu-verlag.de/books/git/>

**Docker Handbuch für Einsteiger: Der leichte Weg Zum Docker-Experten  
(476 Seiten)**

Docker und Kubernetes gehören aktuell zu den heißesten Themen in der Welt der Softwareentwicklung. Entwickler mit den dafür nötigen Fertigkeiten und Kenntnisse werden bei Projekten immer mehr gefragt. Wenn Sie Ihrer Karriere einen Schub nach oben geben möchten, dann ist es Zeit, dass Sie sich in die Welt des Cloud-Computing einarbeiten und mit Entwicklungssystemen wie Docker und Kubernetes befassen. Dieses Buch wird Ihnen beim Einstieg in diese Themen helfen.

Wenn Sie Systeme mit Docker und Kubernetes aufsetzen, dann ist es möglich die vorhandenen Systemressourcen optimal zu nutzen. Microservices lassen sich dadurch flexibel einsetzen und bedarfsgerecht skalieren.

**Inhalte:**

- ▶ Installation und Einrichtung von Docker
- ▶ Arbeiten mit Docker Images
- ▶ Tools zur Arbeit mit Docker
- ▶ Daten speichern in Containern.
- ▶ Arbeiten mit Container Logs
- ▶ Docker Netzwerke
- ▶ Eigene Webseiten mit Docker erstellen
- ▶ Arbeiten mit Docker Compose
- ▶ Einsatz von Docker Swarm und Docker Stack
- ▶ Einführung in Kubernetes und die Google Kubernetes Engine

**Vorteile:**

- ▶ Einfache und praxisnahe Erklärungen tragen zum Verständnis bei
- ▶ Anschauliche Beispiele unterstützen beim Nachvollziehen
- ▶ Zahlreiche Praxisprojekte zum Nachbauen dienen als Vorlage für eigene Projekte

**Hier informieren:** <https://bmu-verlag.de/books/docker/>

**C Programmieren für Einsteiger: Der leichte Weg zum C-Experten (300 Seiten)**

C ist eine der etabliertesten und weitverbreitetsten Programmiersprachen der Welt und Basis vieler moderner objektorientierter Sprachen wie Java, C++ oder C#. In diesem Buch lernen Sie das Programmieren mit C beginnend mit den Grundlagen verständlich und praxisorientiert, ohne dass dabei Vorkenntnisse notwendig wären. Nach dem Durcharbeiten des Buches sind Sie in der Lage eigene komplexere C Anwendungen inklusive grafischer Oberflächen zu entwickeln.

**Inhalte**

- ▶ Alle Grundlagen der Programmierung in C verständlich erklärt
- ▶ Arbeit mit Modulen und Bibliotheken zur Vermeidung von Redundanzen
- ▶ Effiziente Programmierung mit Zeigern und Referenzen
- ▶ Grafische Benutzeroberflächen mit GTK zur einfachen Benutzerinteraktion
- ▶ Dateien und Datenbanken zur dauerhaften Datenspeicherung
- ▶ Professionelles Debugging zur schnellen Fehlersuche

**Vorteile**

- ▶ Praxisnahe Erklärungen tragen zum einfachen Verständnis bei
- ▶ Übungsaufgaben mit Lösungen nach jedem Kapitel sichern den Lernerfolg
- ▶ Umfangreiche Praxisprojekte zum Nachbauen dienen als Vorlagen für eigene Projekte
- ▶ Quellcode zum Download zum schnellen Ausprobieren
- ▶ eBook Ausgabe kostenfrei dabei

**Hier informieren:** <https://bmu-verlag.de/books/c/>

## Professionelle .NET Programme mit C#!

C# ist die, am häufigsten verwendete, Programmiersprache in anspruchsvollen .NET-Projekten. Die Sprache ist vollständig objektorientiert, typsicher und unterstützt zudem die wichtigsten Elemente der Funktionalen Programmierung. Als All-Zweck-Programmiersprache kann C# sowohl in kaufmännischen Geschäftsanwendungen als auch im technischen Bereich verwendet werden.

Dieses Kompendium führt Sie praxisnah, Schritt für Schritt in die Programmierung mit C# ein. Es eignet sich für Anfänger und Fortgeschrittene. Anfänger können damit ohne Vorkenntnisse, mit aufeinander aufbauenden Kapiteln, die Programmierung der wichtigsten Anwendungstypen mit C# erlernen. Für Fortgeschrittene ist es als Nachschlagewerk, über die gängigsten Programmierthemen mit C#, geeignet.

### Das C#-Kompendium im Überblick:

#### Grundlagen:

- ▶ Einführung in .NET
- ▶ Die Entwicklungsumgebung
- ▶ Das erste Programm
- ▶ Variablen und Ihre Typen
- ▶ Komplexe Datenstrukturen: Arrays, Listen und Verzeichnisse
- ▶ Den Programmablauf mit Verzweigungen und Schleifen steuern
- ▶ Mit Methoden ein Programm organisieren
- ▶ Einführung in die objektorientierte Programmierung (OOP)

#### Fortgeschrittene Themen

- ▶ Objektorientierung für Fortgeschrittene
- ▶ Objekte verarbeiten mit LINQ
- ▶ Exceptions: ein System zur Behandlung von Fehlern
- ▶ Visual Studio effizient verwenden
- ▶ Dateien verarbeiten
- ▶ Datenbankzugriff
- ▶ Anwendungen fürs Web mit ASP.NET
- ▶ Grafische Benutzeroberflächen erstellen mit WPF
- ▶ Automatisierte Softwaretests mit MS Test

Dieses Buch bietet Ihnen einen umfassenden Einstieg in die Programmierung mit C#!

#### Über den Autor



Robert Schiefele hat sich als Schüler zu Beginn der 80er Jahre zum ersten Mal mit Programmierung beschäftigt. Nach dem Studium der Informatik hat er sein Hobby zum Beruf gemacht und zunächst 4 Jahre als Datenbankprogrammierer gearbeitet. Danach war er 16 Jahre im Angestelltenverhältnis als IT-Consultant in zahlreichen nationalen und internationalen Kundenprojekten tätig. Seit November 2016 arbeitet er als freiberuflicher Softwareentwickler in .Net basierten Projekten für Banken, Versicherungen und die Industrie.