

>>> Vervollständigen Sie Ihre Bibliothek!

KI zum Nachschlagen – unsere Buchreihe

- Einführung in die Künstliche Intelligenz [↗](#)
- ChatGPT und Prompt Engineering [↗](#)
- Der Microsoft Copilot [↗](#)
- KI im Office-Management [↗](#)

Buchreihe öffnen [↗](#)

Unsere Buchreihe: Wissen rund um Netzwerke

- Netzwerke – Grundlagen [↗](#)
- Netzwerke – Sicherheit [↗](#)
- Netzwerke – Netzwerktechnik [↗](#)
- Netzwerke – Protokolle und Dienste [↗](#)

Buchreihe öffnen [↗](#)

Microsoft 365-Workshop

- Mein persönlicher Arbeitsplatz [↗](#)
- Meine Zusammenarbeit im Team [↗](#)
- Meine Projekte mit Teams abbilden [↗](#)
- Intranet mit SharePoint Online [↗](#)
- Dokumentenmanagement mit SharePoint Online [↗](#)

Buchreihe öffnen [↗](#)

Neues aus unserem Verlag

- Power BI Desktop – Grundlagen [↗](#)
- Power Query für Excel – Grundlagen [↗](#)
- Excel 2024 – Grundlagen [↗](#)
- Word 2024 – Grundlagen [↗](#)
- PowerPoint 2024 – Grundlagen [↗](#)
- Überzeugend präsentieren mit PowerPoint 2024 [↗](#)

zu den Neuerscheinungen [↗](#)

C# – Grundlagen der Programmierung

mit Visual Studio 2022



Entdecken Sie die HERDT-Themenvielfalt



WINDOWS



OFFICE



COPILOT



APPLE



PYTHON



PROGRAMMIERUNG



PHOTOSHOP



INDESIGN



ILLUSTRATOR



CHATGPT



CLIENT/SERVER



ACROBAT PDF



Hochschule Aalen - Technik und Wirtschaft

In Kooperation mit dem HERDT-Verlag stellen wir Ihnen eine PDF inkl. Zusatzmedien für Ihre persönliche Weiterbildung zur Verfügung. In Verbindung mit dem Programm HERDT-Campus ALL YOU CAN READ stehen diese PDFs für Forschung und Lehre nur Mitarbeiterinnen und Mitarbeitern, Studierenden sowie Walk-in-Usern der oben genannten Hochschule zur Verfügung. Eine Nutzung oder Weitergabe für andere Zwecke ist ausdrücklich verboten und unterliegt dem Urheberrecht. Jeglicher Verstoß kann zivil- und strafrechtliche Konsequenzen nach sich ziehen.

C# – Grundlagen der Programmierung

mit Visual Studio 2022

Ralph Steyer

1. Ausgabe, Juni 2022

ISBN 978-3-98569-075-6

VCSPNET_2022



Mit Visual Studio arbeiten

1 Informationen zu diesem Buch	4	6.13 Konstanten	60
1.1 Voraussetzungen und Ziele	4	6.14 Arithmetische Operatoren sowie Vorzeichen- und Verkettungsoperatoren	61
1.2 Aufbau und Konventionen	4	6.15 Logische Operatoren	65
1.3 Bevor Sie beginnen ...	5	6.16 Zuweisungsoperatoren für eine verkürzte Schreibweise verwenden	67
2 Visual C# und .NET	6	6.17 Der Operator <code>nameof</code>	68
2.1 Visual C# und seine Entwicklungsumgebung	6	6.18 Übungen	69
2.2 Programme gemäß .NET entwickeln	8		
2.3 Aufgaben einer Entwicklungsumgebung	10		
3 Projekte in Visual Studio	11	7 Kontrollstrukturen	70
3.1 Grundlagen zu Projekten und Projektmappen	11	7.1 Was Kontrollstrukturen einsetzen	70
3.2 Projekte erstellen und speichern	12	7.2 Einseitige Auswahl	71
3.3 Projekte schließen und öffnen	16	7.3 Zweiseitige Auswahl	72
3.4 Projekte in der Entwicklungsumgebung starten	17	7.4 Mehrstufige Auswahl	74
3.5 Mit Dateien im Projektmappen-Explorer arbeiten	17	7.5 Mehrseitige Auswahl (Fallauswahl)	76
3.6 Übung	18	7.6 Schleifen (Wiederholungen)	78
4 Anwendungen erstellen	19	7.7 Kopfgesteuerte <code>while</code> -Anweisung	79
4.1 Grundlagen der Anwendungserstellung	19	7.8 Fußgesteuerte <code>do-while</code> -Anweisung	81
4.2 Eine Windows-Anwendung erstellen	20	7.9 Zählergesteuerte Wiederholung	82
4.3 Mit Ereignissen den Ablauf steuern	22	7.10 Weitere Kontrollstrukturen	84
4.4 Ereignismethode festlegen	23	7.11 Codeausschnitte zu Kontrollstrukturen	85
4.5 IntelliSense nutzen	25	7.12 Übungen	85
4.6 Codeausschnitte einfügen	27		
4.7 Konsolenanwendungen erstellen	29		
4.8 Übung	30		
5 Benutzeroberflächen gestalten	31	Objektorientiert programmieren	
5.1 Grundlegende Bearbeitung	31		
5.2 Positionierhilfen nutzen	33		
5.3 Weitere Möglichkeiten	36		
5.4 Projekte mit mehreren Formularen	37		
5.5 Übung	39		
6 Sprachgrundlagen von C#	40	8 Klassen, Felder und Methoden	87
6.1 Was ist die Syntax?	40	8.1 Grundlagen der objektorientierten Programmierung	87
6.2 Bezeichner und Schlüsselwörter	40	8.2 Klassen und Instanzen	88
6.3 Aufbau eines Programms	42	8.3 Methoden – die Funktionalität der Klassen	92
6.4 Programmcode dokumentieren	43	8.4 Einfache Methoden erstellen	93
6.5 Anweisungen in C# erstellen	47	8.5 Methoden mit Parametern erstellen	96
6.6 Einfache Datentypen	48	8.6 Methoden mit Rückgabewert erstellen	101
6.7 Referenztypen	50	8.7 Ausgabeparameter verwenden	103
6.8 Literale	51	8.8 Vordefinierte Methoden nutzen	104
6.9 Mit Variablen arbeiten	52	8.9 Methoden überladen	105
6.10 Werte zuweisen	55	8.10 Erweiterungsmethoden	107
6.11 Tipps zur Arbeit mit Variablen	55	8.11 Rekursion	108
6.12 Typkompatibilität und Typkonversion	57	8.12 Übungen	110

Mit C# programmieren

6 Sprachgrundlagen von C#	40	9 Kapselung, Konstruktoren und Namensräume	112
6.1 Was ist die Syntax?	40	9.1 Kapselung	112
6.2 Bezeichner und Schlüsselwörter	40	9.2 Eigenschaften	112
6.3 Aufbau eines Programms	42	9.3 Konstruktoren und Destruktoren	116
6.4 Programmcode dokumentieren	43	9.4 Statische Member und statische Klassen	121
6.5 Anweisungen in C# erstellen	47	9.5 Namensräume	123
6.6 Einfache Datentypen	48	9.6 Partielle Klassen erstellen	127
6.7 Referenztypen	50	9.7 Partielle Methoden	127
6.8 Literale	51	9.8 Informationen zu Klassen erhalten	129
6.9 Mit Variablen arbeiten	52	9.9 Übungen	130
6.10 Werte zuweisen	55		
6.11 Tipps zur Arbeit mit Variablen	55		
6.12 Typkompatibilität und Typkonversion	57		

10 Vererbung	132	13.13 Hash-Tabelle	188
10.1 Grundlagen zur Vererbung	132	13.14 Mit Aufzählungstypen arbeiten	189
10.2 Klassen ableiten und erweitern	132	13.15 Strukturen	190
10.3 Vererbungsketten	134	13.16 Speicherverwaltung	193
10.4 Mit Klassendiagrammen arbeiten	136	13.17 Strukturen und Klassen	194
10.5 Übungen	141	13.18 Übungen	195
11 Polymorphismus	143	Weitere Möglichkeiten	
11.1 Polymorphie in der Vererbung	143	14 Fehlerbehandlung und Fehlersuche	197
11.2 Member verbergen	144	14.1 Fehlerarten	197
11.3 Member überschreiben	145	14.2 Strukturierte Fehlerbehandlung	199
11.4 Member überladen	148	14.3 Eigene Ausnahmen erzeugen	201
11.5 Abstrakte Klassen und Methoden	149	14.4 Fehler aufspüren und beseitigen	202
11.6 Typprüfung und -konvertierung	152	14.5 Programmablauf kontrollieren	204
11.7 Operatoren überladen	155	14.6 Programme an einer bestimmten Stelle anhalten	205
11.8 Übungen	157	14.7 Prüfen und Korrigieren	206
12 Schnittstellen (Interfaces)	159	14.8 Aufruferinformationen auswerten	208
12.1 Einführung zu Schnittstellen	159	14.9 Übungen	210
12.2 Schnittstellen deklarieren	160		
12.3 Schnittstellen implementieren	160		
12.4 Member einer Schnittstelle verdecken	164	15 System-, Datei- und Laufwerkszugriffe	211
12.5 Typprüfung und -konvertierung	165	15.1 Systemzugriffe über Klassen des .NET Frameworks	211
12.6 Übungen	167	15.2 Klassen für den Dateizugriff	212
13 Komplexe Datentypen	168	15.3 Mit Laufwerken, Ordnern und Dateien arbeiten	212
13.1 Eindimensionale Arrays	168	15.4 Mit Textdateien arbeiten	214
13.2 Mehrdimensionale und verzweigte Arrays	172	15.5 Übungen	217
13.3 Mit Arrays arbeiten	173		
13.4 Parameter-Arrays	174		
13.5 Auflistungen	178	16 Anwendungen weitergeben	219
13.6 Die Klasse <code>ArrayList</code>	178	16.1 Voraussetzungen für die Weitergabe	219
13.7 Auflistungsinitialisierer	180	16.2 Weitergeben durch Kopieren	220
13.8 Listen mit einem Enumerator durchlaufen	180	16.3 Anwendungen mit Visual Studio veröffentlichen	220
13.9 Typsicherheit bei Auflistungen und Generics	182	16.4 Übung	223
13.10 Indexer	183		
13.11 Warteschlangen	185		
13.12 Stapel	187		
		Stichwortverzeichnis	224

1 Informationen zu diesem Buch

1.1 Voraussetzungen und Ziele

Zielgruppe

Dieses Buch richtet sich an Neueinsteiger sowie Umsteiger, die die Anwendungsentwicklung mittels der Programmiersprache C# erlernen möchten.

Empfohlene Vorkenntnisse

Für das Programmieren mit C# sollten Sie über folgende Grundkenntnisse verfügen:

- ✓ PC-Grundlagenkenntnisse
- ✓ Grundkenntnisse im Umgang mit Windows ab Version 10 bzw. 11 (eingeschränkt Windows 7)

Hinweise zu Soft- und Hardware

In den Funktionsbeschreibungen des Buches wird von einer Erstinstallation der Software Microsoft Visual Studio und des darin enthaltenen Microsoft Visual C# 2022 in der Community Edition unter dem Betriebssystem Windows 10 bzw. 11 und den Standardeinstellungen für diese Programme ausgegangen. Allerdings sollten sich hinsichtlich C# weder bei der Verwendung eines anderen Betriebssystems ab Windows 7 noch bei einer anderen Version von Visual Studio ab der Version 2010 größere Abweichungen ergeben. Abhängig von der Version des Betriebssystems, Visual Studio und dessen Einstellungen sowie von der Bildschirmauflösung bzw. der Hardware Ihres Computers kann das Aussehen der Symbole und Schaltflächen in den Programmen und die Fensterdarstellung unter Windows ggf. von den Abbildungen im Buch abweichen. Einige Funktionalitäten von Visual Studio werden auch über die Entwicklung des Programms verlagert, umstrukturiert oder erweitert. Aber auch das ist für die reine Programmierung mit C# vollkommen irrelevant.

1.2 Aufbau und Konventionen

Aufbau des Buchs

- ✓ Das Buch ist in verschiedene Bereiche unterteilt, um Ihnen so einen Überblick über seinen Inhalt zu geben und zudem das Festlegen der Lernschwerpunkte zu erleichtern.
- ✓ Am Anfang jedes Kapitels finden Sie die Lernziele und am Ende der meisten Kapitel Übungen, mit deren Hilfe Sie die jeweiligen Inhalte einüben können.

Inhaltliche Gliederung

- ✓ Zuerst erhalten Sie eine kurze Einführung in C# und die Arbeitsweise von Visual Studio und .NET. In den frei verfügbaren BuchPlus-Dateien wird Ihnen die Entwicklungsumgebung Visual Studio vorgestellt, da Sie diese von Beginn an für die Erstellung der Beispiele nutzen werden.
- ✓ Danach werden Sie mit den Anwendungen vertraut gemacht, die in diesem Buch genutzt werden.
- ✓ Der Hauptteil des Buches widmet sich der Sprache C#.
- ✓ Am Ende des Buches werden noch einige ausgewählte Themen wie das Debuggen und der Dateizugriff behandelt.

Typografische Konventionen

Im Text erkennen Sie bestimmte Programmelemente an der Formatierung. So werden z. B. Bezeichnungen für Programmelemente wie Register immer *kursiv* geschrieben und wichtige Begriffe **fett** hervorgehoben.

<i>Kursivschrift</i>	kennzeichnet alle von Programmen vorgegebenen Bezeichnungen für Schaltflächen, Dialogfenster, Symbolleisten, Menüs bzw. Menüpunkte (z. B. <i>Datei - Schließen</i>) sowie alle vom Anwender zugewiesenen Namen wie Dateinamen, Ordnernamen, eigene Symbolleisten, Hyperlinks und Pfadnamen.
<i>Courier New</i>	kennzeichnet Programmtext, Programmnamen, Funktionsnamen, Variablennamen, Datentypen, Operatoren etc.
<i>Courier New kursiv</i>	kennzeichnet Zeichenfolgen, die vom Anwendungsprogramm ausgegeben oder in das Programm eingegeben werden.
[]	Bei Darstellungen der Syntax einer Programmiersprache kennzeichnen eckige Klammern optionale Angaben.
	Bei Darstellungen der Syntax einer Programmiersprache werden alternative Elemente durch einen senkrechten Strich voneinander getrennt.

Was bedeuten die Symbole im Buch?



Praxistipp



Warnhinweis

1.3 Bevor Sie beginnen ...

HERDT BuchPlus – unser Konzept:

Problemlos einsteigen – Effizient lernen – Zielgerichtet nachschlagen

(weitere Infos unter www.herdt.com/BuchPlus)

Nutzen Sie dabei unsere maßgeschneiderten, im Internet frei verfügbaren Medien:



Wie Sie schnell auf diese BuchPlus-Medien zugreifen können, erfahren Sie unter www.herdt.com/BuchPlus.

Beachten Sie, dass es bei einem Teil der Beispiele und Übungen zu Problemen mit Zugriffsrechten kommen kann, wenn sie von einem Netzlaufwerk aus gestartet werden. Es ist daher zu empfehlen, die Beispiele und Übungen auf ein Laufwerk des Rechners zu kopieren und sie von dort aus zu starten.



2 Visual C# und .NET

2.1 Visual C# und seine Entwicklungsumgebung

Was ist Visual C#?

Visual C# (sprich: si scharp) ist eine objektorientierte Programmiersprache von Microsoft. Mit Visual C#, das zum Zeitpunkt der Erstellung des Buches in der Version 10 vorliegt (C# 10), können Sie unter anderem folgende Anwendungstypen entwickeln:

- ✓ Anwendungsprogramme für Windows und die Konsole (Eingabeaufforderung)
- ✓ Webanwendungen (ASP)
- ✓ Anwendungen (Apps) für mobile Geräte
- ✓ Anwendungen für Windows
- ✓ Portable Klassenbibliotheken für die Verwendung auf unterschiedlichen Systemen

Visual C#-Programme mit Visual Studio entwickeln

Visual C# ist in die Entwicklungsumgebung **Visual Studio** von Microsoft (vgl. Abschnitt 2.3) eingebettet.

In Visual Studio sind verschiedene Programmiersprachen vereint:

- | | |
|----------------|--------------|
| ✓ Visual Basic | ✓ Visual C++ |
| ✓ Visual C# | ✓ Visual F# |

Dazu gibt es Unterstützung für XAML (Extensible Application Markup Language) und zahlreiche Webtechnologien wie JavaScript, HTML (Hypertext Markup Language), CSS (Cascading Style Sheet) und andere.

Strategisch soll die weitere Entwicklung der Sprachen von .NET allerdings auf Visual C# und Visual C++ beschränkt werden.

Normalerweise verzichtet man bei der Benennung der Sprachen C#, C++ und F# auf das vorangestellte „Visual“. Grundsätzlich aber gilt, dass Visual C# und C# (analog die anderen Fälle) die gleiche Programmiersprache bezeichnen, Microsoft aber vermutlich über das vorangestellte „Visual“ aus Marketinggründen eine Zusammengehörigkeit der .NET-Sprachen und Visual Studio suggerieren möchte. Bei Programmierern wird allerdings so gut wie immer auf das vorangestellte „Visual“ verzichtet. Selbst in Visual Studio taucht an fast allen Stellen nur die Bezeichnung C# auf. Nur in einem Fall ist ein vorangestelltes „Visual“ üblich – bei **Visual** Basic oder kurz VB.

Für alle enthaltenen Sprachen stellt Visual Studio die Entwicklungsumgebung zur Verfügung. Visual Studio zeichnet sich dadurch aus, dass diese Entwicklungsumgebung **programmiersprachenunabhängig** ist.

Visual Studio und .NET

Visual Studio ist eine Entwicklungsumgebung für das **.NET Framework** von Microsoft, das zum Zeitpunkt der Erstellung des Buches in der Version 6 vorliegt. .NET (sprich: dotnet) beschreibt eine Softwareumgebung (Framework) zur Entwicklung **programmiersprachen- und plattformunabhängiger** Software, in dem der Programmcode verschiedener Programmiersprachen mittels eines Compilers in eine Zwischensprache übersetzt wird, die für alle Programmiersprachen gleich ist. Diese Zwischensprache ist plattformneutral und kann auf unterschiedlichen Betriebssystemen wie Windows oder Linux ausgeführt werden.

**Ergänzende Lerninhalte: Neuerungen.pdf**

Hier finden Sie einen Überblick über die Neuerungen in Visual Studio, bei C# und dem .NET-Framework.

Vorteile der Programmiersprachenunabhängigkeit von .NET

- ✓ Durch das Konzept der Zwischensprache sind alle .NET-Sprachen gleichwertig, auch in Bezug auf die Ausführungsgeschwindigkeit.
- ✓ Ein Vorteil liegt außerdem darin, dass Entwickler mit **verschiedenen** Programmiersprachen an **einem** Projekt arbeiten können. Dies ist z. B. dann sinnvoll, wenn eine neue Anwendung mit C# entwickelt werden soll, aber noch alte Programmteile in C++ oder Visual Basic vorliegen. Allerdings sollten in neuen Projekten nicht von vornherein mehrere Programmiersprachen verwendet werden.

Aufbau des .NET Frameworks

Das **.NET Framework**, auch kurz **.NET** genannt, beinhaltet:

- ✓ eine umfangreiche Klassenbibliothek (FCL – **Framework Class Library**) zur Verwendung mit dem Programmcode;
- ✓ eine Laufzeitumgebung (CLR – **Common Language Runtime**), die z. B. für die Ausführung der durch den Compiler erzeugten Zwischensprache verantwortlich ist.

.NET Core versus .NET Framework sowie .NET 6

Seit einigen Jahren wird man bei C# und den anderen Techniken und Sprachen des Frameworks mit dem Begriff **.NET Core** konfrontiert und auch in neueren Versionen von Visual Studio taucht dieser Begriff bei Varianten vor der Version 2022 an verschiedenen Stellen auf (insbesondere bei Projektvorlagen). Es stellt sich die Frage, wie .NET Core im Kontext des .NET Frameworks zu sehen ist.

Vereinfacht gesagt ist .NET Core die Zukunft des Frameworks! Das zeigt sich ab Visual Studio 2022 indirekt daran, dass bei den Projektvorlagen der Bezeichner „Core“ – im Gegensatz zu neueren Versionen von Visual Studio bis zur Version 2019 – fast gar nicht mehr bei den Namen der Projektvorlagen auftaucht, sondern stattdessen nur noch explizit **.NET Framework** in Klammern. Das zeigt die Zielrichtung. Das als .NET Core eingeführte API (Application Programming Interface) soll der Standardfall werden und man muss sich als Programmierer bewusst für eine .NET Framework-Version entscheiden, wenn es dafür Gründe gibt. In den Unterlagen werden wir aber den Bezeichner .NET Core wegen der Klarheit verwenden, sofern eine Differenzierung zu .NET Framework notwendig ist.

Das eigentliche .NET-Framework gibt es seit Anfang der 2000er Jahre und es wird als sehr ausgereift und sehr stabil angesehen. Viele Projekte arbeiten damit und das .NET-Framework wird auch von Microsoft weiterhin unterstützt. Allerdings ist das Framework auf Grund vieler Unternehmensprojekte, die davon abhängen, unflexibel. Erweiterungen und Neuerungen müssen auf jeden Fall abwärtskompatibel sein und Updates dürfen nicht in zu kurzen Zyklen erscheinen. Ebenso ist das .NET-Framework nicht portabel, sondern nur weitgehend auf die Windows-Plattform beschränkt, und sehr schwergewichtig.

.NET Core ist hingegen die Vision von Microsoft für die Zukunft, denn ...

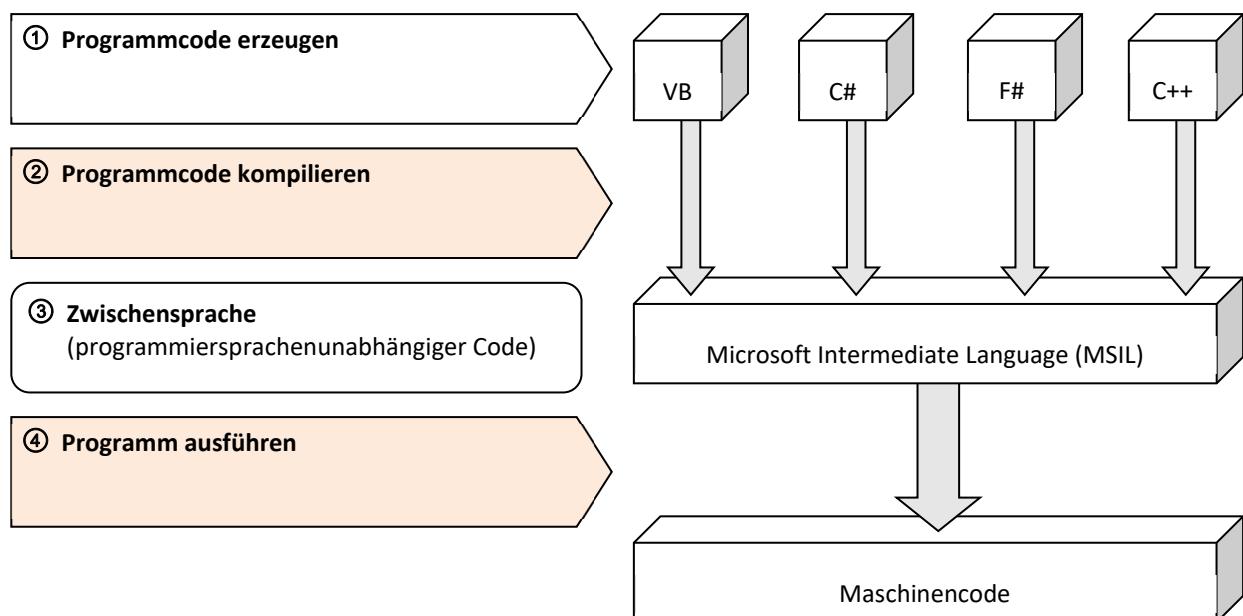
- ✓ Änderungen und Neuerungen werden in .NET Core durch einen sehr schnellen Release-Zyklus zeitnah bereitgestellt,
- ✓ das System ist modular,
- ✓ skalierbar,
- ✓ performant und
- ✓ .NET Core ist plattformübergreifend.

Allerdings sind in .NET Core noch nicht alle Features verfügbar, die es im etablierten .NET-Framework gibt. Aber es werden mit jedem neuen Update immer mehr Features bereitgestellt. Sehr viel ist bereits mit .NET 6 passiert. Allgemein gilt aber, dass .NET Core teils noch experimentell und .NET-Framework dagegen als Basis oft zuverlässiger ist. Aber es ist absehbar, dass die Vereinheitlichung, die das .NET 6-API bereits bringt und kommende Versionen sicher forcieren, sehr bald das bisher .NET Core genannte API zum Regelfall werden lässt.

Für den Stoff in dem Buch ist es zudem fast immer irrelevant, ob wir mit .NET Core oder dem .NET-Framework als Basis arbeiten. Die hier behandelten Techniken von C# sind identisch in beiden Welten vorhanden. Nur in Ausnahmefällen kann es sein, dass bei Verwendung von .NET Core noch ein paar Features fehlen.

Die Beispiele in dem Buch basieren explizit auf der Verwendung des etablierten .NET-Frameworks.

2.2 Programme gemäß .NET entwickeln



① Programmcode erzeugen

Programmcode kann in verschiedenen .NET-Programmiersprachen, wie z. B. Visual Basic, C# und C++, erstellt werden.

② Programmcode kompilieren

Mithilfe integrierter entsprechender Compiler wird der Programmcode in eine **einheitliche** Zwischensprache übersetzt.

③ Zwischensprache

Die Zwischensprache ist für alle .NET-Programmiersprachen gleich. Sie wird **Microsoft Intermediate Language** (MSIL) oder allgemein **Common Intermediate Language** (CIL) oder manchmal auch nur ganz kurz IL genannt.

- ✓ Der MSIL-Code ist programmiersprachenunabhängig.
- ✓ Alle .NET-Programmiersprachen müssen bestimmten Richtlinien, den **CLS** (Common Language Specifications), entsprechen. Dazu gehört auch ein einheitliches Datentypsysteem **CTS** (Common Type System).

④ Programm ausführen

- ✓ Der Code der Zwischensprache wird bei der Programmausführung nicht interpretiert, sondern durch einen JIT(Just In Time)-Compiler in den Maschinencode des entsprechenden Prozessors übersetzt.
- ✓ Die Laufzeitumgebung übernimmt nicht nur das Übersetzen und Optimieren des Programmcodes, sondern auch die Sicherheits- und Versionsprüfung sowie die Fehler- und Speicherverwaltung. Deshalb wird der auf diese Weise behandelte Programmcode auch „Managed Code“ (verwalteter Programmcode) genannt. Er lässt sich sicherer und stabiler ausführen als herkömmlicher „Unmanaged Code“.

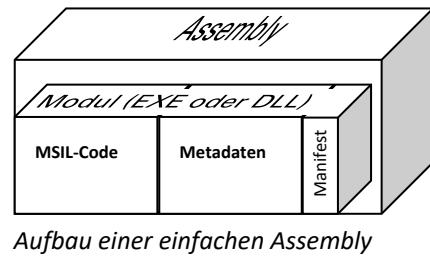
Dynamic Language Runtime

Die Besonderheit einer Sprache wie C# ist, dass alle Datentypen bereits zur Übersetzungszeit bekannt sein müssen. Die Einbindung von sogenannten dynamischen Sprachen wie z. B. JavaScript fällt dadurch schwer, z. B. Bibliotheken, die bereits in JavaScript vorliegen und von C# aus genutzt werden sollen. Bereits mit dem .NET Framework 4.0 wurde die Dynamic Language Runtime eingeführt, die eine bessere Verwendung und Unterstützung solcher Sprachen ermöglicht.

.NET-Anwendungen in Assemblies verwalten

.NET-Anwendungen bestehen aus sogenannten **Assemblies** (Sammlungen). Sobald ein Programm kompiliert wird, liegt ein verwaltetes Modul vor, das zu einer Assembly gehört.

- ✓ Eine Assembly beinhaltet ein verwaltetes Modul (Single-File-Assembly).
- ✓ Das verwaltete Modul muss eine EXE- oder DLL-Datei sein.
- ✓ Das Modul enthält den Programmcode (MSIL-Code) und Metadaten zur Beschreibung des Moduls.
- ✓ Außerdem beinhaltet das Modul ein sogenanntes Manifest zur Beschreibung der gesamten Assembly.



Eine Assembly kann ggf. Ressourcen-Dateien, wie z. B. Bilder, Datenbankdateien und Sounddateien, oder auch mehrere verwaltete Module (Multi-File-Assembly) enthalten. Eines der verwalteten Module muss eine EXE- oder DLL-Datei sein und ein Manifest beinhalten.

Manifest – eine Assembly beschreiben

Das Manifest einer Assembly speichert folgende Informationen:

- ✓ den Namen der Assembly,
- ✓ die Versionsnummer,
- ✓ die Länderinformationen (Culture-Info),
- ✓ eine Liste der enthaltenen Module, Typen und Ressourcen,
- ✓ eine Liste referenzierter Assemblies und deren Versionsnummern.

Durch die Assembly-Technik wird das Registrieren von Komponenten und Versionsnummern in der System-registry überflüssig. Eine Anwendung lässt sich einfach kopieren und ist sofort lauffähig.

2.3 Aufgaben einer Entwicklungsumgebung

Eine Entwicklungsumgebung wie Visual Studio wird auch als **IDE (Integrated Development Environment)** bezeichnet. Sie soll den Programmierer bei der Softwareentwicklung unterstützen und besitzt folgende Merkmale:

- ✓ komfortables Erstellen, Editieren und Übersetzen (Kompilieren) des Programmcodes,
- ✓ integrierte Fehlersuche (Debuggen),
- ✓ integriertes Testen,
- ✓ automatische Erledigung ständig wiederkehrender Aufgaben,
- ✓ individuelle Konfigurationsmöglichkeiten der verschiedenen Bereiche,
- ✓ gezielter und schneller Zugriff auf relevante Hilfethemen.



Ergänzende Lerninhalte: *Entwicklungsumgebung.pdf*

Hier erfahren Sie, wie Sie die Entwicklungsumgebung starten, damit arbeiten und wie Sie diese beenden können. Außerdem lernen Sie den Umgang mit der Hilfe kennen.

3 Projekte in Visual Studio

3.1 Grundlagen zu Projekten und Projektmappen

Was sind Projekte?

Als Projekt wird in Visual Studio die Zusammenstellung aller Informationen bezeichnet, die für die Entwicklung beispielsweise einer Anwendung benötigt werden. Dazu gehören u. a.:

- ✓ Dateien mit Programmcode
- ✓ Dateien mit der Definition von Formularen (z. B. von Dialogfenstern)
- ✓ Verweise auf Programmmodulen, die bereits existieren und verwendet werden sollen
- ✓ das fertige ausführbare Programm

Projekttypen und Projektvorlagen

Je nach Einsatzgebiet wird zwischen verschiedenen Projekttypen unterschieden. Zu den Projekttypen, die Sie erstellen können, gehören beispielsweise:

Projekte mit grafischer Benutzeroberfläche

- ✓ Windows-Anwendungen, aber auch immer besser unterstützte Anwendungen für Linux oder macOS
- ✓ WPF (**Windows Presentation Foundation**)- und WCF (**Windows Communication Foundation**)-Anwendungen
- ✓ Web-Anwendungen (Server und Client)
- ✓ Windows Store Apps

Projekte ohne grafische Benutzeroberfläche

- ✓ Konsolenanwendungen
- ✓ Windows-Dienste

Programmkomponenten und sonstige Projektarten

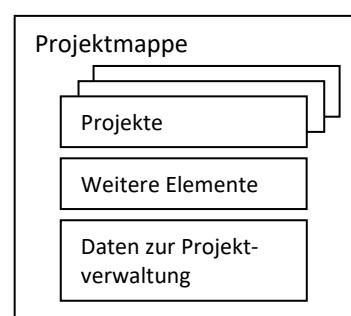
- ✓ Klassenbibliotheken
- ✓ Neue Steuerelemente
- ✓ Setup-Projekte (Erstellung eigener Installationsroutinen für Ihre C#-Anwendungen)
- ✓ Projekte für viele weitere Sprachen wie Python, JavaScript, SQL und mehr

Für die jeweiligen Projekttypen sind im Lieferumfang von Visual Studio **Projektvorlagen** enthalten, die Ihnen jeweils ein Grundgerüst für den ausgewählten Projekttyp bereitstellen.

Projekte in Projektmappen organisieren

In Visual Studio werden Projekte innerhalb sogenannter **Projektmappen** abgelegt.

- ✓ Beim Erstellen eines Projekts wird automatisch auch eine Projektmappe erzeugt, in die das neue Projekt eingefügt wird.
- ✓ Eine Projektmappe kann mehrere Projekte enthalten.
- ✓ Mindestens ein Projekt ist ein sogenanntes Startprojekt. Es wird im Falle mehrerer Projekte in der Projektmappe ausgeführt, wenn kein konkretes Projekt ausgewählt ist.



- ✓ Zu einer Projektmappe können weitere Elemente, z. B. Grafiken oder Textdateien, hinzugefügt werden, die nicht einem bestimmten Projekt zugeordnet werden können.
- ✓ Standardmäßig wird für eine Projektmappe ein gleichnamiger Ordner angelegt, in dem die enthaltenen Projekte jeweils in einzelnen Ordnern gespeichert werden.
- ✓ Zusätzlich enthält der Projektmappenordner eine spezielle Projektmappendatei, in der die erforderlichen Einstellungen gespeichert sind, wie zum Beispiel die Information, welche Projekte und Elemente zu der Projektmappe gehören. So können z. B. Projekte einer Projektmappe auch in unterschiedlichen Ordnern abgelegt werden.

3.2 Projekte erstellen und speichern

Neue Projekte erzeugen und speichern

Wenn Sie Visual Studio 2022 starten, bekommen Sie vor dem endgültigen Start der IDE einen Dialog mit bereits erstellten Projekten sowie eine Schaltfläche *Neues Projekt erstellen* angezeigt. Ist Visual Studio jedoch bereits gestartet, können Sie so vorgehen:

- Rufen Sie den Menüpunkt *Datei - Neu - Projekt* auf.

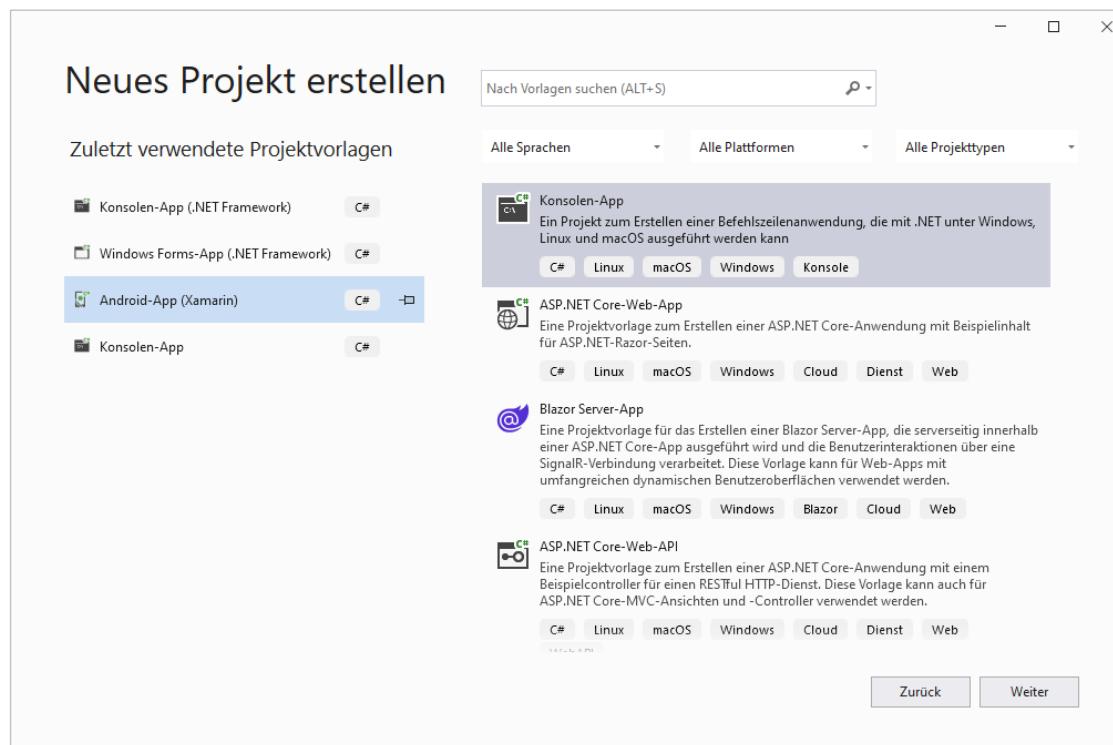
Alternativen:  oder 

oder Wählen Sie den entsprechenden Hyperlink auf der Startseite, falls diese geöffnet ist.

Ist bereits ein Projekt geöffnet, wird dieses geschlossen. Sofern Sie im aktuellen Projekt Änderungen vorgenommen und diese noch nicht gespeichert haben, werden Sie aufgefordert, das aktuelle Projekt zu speichern oder die Änderungen zu verwerfen.

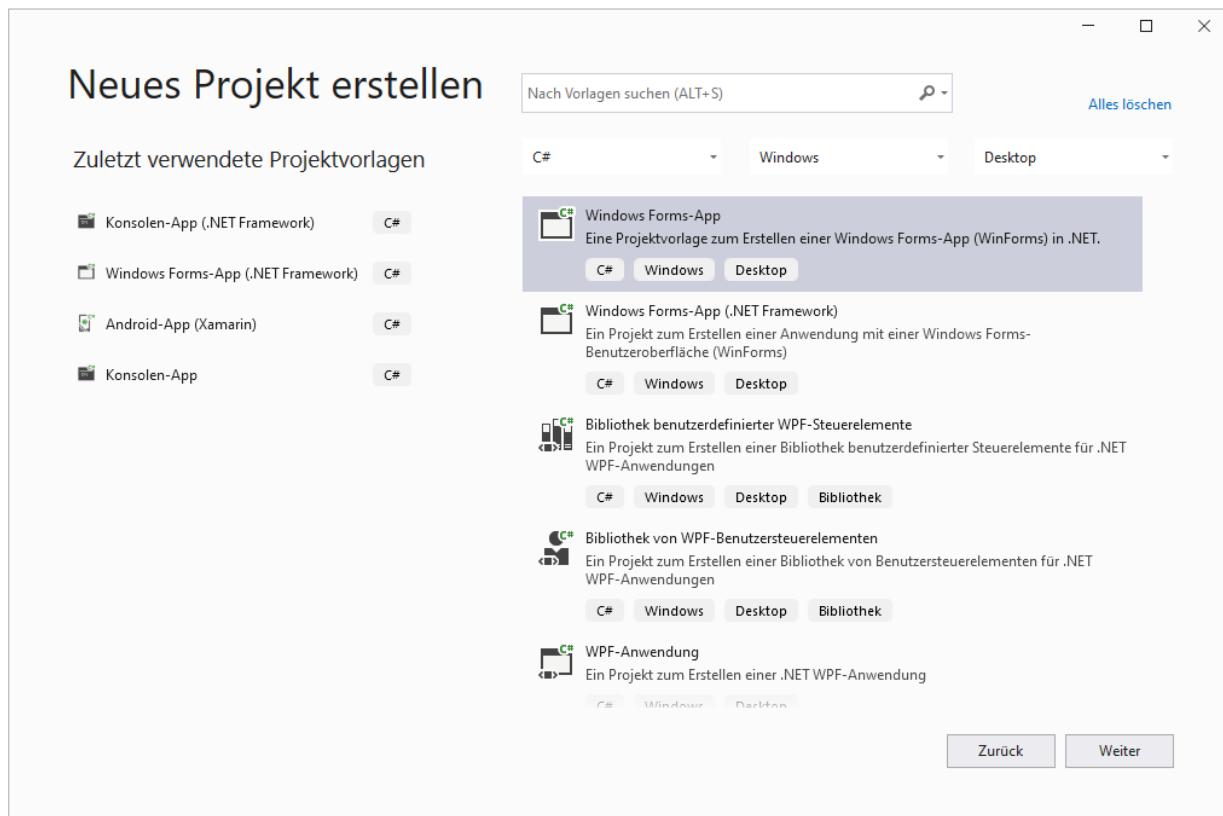
Anschließend wird das Dialogfenster *Neues Projekt erstellen* geöffnet, das in der Version 2022 von Visual Studio gegenüber der Version 2019 kaum, jedoch dessen Vorgängerversionen erheblich erweitert und umstrukturiert wurde. Aber prinzipiell hat sich innerhalb der letzten Versionen von Visual Studio nichts geändert.

Beachten Sie, dass sich das genaue Aussehen des Dialogs und die verfügbaren Vorlagen von den Abbildungen im Buch unterscheiden können. Insbesondere hängt dies an den installierten Features, die oft auch optional sind.



Ein neues Projekt erstellen

- ▶ Durch den mittlerweile sehr großen Umfang an vorhandenen Projektvorlagen kann es sinnvoll sein, dass Sie im Suchfeld für die Vorlagen zumindest die Kategorie für ihr gewünschtes Projekt oder auch schon konkret eine Vorlage suchen. Aber Sie können auch mit den Bildlaufleisten zur passenden Vorlage scrollen. Falls Sie bereits Projektvorlagen verwendet haben, werden Ihnen diese unter *Zuletzt verwendete Projektvorlagen* angezeigt.
- ▶ Hilfreich ist auch die Einschränkung der Sprache (in unserem Fall C#) und Plattform (in unserem Fall Windows), um die Menge der infrage kommenden Vorlagen zu reduzieren und die Auswahl übersichtlicher zu gestalten. Gegebenenfalls schränken Sie weiter über die Projektypen ein. In diesem Buch spielen nur Desktop- und Konsolen-Projektypen eine Rolle.
- ▶ Suchen und wählen Sie für unser erstes Beispielprojekt im Dialogfenster *Neues Projekt erstellen* den Eintrag *Windows Forms App (.NET Framework)* als gewünschten Projekttyp. Damit wird eine *Windows Forms-Anwendung* erstellt. Sie können auch den Eintrag *Windows Forms App (.NET Core)* (bzw. ohne die Kennzeichnung **Core** in neuen Versionen von Visual Studio – das ist dann implizit Core) verwenden. Wie in Kapitel 2 besprochen, ist .NET Core die Vision von Microsoft für die Zukunft des Frameworks, aber diese technischen Basisstrukturen wirken sich in unseren Beispielen nicht aus. Allgemein gilt aber, dass .NET Core oft experimentell ist und .NET-Framework als Basis – gerade im Zusammenspiel mit Visual Studio oft zuverlässiger ist. Im Buch wählen wir immer .NET-Framework.

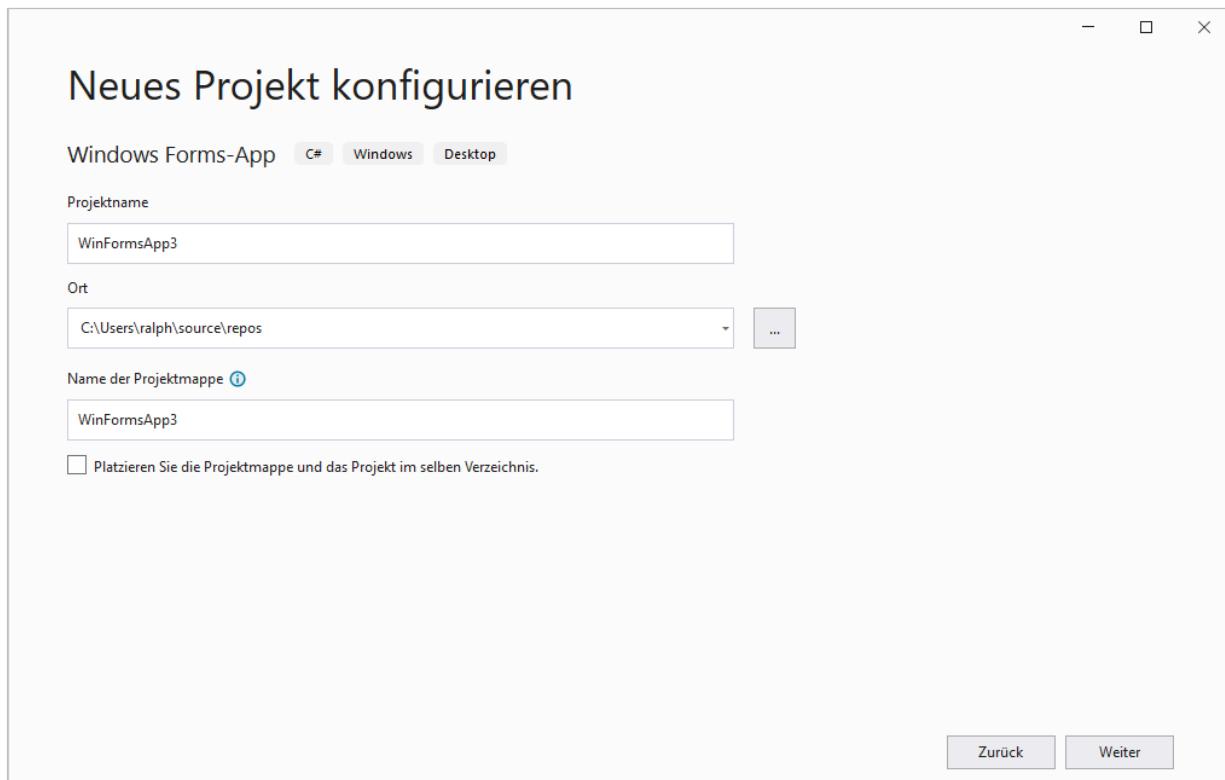


Eine Projektvorlage auswählen – entweder auf Basis von .NET Core oder .NET-Framework

Die für das Projekt ausgewählte Programmiersprache (z. B. Visual C#) kann nachträglich nicht geändert werden.

- ▶ Klicken Sie auf *Weiter*. Sie kommen zu einem neuen Dialogfenster *Neues Projekt konfigurieren*.





Das Projekt konfigurieren

- ▶ Legen Sie im Eingabefeld *Projektname* den Projektnamen fest.
- ▶ Geben Sie den Ordner an, in dem die Projektmappe angelegt werden soll.
Als Speicherort wird der Standardprojektordner vorgegeben.
- ▶ Ändern Sie – falls erforderlich – den Namen der Projektmappe.
Standardmäßig erhält die Projektmappe denselben Namen wie das Projekt.
- oder* Sie können auch über die Auswahl ein neues Projekt in der aktuell geöffneten Projektmappe oder einer neuen Projektmappe unterbringen. Bestätigen Sie mit *Weiter*.
- ▶ Im nächsten Schritt können Sie bei Bedarf unter *Framework* die Version des .NET-Frameworks auswählen, die Ihnen geeignet erscheint. In der Regel können Sie bei den Voreinstellungen bleiben.
- ▶ Bestätigen Sie mit *Erstellen*.

Ablage der Projektmappe im Dateisystem

Das neue Projekt wird sofort gespeichert:

Dieser PC > Lokaler Datenträger (C:) > Benutzer > ralph > source > repos > WindowsFormsApp1①			
Name	Änderungsdatum	Typ	Größe
.vs	04.03.2020 08:31	Dateiordner	
WindowsFormsApp1 ②	04.03.2020 08:31	Dateiordner	
WindowsFormsApp1.sln ③	04.03.2020 08:31	Microsoft Visual S...	2 KB

- ✓ In dem als *Speicherort* angegebenen Ordner wird ein neuer Unterordner mit dem Namen der Projektmappe ① angelegt.
- ✓ Darin werden der Ordner mit dem Namen des Projekts ② und die Projektappendatei ③ gespeichert. Die Projektappendatei erhält die Dateinamenerweiterung *.sln* (Solution).

- ✓ Der Projektordner ① enthält weitere Ordner und Dateien, die Sie an anderer Stelle in diesem Buch kennenlernen. In ihnen werden z. B. Formulare, Projekteinstellungen und auch die kompilierten Dateien gespeichert.

Standardmäßig ist im Dialogfenster *Neues Projekt* das Kontrollfeld *Projektmappenverzeichnis erstellen* aktiviert. Wenn Sie das Kontrollfeld deaktivieren, wird für die Projektmappe kein eigener Ordner angelegt. Der Projektordner wird dann direkt in dem als Speicherort angegebenen Ordner erstellt und die Projektmappendatei wird im Projektordner gespeichert.

Vor allem, wenn Sie an mehreren zusammengehörigen Projekten arbeiten, sollten Sie das Kontrollfeld aktiviert lassen, um diese Projekte übersichtlich zusammen in einem Projektmappenordner zu speichern.

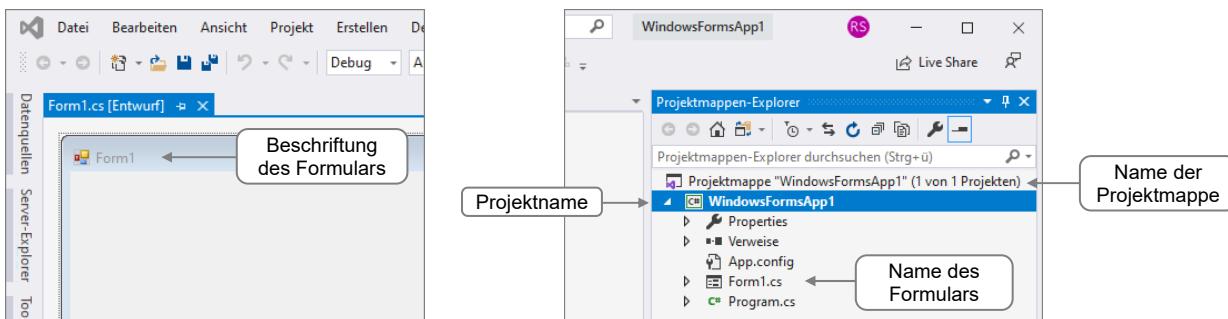


Das neu erzeugte Projekt

Das neue Projekt wird gespeichert und in Visual Studio geöffnet. Bei einer Windows Forms-Anwendung wird ein Standardformular im Windows Forms-Designer angezeigt.

Der Projektmappen-Explorer zeigt u. a.:

- ✓ den Namen der Projektmappe,
- ✓ den Projektnamen,
- ✓ das in dem Projekt enthaltene Formular *Form1* und das Startprogramm *Program.cs* [steht für *C#(Sharp)*].



In Visual Studio wird einem Projekt zusätzlich eine Anwendungskonfigurationsdatei mit dem Namen *App.config* hinzugefügt. Darin können Sie Einstellungen für die Anwendung vornehmen, welche diese wiederum mittels der Möglichkeiten des .NET Frameworks auslesen kann. Nach der Übersetzung des Projekts wird diese Datei unter dem Namen *[ProjektName].exe.config* im Ausgabeverzeichnis abgelegt.

Projekt nochmals (zwischendurch) speichern

Während der Arbeit an einem Projekt sollten Sie das Projekt regelmäßig speichern, um beispielsweise bei einem ungeplanten Programmabbruch größere Datenverluste zu vermeiden.

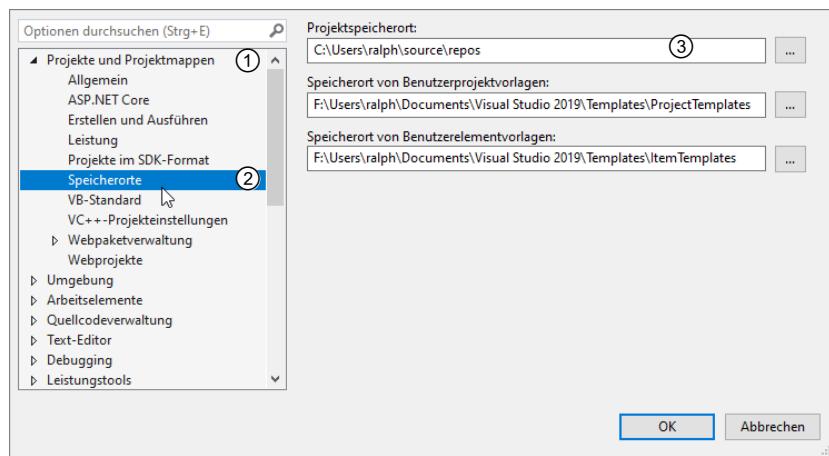
- Rufen Sie den Menüpunkt *Datei - Alle speichern* auf.

Alternativen: oder

Standardprojektordner ändern

Welcher Ordner beim Öffnen und Speichern von Projekten standardmäßig verwendet wird, ist abhängig von Ihren Einstellungen sowie den genutzten Versionen von Visual Studio und Windows. So kann es sein, dass beispielsweise der Ordner ...\\Users\\[Benutzername]\\Documents\\Visual Studio \\Projects vorgegeben ist oder aber auch der Ordner ...\\Users\\[Benutzername]\\source\\repos. Wenn Sie Ihre Projekte in einem anderen Ordner speichern möchten, können Sie den gewünschten Ordner als Standardprojektordner festlegen oder auch beim Speichern individuell auswählen.

- ▶ Rufen Sie den Menüpunkt *Extras - Optionen* auf.
- ▶ Öffnen Sie den Bereich *Projekte und Projektmappen* ①.
- ▶ Klicken Sie auf den Unterbereich *Speicherorte* ②.
- ▶ Legen Sie den Pfad des Ordners fest ③, der zum Speichern bzw. Öffnen von Projekten vorgeschlagen werden soll. Ggf. können Sie hier auch die Ordner für Vorlagen konfigurieren, aber in der Regel bleibt man bei den Standardeinstellungen.



3.3 Projekte schließen und öffnen

Projektmappe schließen

- ▶ Rufen Sie den Menüpunkt *Datei - Projektmappe schließen* auf.

Falls Sie Änderungen vorgenommen und diese noch nicht gespeichert haben, erfolgt eine entsprechende Rückfrage.

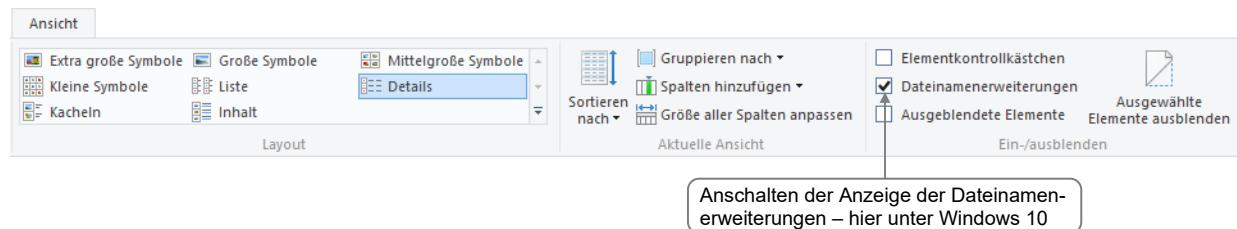
Projekte öffnen

Um ein Projekt zu öffnen, öffnen Sie die entsprechende Projektmappe.

- ▶ Rufen Sie den Menüpunkt *Datei - Öffnen - Projekt/Projektmappe* auf.
Alternative: **Strg** **O**
- ▶ Öffnen Sie im Dialogfenster *Projekt öffnen* den Ordner mit dem Namen der Projektmappe, sofern Sie Projektmappen, wie standardmäßig vorgegeben, in separaten Ordnern speichern. (Andernfalls öffnen Sie den Projektordner.)
- ▶ Klicken Sie doppelt auf den Namen der gewünschten Projektmappendatei (*.sln*).



Je nach Konfiguration Ihres Windows-Betriebssystems werden die Dateinamenerweiterungen möglicherweise nicht angezeigt. Um diese anzuzeigen, öffnen Sie im Explorer die Ordneroptionen (z. B. über den Menüpunkt *Ansicht* und dann die Schaltfläche *Optionen* (alternativ über die Systemsteuerung, Unterpunkt *Ordneroptionen*)). Je nach Betriebssystemversion kann es nun sein, dass Sie die notwendigen Einstellungen unter dem Untermenü *Ordner- und Suchoptionen ändern* oder *Ansicht* finden. Suchen und deaktivieren Sie die Option *Erweiterungen bei bekannten Dateitypen ausblenden* bzw. aktivieren Sie das Kontrollfeld *Dateinamenerweiterungen*.



Die zuletzt bearbeiteten Projekte können Sie über Hyperlinks unter dem Bereich *Zuletzt* auf der Startseite erneut öffnen. Alternativ stehen sie auch über den Menüpunkt *Datei - Zuletzt geöffnete Projekte und Projektmappen* zur Verfügung.

Verwalten Sie innerhalb einer Projektmappe mehrere Projekte, bietet Visual Studio die Möglichkeit, alle Projekte auf einmal zusammenzuklappen, sodass schneller eine übersichtlichere Darstellung entsteht. Verwenden Sie dazu  in der Symbolleiste des Projektmappen-Explorers.

3.4 Projekte in der Entwicklungsumgebung starten

Projekt ausführen

Zum Testen einer Anwendung, die sich im Entwicklungsstadium befindet, können Sie den sogenannten Debugger verwenden, der in Visual Studio integriert ist. Mit dem Debugger wird ein Projekt unter besonderen Bedingungen gestartet, die zur Fehlersuche und bedingt auch zum Testen geeignet sind. So lässt sich der Programmablauf beispielsweise unterbrechen und schrittweise fortsetzen. Die Stelle, an der der Programmablauf unterbrochen werden soll, den sogenannten Haltepunkt (vgl. Abschnitt 14.6), legen Sie fest.

- ▶ Drücken Sie zum Starten einer Anwendung im Debugger **F5**.
Alternativen:  oder *Debuggen - Debuggen starten*

Projekt ohne Debugger ausführen

Sie können eine Anwendung auch **ohne** Debugger starten. Die Anwendung wird dann wie außerhalb der Entwicklungsumgebung ausgeführt.

- ▶ Betätigen Sie **Strg F5**, um eine Anwendung ohne Debugger zu starten.
Alternative: *Debuggen - Starten ohne Debugging*

Die Projektausführung vorzeitig beenden

Eine Anwendung, die Sie im Debugger gestartet haben, können Sie vorzeitig abbrechen.

- ▶ Betätigen Sie  **F5**, um das gestartete Programm vorzeitig zu beenden.
Alternativen:  oder *Debuggen - Debugging beenden*

3.5 Mit Dateien im Projektmappen-Explorer arbeiten

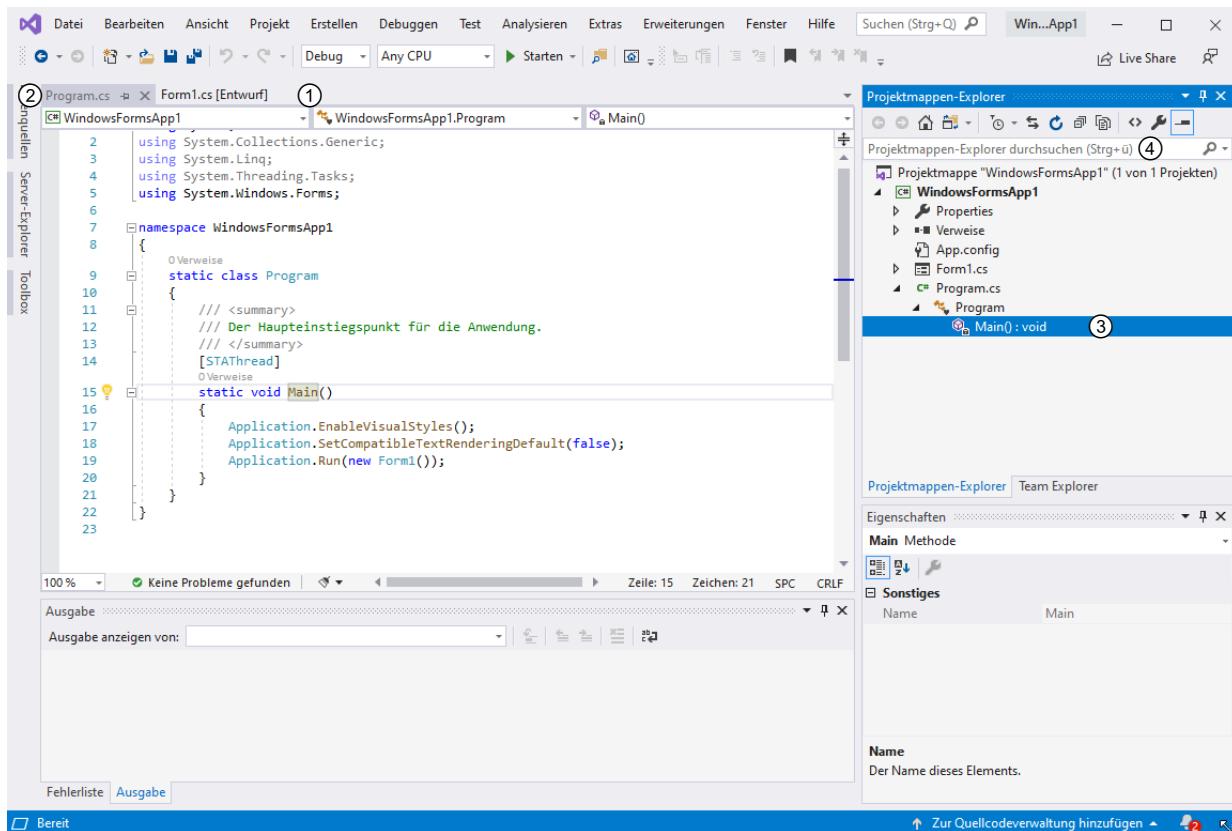
- ▶ Öffnen Sie eine Datei über den Projektmappen-Explorer durch einen Doppelklick auf den Dateinamen.
- ▶ Alternativ aktivieren Sie die Dateivorschau über  in der Titelleiste des Projektmappen-Explorers.
Wenn Sie eine Datei im Projektmappen-Explorer anklicken, wird eine Vorschau der Datei geöffnet. Dies erkennen Sie daran, dass für die Datei die Registerkarte rechts angeordnet wird ①. Es kann immer nur eine Datei in der Vorschau geöffnet werden. Editieren Sie die Datei in der Vorschau, wird diese zum Bearbeiten geöffnet.

Wenn Sie zahlreiche Dateien geöffnet haben, werden die Registerkarten der Dateien nicht mehr alle angezeigt. Möchten Sie sicherstellen, dass eine Registerkarte nicht wegescrollt wird, heften Sie diese über  an. Sie bleibt nun fest am linken Rand ② verankert.

- Visual Studio zeigt außerdem nicht nur die Dateinamen im Projektmappen-Explorer an, es wird im Falle von C#-Dateien auch die Dateistruktur angezeigt ③. Klicken Sie darin auf eine Methode, Variable oder Klasse, wird an die entsprechende Stelle der Datei navigiert (diese wird geöffnet, falls sie noch geschlossen ist).
- Über das Suchfeld ④ können Sie nach einem Programmerelement suchen, indem Sie dessen Namen eingeben.



Ergänzende Lerninhalte: *Entwicklungsumgebung.pdf*



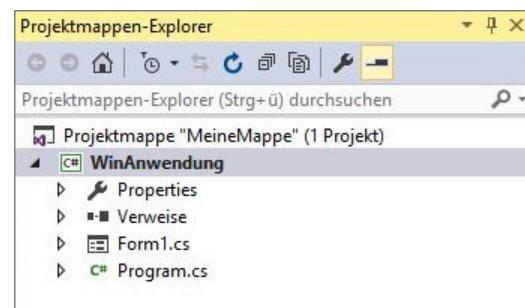
3.6 Übung

Mit Projektmappen und Projekten arbeiten

Übungsdatei: --

Ergebnisdatei: MeineMappe.sln

1. Erstellen Sie eine Windows Forms-Anwendung als neues Windows-Projekt und speichern Sie dabei das Projekt unter dem Namen *WinAnwendung* und die Projektmappe unter dem Namen *MeineMappe*.
2. Starten Sie die Programmausführung mit **F5**.
3. Beenden Sie die Programmausführung ordnungsgemäß, indem Sie das gestartete Programm (leeres Fenster) beenden.
4. In welchem Ordner werden Ihre Projekte standardmäßig gespeichert? (Dialogfenster *Optionen*)
5. Suchen Sie die zur Projektmappe gehörenden Dateien über den Windows-Explorer.



4 Anwendungen erstellen

4.1 Grundlagen der Anwendungserstellung

Anwendungsarten

Im Folgenden werden zwei Anwendungsarten des Projekttyps *Windows* vorgestellt: zum einen **Windows Forms-Anwendungen** mit grafischer Benutzeroberfläche und zum anderen **Konsolenanwendungen**, die keine grafische Benutzeroberfläche besitzen und bei denen Tastatureingaben und Textausgaben über die Konsole (unter Windows ist dies die Eingabeaufforderung) erfolgen.

Das .NET Framework und Visual Studio verfügen über zahlreiche weitere Anwendungstypen, die Sie erstellen können. Sie finden diese im Dialog, der nach Wahl des Menüpunkts *Datei - Neu - Projekt* geöffnet wird.

Ablauf für die Entwicklung einer Windows Forms-Anwendung

Für das Erstellen von Anwendungen dieses Typs mit grafischer Benutzeroberfläche werden **Formulare** verwendet. Ein Formular ist ein in der Regel rechteckiger Bildschirmbereich, der zur Kommunikation mit dem Benutzer verwendet wird. Die Formulare werden auch als **Windows Forms** bzw. kurz als **Forms** bezeichnet.

- ✓ **Projekt anlegen:**

Zunächst legen Sie ein neues Projekt mit der Vorlage *Windows Forms-App* bzw. *Windows Forms-App (.NET Framework)* an. Es enthält dann bereits ein einfaches Formular (Fenster).

Projekt anlegen:
Windows Forms-Anwendung



- ✓ **Formulare gestalten:**

Im Windows Forms-Designer gestalten Sie die Benutzeroberfläche, indem Sie **Steuerelemente** wie z. B. Buttons (Schaltflächen) oder Options- und Eingabefelder aus dem Werkzeugkasten (Toolbox) mithilfe der Maus auf der Formularoberfläche anordnen.

Formulare gestalten:
Steuerelemente einfügen und formatieren



- ✓ **Programmcode erstellen:**

Anschließend legen Sie fest, **auf welche Weise** Schaltflächen (Buttons) oder andere Steuerelemente **auf Ereignisse**, wie z. B. das Klicken mit der Maus, reagieren sollen. Beispielsweise soll eine Berechnung oder die Ausgabe von Ergebnissen erfolgen.

Programmcode erstellen:
Ereignisse (z. B. Klick auf eine Schaltfläche) auswerten und Funktionalität (z. B. Berechnungen) programmieren

Ablauf für die Entwicklung einer Konsolenanwendung

Bei einer Konsolenanwendung erfolgt die Bildschirmausgabe immer als Text. Die Verwendung der Maus wird nicht unterstützt.

- ✓ **Projekt anlegen:**

Zunächst legen Sie ein neues Projekt mit der Vorlage - Konsolen-App (.NET Core) oder Konsolen-App (.NET-Framework) - als Konsolenanwendung an.

Projekt anlegen:
Konsolenanwendung



- ✓ **Programmcode erstellen:**

Anschließend beginnen Sie sofort mit der Erstellung des Programmcodes (kurz **Code** genannt) und legen damit alle Berechnungen sowie Ein- und Ausgaben fest.

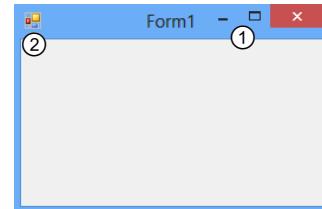
Programmcode erstellen:
Berechnungen, Ein- und Ausgabe etc.

4.2 Eine Windows-Anwendung erstellen

Eine neue Windows-Anwendung erzeugen

- Erzeugen Sie eine Windows-Anwendung als neues Projekt und benennen Sie sie.
Im Beispiel wird sowohl für das Projekt als auch für die Projektmappe der Name *SimpleWinApp* verwendet.

Die Anwendung enthält standardmäßig ein Formular, das im Windows Forms-Designer angezeigt wird. In dem Formular sind bereits die Schaltflächen ① für die Fenstersteuerung und das Systemmenü ② vorhanden. Durch das Platzieren von **Steuerelementen** (visuellen Komponenten, z. B. Eingabefeldern, Optionsfeldern, Schaltflächen etc.) auf dem Formular gestalten Sie die Benutzeroberfläche des Programms.

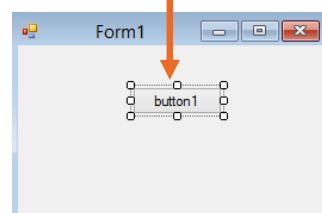
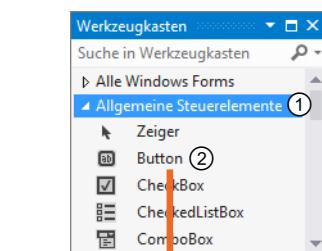


Formular mit Standardschaltflächen

Steuerelemente in ein Formular einfügen

Neue Steuerelemente fügen Sie mithilfe des Werkzeugkastens in ein Formular ein. Sie sind im Werkzeugkasten (Toolbox) auf verschiedene Bereiche aufgeteilt. Der Bereich **Allgemeine Steuerelemente** enthält beispielsweise häufig verwendete Steuerelemente. Zur besseren Übersicht können Sie die Inhalte einzelner Bereiche aus- bzw. einblenden.

- Öffnen Sie im Werkzeugkasten den gewünschten Bereich, beispielsweise **Allgemeine Steuerelemente** ①.
- Klicken Sie auf den Eintrag für das gewünschte Steuerelement, z. B. auf den Eintrag **Button** ②, und ziehen Sie die Komponente mit gedrückter linker Maustaste an die gewünschte Position im Formular.



Einen Button einfügen



Lassen Sie zum Einfügen neuer Komponenten den Werkzeugkasten permanent eingeblendet. Klicken Sie dazu in der Titelleiste des Werkzeugkasten-Fensters auf

Zur Verfügung stehende Komponenten

- ✓ Abhängig von der Art des Projekts bzw. des aktvierten Elements im Projekt werden unterschiedliche Komponenten bzw. Einträge im Werkzeugkasten aufgeführt.
- ✓ Neben den Steuerelementen, die im Formular als sichtbare Komponenten eingefügt werden können, existieren auch noch Komponenten, die nicht im Formular angezeigt werden und bei der Entwicklung unterhalb des Windows Form-Designers, im sogenannten Komponentenfach, erscheinen. Hierzu zählen beispielsweise die Komponente *Timer*, mit der ein Zeitgeber implementiert werden kann, und die Komponente *ImageList*, mit der mehrere Bilder für ein Projekt in einer Liste verwaltet werden können.
- ✓ Wenn Sie mit der Maus auf einen Eintrag im Werkzeugkasten zeigen, erhalten Sie als QuickInfo eine Kurzinformation über das Steuerelement.

Formulare und Steuerelemente beschriften

Das Formular und die meisten Steuerelemente besitzen nach der Erstellung eine durchnummerierte Standardbeschriftung, z. B. *Form1* oder *button1*. Um Ihre Benutzeroberfläche anwenderfreundlich zu gestalten, legen Sie für das Formular und die Steuerelemente individuelle Beschriftungen fest.

Die Beschriftung ist eine Eigenschaft des Formulars bzw. des Steuerelements und wird über das Eigenschaftenfenster geändert.

Eigenschaftenfenster öffnen

- Rufen Sie den Menüpunkt *Ansicht - Eigenschaftenfenster* auf.

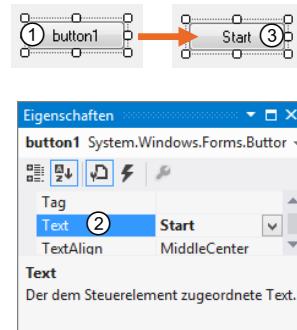
Alternativen:  im Projektmappen-Explorer (oder es muss manuell der Standard-Symbolleiste hinzugefügt werden) oder **Strg** **W** und dann **P**

Das Eigenschaftenfenster wird standardmäßig unterhalb des Projektmappen-Explorers angeordnet.

Beschriftung festlegen

- Markieren Sie das Formular bzw. das Steuerelement ①.
 - Klicken Sie im Eigenschaftenfenster auf die Eigenschaft *Text* ②.
 - Geben Sie die neue Beschriftung ein und bestätigen Sie die Eingabe mit .
- Die Beschriftung wird im Formular bzw. Steuerelement angezeigt ③.

Im Eigenschaftenfenster werden standardmäßig alle Eigenschaften alphabetisch angezeigt (Symbol ). Mit  können Sie die Eigenschaften nach Kategorien (Themenbereichen) gruppiert anzeigen. Die Eigenschaft *Text* finden Sie beispielsweise in der Kategorie *Darstellung*.



Steuerelemente eines Formulars benennen

Alle Steuerelemente eines Formulars besitzen einen **eindeutigen Namen**, mit dem das Element identifiziert und somit z. B. im Programmcode angesprochen werden kann. Dieser Name wird beim Einfügen der Komponente **automatisch** von Visual Studio vorgegeben. So wird beispielsweise eine Schaltfläche (ein Button) standardmäßig mit *button* und einer angehängten fortlaufenden Nummer benannt. Andere Steuerelemente werden entsprechend mit einem zum Typ passenden Namen und einer Nummer benannt.

Zur besseren Übersicht sollten Sie die automatisch generierten Namen immer durch aussagekräftige Namen ersetzen.

- Markieren Sie das Steuerelement und legen Sie im Eigenschaftenfenster über die Eigenschaft (*Name*) den Namen fest, beispielsweise *cmdStart* (command Start).
- ✓ Bei der Darstellung in Kategorien finden Sie die Eigenschaft (*Name*) in der Kategorie *Entwurf*.
- ✓ Die Eigenschaft (*Name*) ist in Klammern gesetzt. Das bedeutet, dass der Eintrag nicht leer sein darf.

Benennen Sie die Elemente **sofort nach der Erstellung**. Spätere Änderungen haben oft weitreichende Korrekturen im Programmcode zur Folge (Visual Studio unterstützt Sie allerdings dabei). Außerdem lassen sich selbst erklärende Namen besser lesen und man kann bereits erkennen, um welche Komponente es sich handelt und welche Aufgabe sie wahrnimmt.



Regeln für die Namensgebung

- ✓ Die Benennung muss **eindeutig** sein. In einem Formular dürfen nicht zwei Elemente denselben Namen erhalten.
- ✓ Die Benennung ist **case-sensitive**, d. h., Groß- und Kleinbuchstaben werden unterschieden.

Zur besseren Lesbarkeit sollten Sie eine einheitliche Schreibweise verwenden.



Die nachfolgenden Konventionen werden häufig für die Benennung von Formularelementen verwendet; sie sind aber nicht zwingend vorgeschrieben.

- ✓ Namen sollten sich nicht nur durch die Groß- und Kleinbuchstaben unterscheiden.
- ✓ Beginnen Sie bei mehreren Wörtern das erste Wort mit einem Kleinbuchstaben (wenn es sich um sogenannte private Elemente handelt) und jedes weitere Wort mit einem Großbuchstaben (Kamelschreibweise), beispielsweise `cmdStartCalculation`; Leerzeichen sind nicht erlaubt. Für öffentliche Elemente schreiben Sie auch den ersten Buchstaben des ersten Hauptwortes groß (Pascalschreibweise), z. B. `KundeAnlegen`.
- ✓ Häufig wird dem Namen eine Kennung vorangestellt, an der Sie den Typ des Formularelements erkennen können.
- ✓ Verwenden Sie beispielsweise `cmd`, um einen Button, oder `lbl`, um ein Label (Beschriftungstext) zu benennen.
- ✓ Ergänzen Sie diese Kennung durch einen beschreibenden Namen, wie z. B. `cmdStart`.

Steuerelemente in einem Formular löschen

- Markieren Sie das Steuerelement und betätigen Sie die `[Entf]`-Taste. Ggf. müssen Sie dazugehörige Ereignismethoden vorher entfernen (vgl. später in diesem Kapitel).

4.3 Mit Ereignissen den Ablauf steuern

Die ereignisgesteuerte Programmierung

Wenn Sie die zuvor erstellte Windows-Anwendung *SimpleWinApp* ausführen, wird das Fenster mit der eingefügten Schaltfläche *Start* angezeigt.

Klicken Sie auf die Schaltfläche, so erscheint diese gedrückt. Das Betätigen der Schaltfläche hat außer dieser Animation noch keine Auswirkung. Die Funktionalität der Schaltfläche, dass z. B. beim Klicken eine Textmeldung erscheinen soll, ist noch nicht programmiert.

Die Aktion eines Anwenders, beispielsweise das Klicken auf ein Steuerelement, löst ein Ereignis aus. Die Reaktion auf das Ereignis wird für die Komponente programmiert. Man spricht daher auch von **ereignisgesteuerter Programmierung**.

Beispiele für Ereignisse

Alle Aktionen (Tastatureingaben, Mausbewegungen) eines Benutzers sind Ereignisse, z. B.:

- ✓ Klicken oder Doppelklicken einer Schaltfläche (eines Buttons),
- ✓ Verschieben, Öffnen oder Schließen eines Fensters mit der Maus,
- ✓ Positionieren des Cursors in eine Textbox (ein Eingabefeld) mit der `[←]`-Taste,
- ✓ Tastatureingaben in eine Textbox.

Auf Ereignisse reagieren

Über Ereignisse können interne Programmabläufe ausgelöst werden, z. B.:

- ✓ das Durchführen von Berechnungen,
- ✓ das Öffnen und Schließen eines Fensters,
- ✓ die Aktualisierung der angezeigten Daten.

Ereignisse in Ereignismethoden behandeln

Die Reaktion, die beim Eintritt des Ereignisses erfolgen soll, wird in Form von Visual-C#-Anweisungen in einer speziellen Programmeinheit erstellt, die als **Ereignismethode** bezeichnet wird. Eine Ereignismethode enthält die Anweisungen, die ausgeführt werden, sobald ein mit dieser Methode verknüpftes Ereignis eintritt.

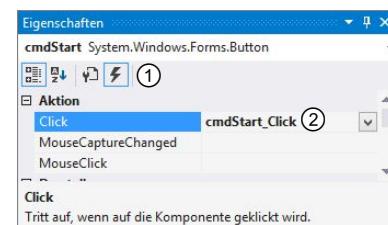
Ein Formular und die Steuerelemente (z. B. Button, Textbox, Checkbox) können auf bestimmte Ereignisse reagieren. Wenn eins dieser Ereignisse eintritt, wird die entsprechende Ereignismethode aufgerufen. Da die Reaktion auf ein Ereignis eine Eigenschaft der Komponente ist, erreichen Sie die Ereignisse über das Eigenschaftenfenster.

4.4 Ereignismethode festlegen

Ereignisse behandeln

- ▶ Markieren Sie durch Anklicken das Steuerelement, für das Sie ein Ereignis behandeln möchten (z. B. einen neuen Button *cmdStart*).
- ▶ Zeigen Sie das Eigenschaftenfenster an.
- ▶ Klicken Sie in der Symbolleiste des Eigenschaftenfensters auf ① um die Ereignisse aufzulisten, die in Bezug auf das markierte Steuerelement eintreten können.
- ▶ Klicken Sie doppelt auf das gewünschte Ereignis ②.
(Im Bereich ③ erhalten Sie eine kurze Erläuterung.)

Im Editor wird das Grundgerüst der Ereignismethode für das gewählte Ereignis eingefügt ④. Der Name der Ereignismethode setzt sich aus dem Namen des Steuerelements und dem Ereignis ⑤ zusammen.



```
public Form1()
{
    InitializeComponent();
}

private void cmdStart_Click(object sender, EventArgs ④)
{
    ⑤
}
```

Programmcode des Formulars
im Editor (Ausschnitt)

Das Standardereignis behandeln

Jedes Steuerelement besitzt ein sogenanntes **Standardereignis**. Dies ist das Ereignis, das mit dem Steuerelement erfahrungsgemäß am häufigsten verbunden ist. Beispielsweise ist das Standardereignis einer Schaltfläche (eines Buttons) das Anklicken (*Click*).

- ▶ Um eine Ereignismethode für das **Standardereignis** zu programmieren, klicken Sie im Windows Forms-Designer doppelt auf das entsprechende Steuerelement, im Beispiel auf die Schaltfläche *Start*.

Code für Ereignismethode

Innerhalb der Ereignismethode legen Sie nun fest, welche Reaktion auf das Ereignis erfolgen soll. In unserem Beispiel soll durch das Anklicken (Ereignis) der Schaltfläche *Start* (Ereignis *Click* bzw. Standardereignis eines Buttons) ein Hinweisfenster geöffnet werden. Das Hinweisfenster soll eine Textmeldung anzeigen und enthält standardmäßig zur Bestätigung der Meldung eine Schaltfläche *OK*.

- ▶ Geben Sie hinter der Programmzeile `private void cmdStart_Click(...` nach der geschweiften Klammer ① die folgende Programmzeile ein ①:
`MessageBox.Show("Die Schaltfläche wurde angeklickt.");`
Beachten Sie die Groß- und Kleinschreibung und das Semikolon am Ende der Programmzeile.

Während der Eingabe erscheinen automatisch mehrere Fenster der sogenannten IntelliSense, die hier noch nicht berücksichtigt werden sollen. Wie Sie die IntelliSense nutzen, erfahren Sie im folgenden Abschnitt.

<pre>... private void cmdStart_Click(object sender, EventArgs e) { ① MessageBox.Show("Die Schaltfläche wurde angeklickt."); }</pre>	<p><i>Das behandelte Ereignis Click</i></p>
---	---

Die eingegebene Zeile ① bewirkt, dass beim Anklicken der Schaltfläche (Button) *Start* ein Meldungsfenster (MessageBox) geöffnet wird, das den Text Die Schaltfläche wurde angeklickt. ausgibt.

Zwischen Programmcode-Editor und Windows Forms-Designer wechseln

Sie können übrigens schnell zwischen dem Editorfenster und dem Designerfenster eines Formulars umschalten, indem Sie im Kontextmenü des Formulars den Befehl *Code anzeigen* bzw. im Kontextmenü des Programmcodes den Befehl *Ansicht-Designer* wählen. Alternativ können Sie betätigen, um von der Code- zur Designer-Ansicht zu wechseln, und um von der Designer- in die Code-Ansicht zu wechseln.

Zusammenstellung einiger Maus- und Tastaturereignisse

Click	Ein Steuerelement wird angeklickt.
KeyDown	Eine Taste oder Tastenkombination wird betätigt.
KeyPress	Eine Taste mit einem bestimmten Zeichen oder ein Symbol wird betätigt.
KeyUp	Eine gedrückte Taste wird losgelassen.
MouseDown	Eine Maustaste wird betätigt.
MouseEnter	Der Mauszeiger erreicht das Steuerelement.
MouseHover	Der Mauszeiger befindet sich über dem betreffenden Steuerelement.
MouseLeave	Der Mauszeiger verlässt das Steuerelement.
MouseMove	Die Maus wird über dem Steuerelement bewegt.
MouseUp	Eine gedrückte Maustaste wird losgelassen.
Resize	Die Größe eines Steuerelements bzw. Formulars ändert sich.
TextChanged	Die Eigenschaft <i>Text</i> eines Steuerelementes hat sich geändert.

Jedes Steuerelement bzw. jede Komponente besitzt spezifische Ereignisse. In der Ereignisanansicht des Eigenschaftenfensters (Symbol) werden jeweils alle Ereignisse angezeigt, die für die jeweilige Komponente zur Verfügung stehen.

Sie haben versehentlich doppelt auf ein Steuerelement geklickt?

Es kann vorkommen, dass Sie mit einem Doppelklick auf ein Steuerelement im Formular versehentlich den Programmcode für die Ereignismethode des Standardereignisses erzeugen. Sie können die Ereignisbehandlung für das Steuerelement wieder aufheben.

Verknüpfung einer Behandlungsmethode mit einem Steuerelement aufheben

- ▶ Markieren Sie das Steuerelement und lassen Sie die Ereignisse im Eigenschaftenfenster anzeigen.
- ▶ Wählen Sie im Kontextmenü des Ereignisnamens (linke Spalte im Eigenschaftenfenster) den Befehl **Zurücksetzen**.
 - ✓ Wenn Sie den automatisch erstellten Programmcode der Behandlungsmethode noch nicht bearbeitet haben, wird dieser wieder entfernt.
 - ✓ Wenn Sie den Programmcode der Behandlungsmethode bereits bearbeitet haben, bleibt der Programmcode der Behandlungsmethode erhalten; das Steuerelement berücksichtigt diese Ereignismethode jedoch nicht mehr.

Eine Behandlungsmethode kann mehrfach für verschiedene Ereignisse (auch) verschiedener Steuerelemente genutzt werden, beispielsweise wenn derselbe Programmcode beim Anklicken eines Buttons und beim Aufruf eines Menüpunkts ausgeführt werden soll. Wählen Sie dazu im Listenfeld des Ereignisses die bereits vorhandene Methode aus.

Ereignismethode entfernen

Bevor Sie eine Behandlungsmethode vollständig entfernen, stellen Sie sicher, dass diese nicht mehr verwendet wird, und heben Sie – wie zuvor beschrieben – ggf. noch bestehende Verknüpfungen auf.

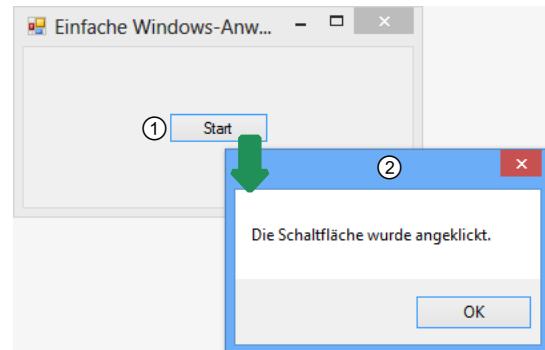


- ▶ Markieren Sie im Editor die gesamte Behandlungsmethode von der Zeile `private void ...` bis einschließlich der Zeile mit der schließenden geschweiften Klammer `}`.
- ▶ Betätigen Sie `[Entf]`, um den Programmcode zu löschen.

Anwendung ausführen und testen

Mithilfe von `[F5]` können Sie die Anwendung starten. Immer wenn Sie auf **Start** ① klicken, wird das Meldungsfenster ② angezeigt.

Im Verzeichnis Ihres Projekts finden Sie im Unterordner `bin\Debug` eine ausführbare Datei mit dem Namen Ihres Projekts und der Erweiterung `*.exe`. Diese Datei können Sie auch direkt (z. B. im Windows-Explorer) ausführen.



4.5 IntelliSense nutzen

Die IntelliSense

Die IntelliSense unterstützt Sie als Hilfsmittel bei der Erstellung eines möglichst fehlerfreien Programmcodes. Während der Eingabe stellt sie Ihnen die syntaktisch richtigen und sinnvollen Programmierwörter zur Auswahl. Sie wählen aus, und die IntelliSense fügt den Eintrag korrekt geschrieben und einheitlich formatiert in den Programmcode ein.

Vorteile der IntelliSense

- ✓ Reduzierung des einzutippenden Programmcodes
- ✓ Vermeidung von Tippfehlern
- ✓ Vermeidung von Programmierfehlern
- ✓ Anzeige der im Kontext verwendbaren Methoden, Eigenschaften etc.

Wörter vervollständigen lassen

Wenn Sie die Eingabe einer Programmzeile beginnen, genügt es meistens, nur den Anfang des entsprechenden Wortes einzutippen. Anschließend werden automatisch die passenden Programmierwörter zur Auswahl in einem Kontextmenü angezeigt. Visual Studio ergänzt das eingegebene Wort entsprechend, wenn Sie einen solchen Eintrag auswählen.

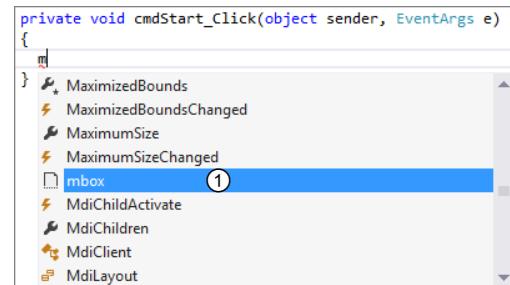
- Geben Sie den ersten Buchstaben des Programmierwortes ein, beispielsweise **M**, um *MessageBox* einzugeben. Das Kontextmenü wird automatisch aufgeklappt. Geben Sie weitere Buchstaben ein, wird die Auswahl angepasst.

oder

- Wird die Auswahlliste nicht angezeigt, betätigen Sie **Strg K** und dann **W**.

Alternativen: **A** in der Symbolleiste *Text-Editor* (muss manuell eingeblendet werden)
oder *Bearbeiten - Intellisense - Wort vervollständigen*

- ✓ Sofern der Wortanfang eindeutige Rückschlüsse darauf ziehen lässt, was Sie eingeben möchten, ergänzt Visual Studio das Wort automatisch.
- ✓ Falls mehrere Möglichkeiten bestehen, wie Sie die Eingabe fortsetzen können, wird die sogenannte **Memberliste** ① geöffnet, in der die möglichen Programmierwörter aufgelistet sind.
- ✓ In der Memberliste ist ein Programmierwort mit dem eingegebenen Wortanfang bereits markiert. Dies ist nicht zwangsläufig das erste, sondern ggf. das zuletzt verwendete Wort.



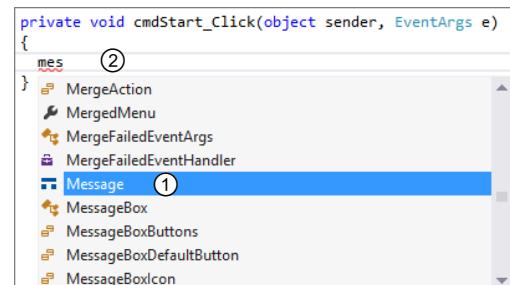
Programmierwort aus der Memberliste auswählen

Nachdem Visual Studio entsprechend dem eingegebenen Wortanfang bereits einen Eintrag in der Memberliste markiert hat, haben Sie verschiedene Möglichkeiten, den von Ihnen gewünschten Eintrag auszuwählen:

- Klicken Sie in der Memberliste auf das gewünschte Programmierwort ①, beispielsweise *Message*, das nun in den Sichtbereich gekommen ist.

oder Wählen Sie das Programmierwort mithilfe der Cursortasten **↓** und **↑**.

oder Geben Sie weitere Zeichen ein ②, um die Auswahl einzuschränken bzw. bis das gewünschte Wort in der Liste markiert ist.



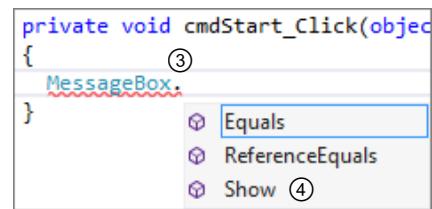
Programmierwort aus der Memberliste übernehmen

- Betätigen Sie die **Enter**- oder **Esc**-Taste oder klicken Sie doppelt auf den gewünschten Eintrag in der Memberliste, um das ausgewählte Programmierwort zu übernehmen.

oder Setzen Sie die Eingabe anschließend mit dem Zeichen fort, das auf das Programmierwort folgt, im Beispiel mit dem Punkt **.**

Das in der Memberliste markierte Wort wird ergänzt und das eingegebene nachfolgende Zeichen eingefügt ③.

Um wie im Beispiel die Programmzeile `MessageBox.Show ("Die Schaltfläche wurde angeklickt.");` einzugeben, bestätigen Sie nach der Auswahl des Wortes `MessageBox` mit . Wählen Sie dann ebenfalls über die Memberliste das Programmierwort `Show` ④ und geben Sie anschließend die Klammern, die Anführungszeichen, den darin eingeschlossenen Text und das Semikolon ein.



Hinweise zur Anzeige der Memberliste

- ✓ Falls die Memberliste nicht geöffnet ist, können Sie sie über das Symbol (muss manuell der Symbolleiste hinzugefügt werden) der Symbolleiste *Text-Editor*, über und dann oder über den Menüpunkt *Bearbeiten - Intellisense - Member auflisten* einblenden.
- ✓ Mit der -Taste können Sie die Memberliste jederzeit schließen.

Die Memberliste zur Bearbeitung von Programmcode nutzen

- ▶ Markieren Sie das Programmierwort, das Sie nachträglich ändern möchten.
- ▶ Rufen Sie den Menüpunkt *Bearbeiten - Intellisense - Member auflisten* auf.
Alternative:
- ▶ Klicken Sie doppelt auf den gewünschten Eintrag.
oder Markieren Sie den Eintrag und betätigen Sie die -Taste.

Programmcode generieren lassen

Unter dem Begriff „Generate From Usage“ (generiere bei Verwendung) oder auf Deutsch „Neuen Typ generieren ...“ verbirgt sich die automatische Erzeugung von Programmcode, der auf der Verwendung noch nicht existierender Klassen und Methoden beruht. Verwenden Sie beispielsweise eine Methode einer Klasse, die noch nicht existiert, wird am rechten Rand des Editors ein kleines Symbol mit einer Lampe (Smarttag-Anzeiger) angezeigt. Dieser Smarttag kann auch direkt im Code auftauchen, wenn Sie mit dem Mauszeiger über den Code fahren, der den noch nicht vorhandenen Code verwendet. Die Generierung kann für Klassen, Strukturen, Schnittstellen, Aufzählungen, Methoden, Eigenschaften und Felder erfolgen.

- ▶ Öffnen Sie das Smarttag durch Anklicken oder durch , während sich der Cursor auf dem betreffenden Bezeichner befindet.
- ▶ Lassen Sie sich über den Menüpunkt den Code generieren. Im Falle einer Klasse wird eine neue Datei angelegt und die Klasse darin erzeugt. Methoden werden in den betreffenden Klassen eingefügt.

4.6 Codeausschnitte einfügen

Codeausschnitte

Codeausschnitte, auch Snippets (dt. Schnipsel) genannt, sind vorgefertigte Programmcode-Fragmente, die Sie schnell in Ihr Dokument einfügen können. Codeausschnitte bestehen aus einem Grundgerüst für eine Programmstruktur, und einige beinhalten Platzhalter, über die Sie die individuellen Anpassungen vornehmen.

- ✓ Codeausschnitte reduzieren den Schreibaufwand.
- ✓ Codeausschnitte helfen, Syntaxfehler zu vermeiden, da sie das Grundgerüst der Programmstruktur vorgeben.

Die Erstellung neuer Codeausschnitte und die Verwaltung von Codeausschnitten ist nicht Bestandteil dieses Buchs.

Codeausschnitte einfügen

- ▶ Setzen Sie den Cursor im Editor an die Stelle, an der Sie den vorgefertigten Programmcode einfügen möchten.
- ▶ Rufen Sie den Menüpunkt *Bearbeiten - Intellisense - Ausschnitt einfügen* auf.
Alternative: **Strg K** und dann **X**
Die Liste der Codeausschnitte wird angezeigt. Existieren mehrere Kategorien, müssen Sie zuerst eine Kategorie, hier *Visual C#*, auswählen.
- ▶ Klicken Sie doppelt auf den gewünschten Codeausschnitt (z. B. *mbox*) ①.
Der Programmcode wird eingefügt ②.



Durch die Eingabe von Buchstaben können Sie wie in anderen IntelliSense-Listen die Auswahl einschränken.

```
private void cmdStart_Click(object sender, EventArgs e)
{
    Ausschnitt einfügen: Visual C# >
}
    iterator
    iterindex
    lock
    mbox ①
    namespaces
    stop
    propfull
    propog
    sim
```

Code vervollständigen

Viele Codeausschnitte enthalten Platzhalter, die im Editor farbig hinterlegt dargestellt werden. Der erste Platzhalter ist standardmäßig markiert ③.

- ▶ Tippen Sie den gewünschten Programmcode ein, um den markierten Platzhalter zu ersetzen.
 - ▶ Mit **Esc** springen Sie jeweils zum nächsten Platzhalter.
- oder* Klicken Sie auf einen Platzhalter, um ihn zu markieren.

```
private void cmdStart_Click(object sender, EventArgs e)
{
    ② MessageBox.Show("Test");
}
```

Platzhalter mit neuem Inhalt

Wenn Sie das Projekt speichern, schließen und erneut öffnen, sind die Platzhalter nicht mehr vorhanden. Der Inhalt der Platzhalter wird als Programmcode angezeigt.

Codeausschnitte schnell einfügen

Für jeden Codeausschnitt existiert eine sogenannte **Verknüpfung (Shortcut)**, die Ihnen beispielsweise in der QuickInfo ① bei der Auswahl eines Codeausschnittes angezeigt wird.

mbox ①
Codeausschnitt für MessageBox.Show
QuickInfo eines Codeausschnitts

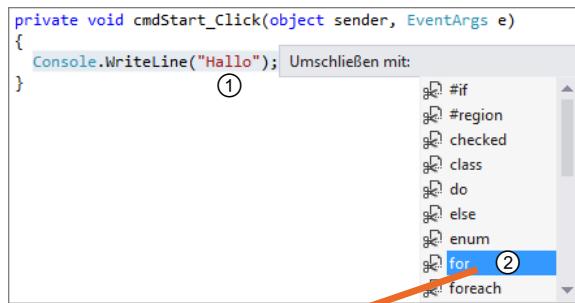
- ▶ Geben Sie den Namen der Verknüpfung ein (beispielsweise *mbox*).
- ▶ Betätigen Sie **Enter**, um über die Intellisense den Namen der Verknüpfung im Quellcode einzutragen.
- ▶ Drücken Sie **Enter** noch einmal, um die Verknüpfung (den Shortcut) auszuführen und durch den entsprechenden Codeausschnitt zu ersetzen.

Umschließende Codeausschnitte einsetzen

Bestehende Programmzeilen lassen sich in vorgefertigte Programmstrukturen einschließen.

- ▶ Markieren Sie die gewünschten Programmzeilen ①.
- ▶ Rufen Sie den Menüpunkt *Bearbeiten - Intellisense - Umschließen mit:* auf.
Alternative: **Strg K** und dann **S**
- Die Liste der Codeausschnitte wird angezeigt.
- ▶ Klicken Sie doppelt auf den gewünschten Codeausschnitt (z. B. *for*) ②.
- Der Programmcode wird eingefügt ③.

Auch einige dieser Codeausschnitte enthalten Platzhalter ④, die Ihnen die weitere Bearbeitung des Programmcodes erleichtern.



```
private void cmdStart_Click(object sender, EventArgs e)
{
    Console.WriteLine("Hallo");
}
```

③ for (int i = 0; i < length; i++)
{
 Console.WriteLine("Hallo");
}

Programmcode umschließen lassen

4.7 Konsolenanwendungen erstellen

Eine neue Konsolenanwendung erzeugen

Sie können mit C# Anwendungen erstellen, die nicht mit einer grafischen Benutzeroberfläche ausgestattet sind, aber Ein- und Ausgaben in Textform zulassen. Diese Ein- und Ausgaben erfolgen über ein Konsolenfenster (unter Windows entspricht dies der Eingabeaufforderung). Nach Auswahl des Projekttyps für C# und Windows wird Ihnen die entsprechende Projektvorlage mit dem Namen *KonsolenApp* (entweder für .NET Core oder .NET-Framework) bereitgestellt. Weil Konsolenanwendungen auf alle grafikorientierten Merkmale verzichten, sind sie besonders schlank.

- ▶ Erzeugen Sie eine neue Konsolenanwendung und benennen Sie sie.
Im Beispiel wird der Name *HalloWelt* verwendet.

Ein Editorfenster wird als Dokumentfenster geöffnet. Das Grundgerüst einer Konsolenanwendung wird automatisch eingefügt.

```
...
namespace HalloWelt
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Programmcode für eine Konsolenanwendung erstellen

Da keine grafischen Elemente zu positionieren sind, beschränkt sich die Erstellung einer Konsolenanwendung auf die Codierung des Programmcodes im Editor.

Um beispielsweise eine Anwendung zu erstellen, die den Text *Hello Welt!* ausgibt, setzen Sie den Cursor hinter die öffnende geschweifte Klammer ① in der Zeile ①. Betätigen Sie die **←**-Taste, um eine neue Zeile einzufügen, und geben Sie die folgende Programmzeile ein:

```
Console.WriteLine("Hallo Welt!");
```

Sie können sich bei der Eingabe von der IntelliSense unterstützen lassen oder die Textausgabe schnell mit dem Codeausschnitt cw (`Console.WriteLine`) erstellen:

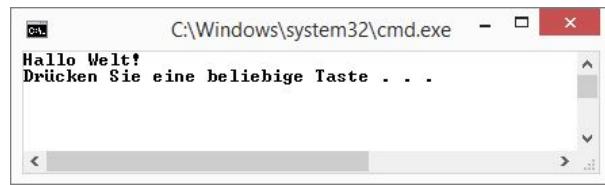
- ▶ Tippen Sie cw ein und drücken Sie zweimal .
- ▶ Geben Sie innerhalb des Klammerpaars   ① den Text "Hallo Welt!" ein.

```
static void Main(string[] args)
{
    Console.WriteLine(); ①
}
```

Eine Konsolenanwendung ausführen

- ▶ Betätigen Sie **Strg F5**, um die Anwendung zu starten.

Es erscheint ein Konsolenfenster mit der Ausgabe von *Hello Welt!* und dem Hinweis, dass Sie mit einer beliebigen Taste das Fenster (nicht die Anwendung) schließen können.



Ohne Debugger gestartete Konsolenanwendung
"HelloWelt.sln"

Wenn Sie die Anwendung im sogenannten Debug-Modus, d. h. mit **F5** bzw. über den Menüpunkt *Debuggen -> Debugging starten*, ausführen, wird die Anwendung nach der Bildschirmausgabe sofort beendet. Mögliche Ausgaben der Anwendung können Sie in diesem Fall nicht mehr überprüfen. Verwenden Sie ggf. einen Aufruf `Console.ReadKey()`; am Ende Ihrer Anwendung, der abschließend auf die Eingabe eines beliebigen Zeichens wartet.

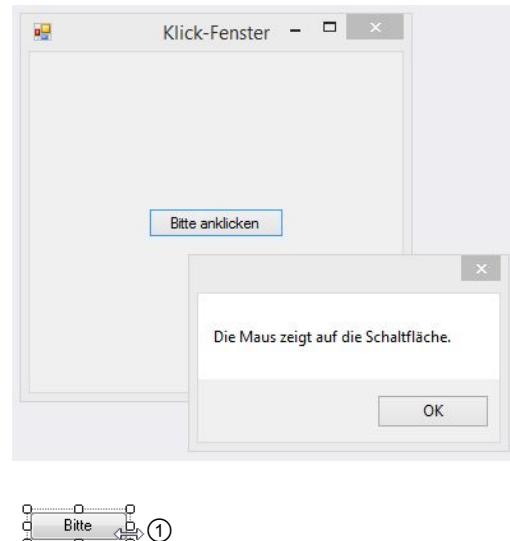
4.8 Übung

Eine Schaltfläche, die sich nicht anklicken lässt

Übungsdatei: --

Ergebnisdatei: TryToClick.sln

1. Erzeugen Sie ein neues Projekt als Windows-Anwendung mit dem Namen *Klick*.
2. Speichern Sie die Projektmappe unter dem Namen *TryToClick*; das Projekt soll weiterhin *Klick* heißen.
3. Geben Sie für die Titelleiste des Formulars den Text *Klick-Fenster* ein.
4. Platzieren Sie in dem Formular einen Button (eine Schaltfläche) mit dem Namen *cmdKlick* und beschriften Sie ihn mit dem Text *Bitte anklicken*.
Tipp: Falls der Text nicht vollständig angezeigt wird, markieren Sie den Button und ziehen Sie den Anfasser ① nach rechts, um den Button zu verbreitern.
5. Erstellen Sie für den Button eine Ereignisbehandlung, die auf das Ereignis *MouseEnter* reagiert. Es soll ein Hinweisfenster (MessageBox) erscheinen, das den Text „Die Maus zeigt auf die Schaltfläche.“ enthält.
6. Testen Sie die Anwendung.



Anmerkung: Sie können die Schaltfläche **nicht** anklicken. Sobald Sie die Maus auf die Schaltfläche bewegen, erscheint das Hinweisfenster.

5 Benutzeroberflächen gestalten

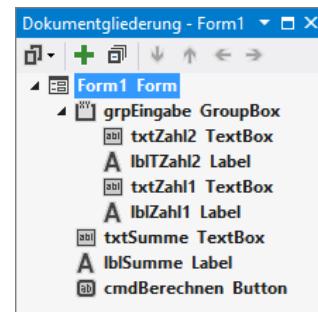
5.1 Grundlegende Bearbeitung

Formularstruktur anzeigen

Mit dem Fenster *Dokumentgliederung* können Sie die Struktur eines Formulars einsehen.

- Markieren Sie ein Formular und rufen Sie den Menüpunkt *Ansicht - Weitere Fenster - Dokumentgliederung* auf.

In der obersten Ebene befindet sich das Formular (hier: Form1). Darunter werden die Steuerelemente des Formulars aufgelistet. Das Fenster zeigt nicht nur den Aufbau des Formulars, sondern es kann auch zum Bearbeiten der Struktur verwendet werden.

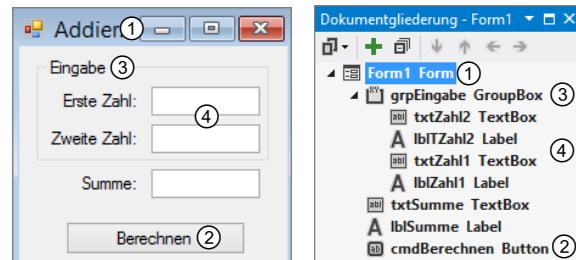


Steuerelemente in Gruppen zusammenfassen

Visual C# stellt eine Vielzahl von Steuerelementen zur Verfügung, mit denen Sie Ihre Benutzeroberflächen gestalten können. Einige Steuerelemente dienen als Container, d. h., sie können wiederum Steuerelemente aufnehmen. Ein solches Containerelement ist beispielsweise das Steuerelement *GroupBox* ③. Containerelemente finden Sie im Werkzeugkasten im Bereich *Container*.

Durch die Verwendung dieser Steuerelemente entsteht eine hierarchische Struktur des Formulars, die in der Dokumentgliederung dargestellt wird.

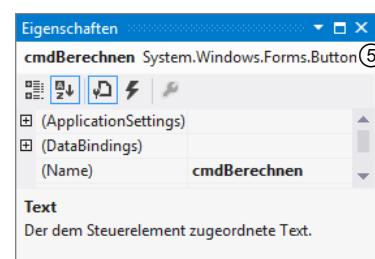
Innerhalb des Formulars ① befinden sich Steuerelemente ② oder Containerelemente ③, die wiederum Steuer- elemente ④ oder ebenfalls Containerelemente enthalten können.



Formular und Dokumentgliederung

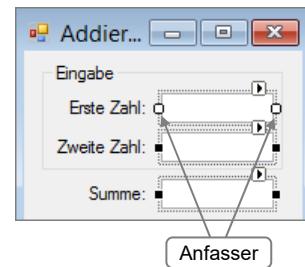
Steuerelemente markieren

- Klicken Sie auf ein Steuerelement, um es zu markieren.
- oder* Klicken Sie auf den Namen des Steuerelements im Fenster *Dokumentgliederung*.
- oder* Wählen Sie das Element über das Listenfeld ⑤ im Eigenschaftenfenster.



Mehrere Steuerelemente markieren

- ▶ Markieren Sie mehrere Steuerelemente, indem Sie diese bei gedrückter **Strg**-Taste nacheinander anklicken.
 - ▶ Das zuerst markierte Steuerelement erhält weiße, alle anderen schwarze **Anfasser**.
- oder* Ziehen Sie im Formular einen Rahmen auf, der die gewünschten Steuerelemente zumindest teilweise enthält.



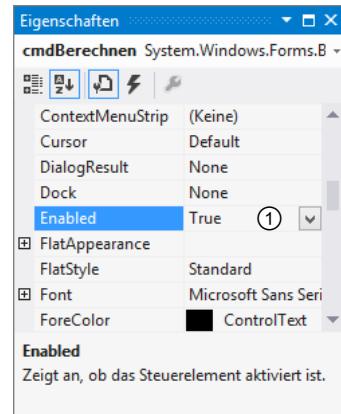
Formular markieren

- ▶ Klicken Sie in einen freien Bereich des Formulars, um das Formular selbst zu markieren.
- oder* Klicken Sie im Fenster *Dokumentgliederung* auf den Formularnamen.

Eigenschaften festlegen bzw. ändern

Die Eigenschaften eines Steuerelements bzw. Formulars können Sie über das Eigenschaftenfenster bearbeiten.

- ▶ Markieren Sie das Steuerelement bzw. Formular.
 - ▶ Klicken Sie in der Liste der Eigenschaften des ausgewählten Objekts auf den zu ändernden Wert ①.
- Je nach Eigenschaft können Sie ...
- ✓ den vorgegebenen Wert direkt überschreiben,
 - ✓ durch Klicken auf den Pfeil ▾ ein Listenfeld zur Auswahl eines Wertes öffnen,
 - ✓ durch Klicken auf die Schaltfläche [...] ein Dialogfenster zum Erstellen des Wertes anzeigen.



Bei Eigenschaften, zu denen eine Auswahlliste (▾) zur Verfügung steht, wechseln Sie schnell zwischen den Auswahlmöglichkeiten, indem Sie doppelt auf den Wert der Eigenschaft klicken.

Steuerelemente schnell umbenennen

Sie können ein Steuerelement schnell umbenennen, indem Sie in der Dokumentgliederung einmal auf den Namen eines markierten Steuerelements klicken.

Übersicht ausgewählter Eigenschaften

BackColor	Hintergrundfarbe des Steuerelements bzw. Formulars
BorderStyle	Art der Objektumrahmung
Enabled	Aktiviert oder deaktiviert das Steuerelement, um beispielsweise auf Ereignisse zu reagieren bzw. nicht zu reagieren
Font	Textformatierung wie Schriftart, Schriftschnitt, Schriftgröße etc.
FormBorderStyle	Darstellungsart des Fensters, beispielsweise als Toolfenster, als Dialogfenster mit fester Größe oder als skalierbares Fenster
ForeColor	Schriftfarbe des Textes
Location	Koordinaten der linken oberen Ecke des Steuerelements bzw. Formulars

Locked	Legt fest, ob ein Steuerelement zur Entwurfszeit auf dem Formular verändert werden kann
(Name)	Name bzw. interne Bezeichnung des Objekts
Size	Breite (Width) und Höhe (Height) des Steuerelements bzw. Formulars
TabStop	Legt fest, ob das Steuerelement mit der -Taste angesprungen werden kann
TabIndex	Die Steuerelemente können in der von diesem Index festgelegten Reihenfolge mit der -Taste angesprungen werden.
Text	Text, Überschrift oder Beschriftung des Steuerelements bzw. Titel des Formulars
TextAlign	Legt die Textausrichtung fest
Visible	Legt fest, ob das Steuerelement während des Programmlaufs sichtbar ist

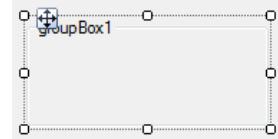
Größe eines Steuerelements oder eines Formulars ändern

- ▶ Markieren Sie das Steuerelement bzw. Formular.
- ▶ Legen Sie durch Ziehen des entsprechenden Anfassers die Größe fest.
oder Halten Sie die -Taste gedrückt und stellen Sie mit den Tasten , , und die Größe ein.

Steuerelemente verschieben

- ▶ Markieren Sie das gewünschte Steuerelement.
- ▶ Klicken Sie in das Steuerelement und ziehen Sie es mit gedrückter Maustaste an die gewünschte Position.
oder Positionieren Sie das Steuerelement mit den Tasten , , und .

Steuerelemente, die als Container dienen, besitzen ein Verschiebesymbol . Um solche Steuerelemente zu verschieben, klicken Sie auf das Verschiebesymbol und ziehen Sie das Steuerelement mit gedrückter Maustaste an die gewünschte Position.

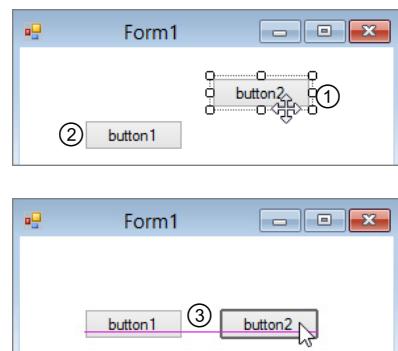


5.2 Positionierhilfen nutzen

Steuerelemente mit Ausrichtungslinien positionieren

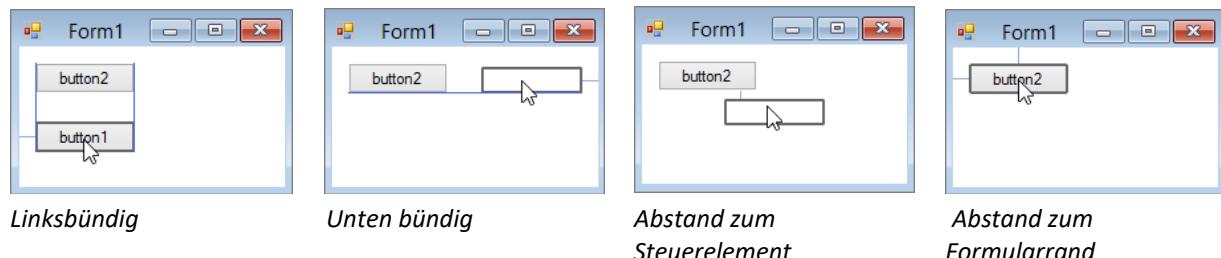
Die Entwicklungsumgebung bietet Ihnen Hilfslinien, sogenannte **Ausrichtungslinien** (SnapLines), mit deren Unterstützung Sie auf einem Formular Steuerelemente schnell zueinander ausrichten können.

- ▶ Klicken Sie auf das Steuerelement, das Sie positionieren möchten ①, und verschieben Sie es in Richtung des gewünschten Bezugselements (hier ②).
Sobald sich das Steuerelement **ungefähr** auf gleicher Höhe mit einem Bezugselement befindet, wird eine violette bzw. blaue Hilfslinie, eine sogenannte **Ausrichtungslinie** ③, eingeblendet. Das zu verschiebende Steuerelement wird **exakt** an der Hilfslinie ausgerichtet.
- ▶ Lassen Sie die Maustaste los, um die Position zu übernehmen.



Es stehen verschiedene Hilfslinien zur Verfügung, mit denen Sie das Steuerelement sowohl vertikal als auch horizontal an anderen Steuerelementen ausrichten können. Auch einen voreingestellten **Abstand** zu anderen Steuerelementen und zum Formularrand können Sie mithilfe von Hilfslinien einhalten.

Die violette Linie ③ bezieht sich auf den **Inhalt**, beispielsweise auf den Beschriftungstext. Blaue Linien beziehen sich auf den **Rand** des Steuerelements. Die nachfolgenden Abbildungen zeigen einige Beispiele:

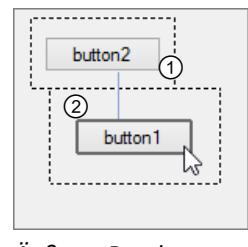


Schnell verschieben Sie das markierte Steuerelement zur nächsten Ausrichtungslinie, indem Sie die **[Strg]**-Taste gedrückt halten und es mit den Pfeiltasten (**←**, **↑**, **↓**, **→**) in die gewünschte Richtung bewegen.

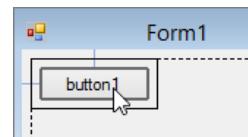
Einstellungen für die Ausrichtungslinien vornehmen

Ausrichtungslinien, die den Abstand zu einem anderen Steuerelement vorgeben, richten sich nach der Randbreite, die für die Steuerelemente eingestellt ist. Zwei Randarten werden unterschieden:

- ✓ Der **äußere Rand** wird über die Eigenschaft *Margin* festgelegt und beschreibt den Abstand zu einem benachbarten Steuerelement. Die Ausrichtungslinie berücksichtigt den Rand beider Steuerelemente. Die Länge der Ausrichtungslinie ergibt sich aus der Summe der Randbreiten beider Steuerelemente (① und ②).
- ✓ Der **innere Rand** wird über die Eigenschaft *Padding* eingestellt und gibt den Abstand zum Inhalt des Steuerelements vor. Beispielsweise möchten Sie Steuerelemente innerhalb eines Containerelements oder innerhalb des Formulars anordnen. Die Länge der Ausrichtungslinie ergibt sich aus der Summe der Breite des inneren Randes (Containerelement) und der äußeren Randbreite des enthaltenen Steuerelements.



Äußerer Rand



Innerer Rand

Standardmäßig sind für jedes Steuerelement Randbreiten bzw. äußere Abstände (engl. margin) vorgegeben. Sie können diese aber individuell anpassen.

- Blenden Sie für das gewünschte Steuerelement im Eigenschaftenfenster die Eigenschaft *Margin* ein.
 - Ändern Sie den Eintrag *All*, um alle Ränder einheitlich auf denselben Wert einzustellen.
- oder* Legen Sie die Werte für den linken, oberen, rechten und unteren Rand individuell fest.

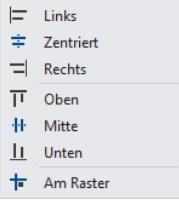
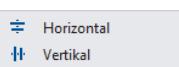
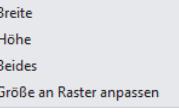
Margin	20; 20; 20; 20
All	20
Left	20
Top	20
Right	20
Bottom	20

Im abgebildeten Beispiel auf der vorherigen Seite besitzt der Button *button1* einen einheitlichen äußeren Rand von 10 pt ①. Für den Button *button2* ist ein äußerer Rand von 20 pt eingestellt ②. Die Ausrichtungslinie hat daher einen Abstand von 30 pt.

Mehrere Steuerelemente anpassen und zueinander ausrichten

- Markieren Sie die Steuerelemente, die Sie ausrichten möchten.
- Wählen Sie eine der folgenden Aktionen, um die Steuerelemente auf dem Formular auszurichten.

Sie können dazu den angegebenen Menüpunkt oder das entsprechende Symbol der Symbolleiste *Layout* verwenden.

Sie möchten markierte Objekte ...		
zueinander ausrichten	► Rufen Sie den Menüpunkt <i>Format - Ausrichten</i> auf und wählen Sie den gewünschten Untermenüpunkt.	 <ul style="list-style-type: none"> Links Zentriert Rechts  <ul style="list-style-type: none"> Oben Mitte Unten  <ul style="list-style-type: none"> Am Raster
im Formular horizontal oder vertikal zentrieren	► Rufen Sie den Menüpunkt <i>Format - Auf Formular zentrieren</i> auf und wählen Sie den gewünschten Untermenüpunkt <i>Horizontal</i> bzw. <i>Vertikal</i> .	 <ul style="list-style-type: none"> Horizontal Vertikal
in der Größe anpassen	► Rufen Sie den Menüpunkt <i>Format - Größe ausgleichen</i> auf und wählen Sie den gewünschten Untermenüpunkt.	 <ul style="list-style-type: none"> Breite Höhe Beides Größe an Raster anpassen
horizontal oder vertikal anpassen	► Rufen Sie den Menüpunkt <i>Format - Horizontaler Abstand</i> bzw. <i>Vertikaler Abstand</i> auf und wählen Sie den gewünschten Untermenüpunkt.	 <ul style="list-style-type: none"> Angleichen Vergroßern Verkleinern Entfernen

Wenn mehrere Steuerelemente zueinander ausgerichtet oder aneinander angepasst werden, erfolgt die Orientierung immer an dem **zuerst** markierten Steuerelement.



Steuerelemente an einem Raster ausrichten

Standardmäßig können Sie beim Verschieben mit der Maus die Steuerelemente mithilfe der Ausrichtungslinien oder, sofern keine Ausrichtungslinie eingeblendet ist, beliebig im Formular platzieren.

Die Verwendung eines Rasters aktivieren

- Rufen Sie den Menüpunkt *Extras - Optionen* auf.
- Wählen Sie im Bereich *Windows Forms-Designer - Allgemein* als *Layoutmodus* den Eintrag *SnapToGrid* und setzen Sie die Eigenschaft *Am Raster ausrichten* auf den Wert *True*.
- Legen Sie die Größe des Rasters mit der Eigenschaft *Rasterzellen-Standardgröße* fest.
- Bestätigen Sie mit *OK*, schließen Sie den Windows Forms-Designer und öffnen Sie ihn erneut, z. B. im Projektmappen-Explorer mit einem Doppelklick auf den Formularnamen.

Die Einstellungen werden erst nach einem Neustart des Windows Forms-Designers wirksam. Die Ausrichtungslinien werden nicht mehr eingeblendet; stattdessen lassen sich die Steuerelemente an dem Raster ausrichten. Auch bei der Größeneinstellung eines Steuerelements wird das Raster verwendet.

Steuerelemente frei positionieren

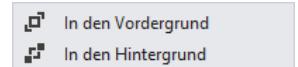
- ✓ Um ein Steuerelement unabhängig von Ausrichtungslinien bzw. einem Raster völlig frei zu positionieren, verschieben Sie das Steuerelement mit den Tasten *←*, *↑*, *↓* und *→*.
- ✓ Bei aktiviertem Raster halten Sie zusätzlich die *Strg*-Taste gedrückt, um die Steuerelemente frei zu positionieren.
- ✓ Über die Eigenschaft *Location* können Sie im Eigenschaftenfenster die horizontale Position X und die vertikale Position Y bei allen grafischen Steuerelementen direkt als Zahlenwert eingeben.

5.3 Weitere Möglichkeiten

Steuerelemente in den Hintergrund oder Vordergrund setzen

Einzelne Steuerelemente können sich überdecken. In diesen Fällen legen Sie fest, welche Steuerelemente im Vordergrund und welche im Hintergrund liegen, d. h. überdeckt werden.

- ▶ Markieren Sie das entsprechende Steuerelement.
- ▶ Wählen Sie den Menüpunkt *Format - Reihenfolge - In den Vordergrund* bzw. *In den Hintergrund*.
- ▶ *In den Hintergrund*.
Alternative:  bzw. .



Über das Fenster *Dokumentgliederung* haben Sie die Möglichkeit, ein Steuerelement schrittweise jeweils eine Ebene weiter in den Hintergrund (in der Gliederung nach oben) oder den Vordergrund (in der Gliederung nach unten) zu stellen. Markieren Sie dazu in der Dokumentgliederung das gewünschte Objekt und stellen Sie es mit dem Symbol  eine Ebene weiter in den Vordergrund bzw. mit  eine Ebene weiter in den Hintergrund.

Steuerelemente sperren

Um zu verhindern, dass Steuerelemente versehentlich durch Ziehen mit der Maus verschoben oder in der Größe verändert werden, können Sie sie sperren. Sie sind dann an ihrer aktuellen Position auf dem Formular fixiert.

- ▶ Rufen Sie den Menüpunkt *Format - Steuerelemente anpassen* auf.
- ✓ Die Sperrung betrifft alle bisher auf dem Formular platzierten Steuerelemente und das Formular selbst.
- ✓ Wenn Sie anschließend weitere Steuerelemente hinzufügen, sind diese nicht gesperrt.
- ✓ Gesperrte Steuerelemente besitzen nur einen Anfasser.

Um die Sperrung wieder aufzuheben, rufen Sie den Menüpunkt erneut auf.

Einzelne Steuerelemente sperren bzw. entsperren

Zum Sperren bzw. Entsperren eines einzelnen Steuerelements setzen Sie dessen Eigenschaft `Locked` auf den Wert `True` (gesperrt) bzw. `False` (entsperrt).

Aktivierungsreihenfolge festlegen

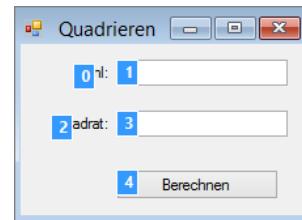
Bei Windows-Anwendungen können Sie in einem Fenster, z. B. einem Dialogfenster, mithilfe der -Taste jeweils zum nächsten Fensterelement gelangen. Die entsprechende Reihenfolge ist im Formular festgelegt.

Labels werden beispielsweise mit der -Taste nicht angesprungen. Grundsätzlich sind mit der -Taste nur die Steuerelemente anwählbar, die ein Benutzer auch sonst per Maus und Tastatur nutzen kann.

Aktuelle Reihenfolge anzeigen

- ▶ Markieren Sie das Formular.
- ▶ Rufen Sie den Menüpunkt *Ansicht - Aktivierungsreihenfolge* auf.
Alternative:  (muss manuell der Symbolleiste hinzugefügt werden)
Im Formular wird mit Nummern (`TabIndex`) die aktuelle Reihenfolge angezeigt.

Der Index wird in der Eigenschaft `TabIndex` des jeweiligen Steuerelements gespeichert.



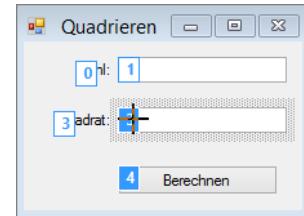
Quadrieren.sln

Bei der Erstellung des Formulars werden die Indexwerte automatisch fortlaufend vergeben. In diesem Beispiel wurde das Steuerelement mit dem Index 4 gelöscht. Der Index 4 fehlt daher.

Neue Reihenfolge festlegen

- ▶ Klicken Sie nacheinander in der gewünschten Reihenfolge auf alle Steuerelemente des Formulars.
Die Nummern der bereits angeklickten Elemente erscheinen auf weißem Hintergrund.

Um die Anzeige der Tabulatorreihenfolge wieder auszublenden, rufen Sie den Menüpunkt *Ansicht - Aktivierungsreihenfolge* erneut auf oder klicken Sie erneut auf das Symbol .



5.4 Projekte mit mehreren Formularen

Wenn Sie eine umfangreiche Windows-Anwendung erstellen, wird diese üblicherweise mehrere Fenster (Formulare) enthalten. Beispielsweise existiert ein Hauptanwendungsfenster; für Einstellungen und die Entgegennahme von Daten benötigen Sie Dialogfenster.

Weitere Formulare erstellen

- ▶ Rufen Sie den Menüpunkt *Projekt - Formular hinzufügen (Windows Form)* ... auf, um dem aktuellen Projekt ein neues Formular hinzuzufügen.
Alternative: Pfeilschaltfläche des Symbols  (muss manuell der Symbolleiste hinzugefügt werden) links auf der Standard-Symbolleiste, *Windows Form hinzufügen*

Startformular festlegen bzw. ändern

Damit bei der Ausführung das gewünschte Formular (das Hauptanwendungsfenster) geöffnet wird, muss dieses vom Hauptprogramm aufgerufen werden. Das Hauptprogramm wird beim Starten der Anwendung ausgeführt.

- ▶ Klicken Sie im Projektmappen-Explorer doppelt auf die Programmdatei *Program.cs*.
Diese Datei enthält das Hauptprogramm *Main()*. Die Zeile ① erzeugt das Formular (hier *Form1*) und öffnet es.
- ▶ Ersetzen Sie den Namen *Form1* durch den Namen des Formulars, das Sie als Startformular verwenden möchten.

```
...
static class Program
{
    ...
    static void Main()
    {
        ...
        ① Application.Run(new Form1());
    }
    ...
}
```

Den Namen der Formulardatei und des Formulars ändern

Wenn Sie mit mehreren Formularen arbeiten, sollten Sie vor allem bei großen Projekten die standardmäßig vorgegebenen Namen der Formulare durch aussagekräftige Namen ersetzen.

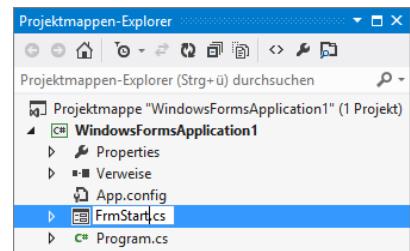
Formularnamen halten sich an die Konventionen für Steuerelemente. Formularnamen beginnen häufig mit *Frm* und heißen beispielsweise *FrmStart*. Standarddialogfenster beginnen oft mit *Dlg* (für Dialog) und heißen z. B. *DlgFileSave*. Die konkrete Benennung bleibt jedoch letztendlich Ihnen überlassen.

Jedes Formular wird in mehreren separaten, zusammengehörigen Dateien, z. B. *Form1.cs*, *Form1.Designer.cs* und *Form1.resx* (*Logik*, *Oberfläche*, *Ressourcen*), gespeichert. Der Basissname dieser Formulardatei wird Ihnen im Projektmappen-Explorer angezeigt. Der Formularname ist in der Eigenschaft *Name* gespeichert.

Den Namen der Formulardatei ändern

Standardmäßig sind der Name der Formulardatei und der Formularname nach der Erstellung des Formulars gleich. Wenn Sie den Namen der Formulardatei ändern, wird der Name des Formulars automatisch angepasst.

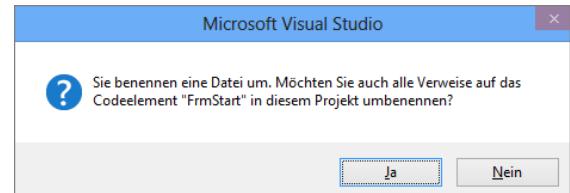
- ▶ Klicken Sie mit der rechten Maustaste auf den Namen des Formulars (z. B. *Form1.cs*) im Projektmappen-Explorer und wählen Sie den Kontextmenüpunkt *Umbenennen*.
Der Name wird markiert und es erscheint ein Textcursor.
- ▶ Ändern Sie den Namen und betätigen Sie .
- Beachten Sie, dass die Erweiterung *.cs* bestehen bleiben muss, anderenfalls kann das Formular nicht mehr verwendet werden.



Formulardatei umbenennen

Anschließend erfolgt eine Rückfrage, ob der Name überall im Programmcode des Projekts entsprechend angepasst werden soll.

- ▶ Bestätigen Sie die Rückfrage mit *Ja*.
- ✓ Wenn Sie den **Formularnamen** über die Eigenschaft *Name* ändern, wird der entsprechende Dateiname **nicht automatisch** angepasst.
- ✓ Sind der Name der Formulardatei und der Formularname verschieden, erfolgt ebenfalls keine automatische Anpassung.



Prüfen Sie anschließend, ob im Hauptprogramm der Name des zu startenden Formulars entsprechend angepasst wurde.

5.5 Übung

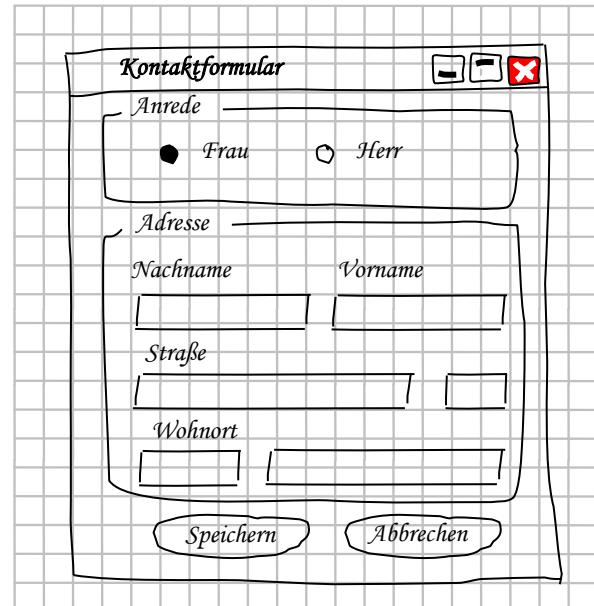
Ein Formular erstellen

Übungsdatei: --

Ergebnisdatei: Kontakt.sln

1. Erstellen Sie eine Windows-Anwendung als neues Projekt mit dem Namen Kontakt.
2. Erstellen Sie ein Formular entsprechend der Skizze.
3. Benennen Sie die Formularelemente mit aussagekräftigen Namen.
4. Benennen Sie das Formular mit dem Namen *KontaktForm*.
5. Speichern Sie die Arbeitsmappe unter dem Namen *Kontakt.sln*.

Hinweis: In dieser Übung soll das Formular lediglich gestaltet werden und besitzt anschließend keine Funktionalität.



6 Sprachgrundlagen von C#

6.1 Was ist die Syntax?

Für jede Programmiersprache gelten bestimmte Regeln, die festlegen,

- ✓ welche Schreibweisen und welche Bezeichnungen bei der Erstellung des Programmcodes erlaubt sind,
- ✓ welche Formulierungen verwendet werden dürfen (Kombination aus Text, Zeichen, Symbolen).

Dieses Regelwerk wird als **Syntax** bezeichnet.

Die Syntax von C# legt somit fest, wie Sie den Programmcode formulieren und welche Sprachmittel die Programmiersprache C# dazu bereitstellt.

Hinweis zur Schreibweise der Syntax in diesem Buch

- ✓ Schlüsselwörter werden fett hervorgehoben.
- ✓ Optionale Angaben stehen in eckigen Klammern `[]`.
- ✓ Alternative Elemente werden durch einen Schrägstrich `|` voneinander getrennt.
- ✓ Drei Punkte (...) kennzeichnen, dass weitere Angaben folgen können.
- ✓ Sofern eckige Klammern, ein Schrägstrich oder drei Punkte als Bestandteil des Programmcodes erforderlich sind, wird in der Erläuterung explizit darauf hingewiesen.

6.2 Bezeichner und Schlüsselwörter

C# Programme bestehen aus **Schlüsselwörtern**, die der Sprache selbst entstammen, und **Bezeichnern**, die Sie als Programmierer festlegen. Bezeichner und Schlüsselwörter werden durch Leerzeichen voneinander getrennt.

Bezeichner festlegen

Bezeichner (Identifier) sind frei wählbare Namen, mit denen Sie in Ihren C#-Programmen beispielsweise Konstanten, Variablen, Klassen und Methoden benennen. Bei der Programmierung können Sie über die Bezeichner auf diese Elemente zugreifen. Für den Aufbau eines Bezeichners gelten folgende Regeln:

- ✓ Die Namen der sogenannten reservierten Schlüsselwörter (vgl. nachfolgenden Abschnitt) dürfen nicht verwendet werden.
- ✓ Die nicht reservierten Schlüsselwörter sowie die Bezeichnungen von sogenannten Namensräumen bzw. Namespaces (z. B. `System`) sollten Sie ebenfalls nicht als Bezeichner verwenden.
- ✓ Sie sollten nur die Buchstaben (A - Z, a - z), die Ziffern (0 - 9) und den Unterstrich (`_`) verwenden, obwohl alle Unicode-Zeichen gültig sind (ein Zeichensatz, der über 2 Byte so gut wie alle weltweit zur Kommunikation notwendigen Zeichen abdeckt).
- ✓ Bezeichner beinhalten keine Leerzeichen und keine Sonderzeichen.
- ✓ Ein Bezeichner muss mit einem Buchstaben oder einem Unterstrich beginnen.
- ✓ Ein Bezeichner, der mit einem Unterstrich beginnt, muss auch mindestens einen Buchstaben oder eine Ziffer enthalten.
- ✓ Die Groß- und Kleinschreibung wird vom Compiler unterschieden.



C# unterscheidet zwar zwischen Groß- und Kleinschreibung, Sie sollten aber vermeiden, dass sich Bezeichner nur hinsichtlich der Groß- und Kleinschreibung unterscheiden. So reduzieren Sie die Verwechslungsgefahr und ermöglichen die gegenseitige Nutzung von Modulen mit anderen .NET-Sprachen, die zwischen Groß- und Kleinschreibung nicht unterscheiden, wie z. B. Visual Basic. Beachten Sie allerdings, dass es in C# spezielle Situationen gibt, in denen Bezeichner sich ausdrücklich nur in der Groß- und Kleinschreibung unterscheiden und damit aber ganz bewusst eine logische Beziehung aufgebaut wird. Darauf kommen wir zurück.

Für die aus Gründen der besseren Lesbarkeit empfehlenswerte Unterscheidung von Groß- und Kleinschreibung bei Bezeichnern gibt Microsoft folgende Richtlinien:

- ✓ Benennung nach der sogenannten Pascal-Schreibweise:
Der erste Buchstabe jedes Teilwortes beginnt mit einem Großbuchstaben, z. B. `ColorPicker`.
- ✓ Benennung nach der sogenannten Kamelschreibweise:
Im Unterschied zur Pascal-Schreibweise beginnt das **erste** Teilwort mit einem Kleinbuchstaben, z. B. `colorPicker`.
Durchgängige Großschreibung nur bei Benennung mit Abkürzungen aus zwei oder mehreren Buchstaben, z. B. `System.IO`.

Bezeichner hervorheben

Wenn Sie im Editor von Visual Studio mit der Maus einen Bezeichner im Quellcode anklicken, werden alle seine Verwendungen im Code des aktuellen Dokuments farbig hervorgehoben.

Zwischen den Fundstellen können Sie über die Tastenkombinationen **Strg** **↑** **↑** bzw. **Strg** **↑** **↓** navigieren.

```
public bool IsPrime(int value)
{
    if (value <= 1)
        return false;
    else
    {
        if (value > 2)
        {
            for (int i = 2; i <= (value / 2) + 1; i++)
                if (value % i == 0)
                    return false;
        }
    }
    return true;
}
```

Schlüsselwörter

C# besitzt **Schlüsselwörter**, die zum Schreiben eines Programms verwendet werden. Schlüsselwörter haben eine feste definierte Bedeutung. Es werden zwei Gruppen unterschieden:

- ✓ reservierte Schlüsselwörter,
- ✓ Kontextschlüsselwörter (nicht reserviert).

Reservierte Schlüsselwörter dürfen Sie grundsätzlich nicht als Bezeichner verwenden. Für die nicht reservierten Kontextschlüsselwörter, die nur an bestimmten Stellen im Programmcode eine spezielle Bedeutung haben, gilt diese Einschränkung nicht; jedoch ist die Verwendung als Bezeichner nicht empfehlenswert, da sie zu Missverständnissen und Fehlern führen kann.

In diesem Buch wird allgemein von Schlüsselwörtern gesprochen und nicht zwischen den beiden Gruppen (reservierte/nicht reservierte) unterschieden.

Schlüsselwörter in C#

Die folgende Übersicht zeigt alphabetisch geordnet die Schlüsselwörter von C#. Schlüsselwörter werden vollständig mit Kleinbuchstaben geschrieben. Der Code-Editor hebt Schlüsselwörter standardmäßig mit einer blauen Schriftfarbe hervor und verwendet automatisch die korrekte Kleinschreibung. In diesem Buch werden die Schlüsselwörter im abgebildeten Programmcode **fett** hervorgehoben.

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

Die folgenden Schlüsselwörter sind **Kontextschlüsselwörter**. Sie sind nicht reserviert, haben aber an bestimmten Programmstellen (in einem bestimmten „Kontext“) eine spezielle Bedeutung:

add	alias	async	await	dynamic	get
global	join	let	orderby	partial	remove
select	set	value	var	where	yield

Es sollte auffallen, dass sämtliche Schlüsselworte in C# kleingeschrieben werden!

6.3 Aufbau eines Programms

C# ist eine **objektorientierte Programmiersprache** (mit Klassen, Objekten und Methoden). Die Sprachelemente Klassen und Methoden werden in späteren Kapiteln vorgestellt.

Wenn Sie eine neue Konsolenanwendung oder eine Windows-Anwendung erstellen, wird automatisch eine sogenannte **Startklasse** angelegt, die standardmäßig den Namen *Program* besitzt. Die Startklasse der Konsolenanwendung unterscheidet sich etwas von der Startklasse der Windows-Anwendung. Dies ist aber für die zunächst folgenden Erläuterungen der grundlegenden Sprachelemente nicht von Bedeutung. Die Startklasse enthält immer die Methode `Main()`, die beim Programmstart zuerst ausgeführt wird.

Beachten Sie, dass `Main()` mit einem Großbuchstaben beginnt.

Aufbau der Startklasse einer Konsolenanwendung

- ✓ Die Startklasse wird mit dem Schlüsselwort `class` eingeleitet.
- ✓ Anschließend folgt der Name des Klasse (hier: `Program`).
- ✓ Der Inhalt der Klasse wird in geschweifte Klammern `{ } ①` eingeschlossen.
- ✓ Innerhalb des Moduls befindet sich die Methode `Main()`, die in jeder ausführbaren Anwendung einmal vorhanden sein muss. Sie wird mit den Schlüsselwörtern `static void` eingeleitet.
- ✓ Auch der Inhalt der Methode `Main()` wird in geschweifte Klammern `{ } ②` eingeschlossen. Die Methode ist direkt nach der Erstellung noch leer.
- ✓ Der Anwendung können Parameter übergeben werden, die beim Programmaufruf hinter dem Anwendungsnamen durch Leerzeichen abgetrennt geschrieben werden. Die Methode `Main()` besitzt Argumente, mit denen diese Parameter entgegengenommen werden können.

```
...
class Program
{   ①
    ...
    static void Main(string[] args)
    {   ②
        ...
    }   ②
    ...
}   ①
...
```

Aufbau der Startklasse einer Windows-Anwendung

- ✓ Eine Windows-Anwendung ist ähnlich aufgebaut. Der Klassendeklaration wird noch das Schlüsselwort `static` vorangestellt, das Sie in einem späteren Kapitel kennenlernen.
- ✓ Für die Methode `Main()` sind standardmäßig keine Argumente vorgesehen.
- ✓ Die Startmethode enthält bereits drei Programmzeilen ①. Die Programmierung nehmen Sie, wie im vorherigen Kapitel gezeigt, vorrangig in den Ereignismethoden der Formulare vor.

```
...
static class Program
{
    ...
    static void Main()
    {
        Application.EnableVisualStyles();
        ① Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
    ...
}
...
```

6.4 Programmcode dokumentieren

Mit Kommentaren den Überblick behalten

Bei der Entwicklung von Programmen sollten Sie mithilfe von Kommentaren den Programmcode an den Stellen erläutern, an denen wichtige Details nicht direkt aus dem Code gelesen werden können:

- ✓ Kommentare erleichtern es Ihnen, zu einem späteren Zeitpunkt die Funktionsweise einzelner Teile wieder nachzuvollziehen (warum wurde eine bestimmte Vorgehensweise gewählt?).

- ✓ Kommentare sollten nicht Dinge beschreiben, die sich direkt aus dem Code ablesen lassen, z. B. „hier kommt eine Schleife“. Die Bezeichner im Code sollten deshalb selbsterklärend sein (statt w oder fkt sollten Namen wie feldIndex oder Addiere verwendet werden).
- ✓ Wenn Kommentare dazu dienen, sehr komplizierten Code zu erläutern, sollten Sie darüber nachdenken, den Code zu vereinfachen. Dann ist er für Sie und andere einfacher verständlich und benötigt im Idealfall überhaupt keinen Kommentar mehr.

Kommentare nehmen Sie direkt in den Programmcode auf. Da die Kommentare aber vom Compiler nicht ausgewertet, sondern überlesen werden sollen, müssen Sie Ihre Erläuterungen im Programmcode speziell als Kommentar kennzeichnen.

Kommentare erstellen

In C# existieren zwei Kommentartypen:

Einzelige Kommentare	//	Alle nachfolgenden Zeichen der Programmzeile werden überlesen.
(mehrzeiliger) Kommentarblock	/* */	Er ermöglicht einen Kommentar auch über mehrere Zeilen. Ab der Zeichenkombination /* werden alle Zeichen im Programmcode überlesen (oder genauer – bei der Übersetzung des Quellcodes ignoriert), bis die Zeichenkombination */ auftritt.

Im Editor von Visual Studio werden Kommentare standardmäßig in grüner Schrift dargestellt.

```
static void Main(string[] args) // Die Startmethode
{
    /* Die Methode Main() enthält in diesem Beispiel nur
     * eine Programmzeile, die der Compiler berücksichtigt.
    ① * Alle anderen Zeilen sind Kommentare */
    // Den Text "Hallo Welt!" ausgeben
    Console.WriteLine("Hallo Welt!");
}
```

Wenn Sie einen Kommentarblock schreiben und am Ende einer Zeile die -Taste betätigen, wird automatisch in die neue Zeile ein Stern gefolgt von einem Leerzeichen eingetragen ①. Auf diese Weise sind Kommentarblöcke vor allem im Schwarz-Weiß-Ausdruck leichter zu erkennen.

Komplette Zeilen als Kommentar kennzeichnen

- Setzen Sie den Cursor in die entsprechende Zeile des Programmcodes.
 - oder Markieren Sie die Zeilen, die Sie als Kommentar kennzeichnen (auskommentieren) möchten.
 - Klicken Sie auf das Symbol .
- Vor die Programmzeile bzw. vor die markierten Programmzeilen werden automatisch zwei Schrägstriche gesetzt.



Mit dem Symbol oder durch Löschen der Schrägstriche // können Sie die Kennzeichnung als Kommentar einer oder mehrerer markierter Zeilen wieder entfernen.

Kommentare schachteln

- ✓ Ein Kommentarblock darf mit // gekennzeichnete einzeilige Kommentare beinhalten.
- ✓ Ein Kommentarblock darf **keinen** weiteren Kommentarblock enthalten.

Aufgabenkommentare einfügen

Eine besondere Form des Kommentars stellen **Aufgabenkommentare** dar. Sie beginnen mit `//` und dem Wort `TODO`. Anschließend geben Sie eine kurze Formulierung der noch zu erledigenden Aufgabe ein ①. Der Doppelpunkt ist optional.

```
static void Main(string[] args)
{
    // Den Text "Hallo Welt!" ausgeben
    Console.WriteLine("Hallo Welt!");
    // TODO: Benutzereingabe programmieren ①
}
```

Über den Menüpunkt *Ansicht - Aufgabenliste* können Sie die Aufgabenliste ② einblenden. Wenn Sie anschließend im Feld ③ auswählen, für welche Dokumente Sie die Kommentare sehen wollen, werden alle Aufgabenkommentare der Projektmappe, des Projekts oder nur eines Dokuments aufgelistet ④.



Das Aufgabenfenster

Die Dateien müssen dazu nicht geöffnet bzw. die Projekte müssen nicht aktiv sein, wenn Sie diese Auswahl getroffen haben. Mit einem Doppelklick auf eine Aufgabe gelangen Sie zur entsprechenden Programmzeile im Programmcode.

Dokumentation erstellen

Sie können mit Visual Studio eine XML-Dokumentation Ihres Programmcodes zusammenstellen. Dabei unterstützt Sie Visual Studio durch mehrere Features und automatisiert Standardvorgänge. XML (Extensible Markup Language) ist eine standardisierte Sprache für ein Dokumentformat, das sowohl von Menschen als auch von Computern gelesen werden kann. Bei der Erzeugung einer XML-Dokumentation werden spezielle Kommentarzeilen des Programmcodes ausgewertet, die jeweils mit drei Schrägstrichen beginnen und fest vorgegebene XML-Tags sowie Ihre eigenen Kommentare enthalten.

Dokumentationskommentare einfügen

- ▶ Setzen Sie den Cursor in eine leere Zeile **oberhalb** der Methode ⑤, die Sie dokumentieren möchten.
 - ▶ Betätigen Sie **dreimal** die `Shift`-Taste. In den Programmcode wird automatisch das Grundgerüst für die Dokumentation eingefügt. Sogenannte **Tags**, die paarweise vorhanden sind, gliedern die Informationen.
 - ▶ Schreiben Sie Ihre Erläuterungen zwischen die jeweiligen Tag-Paare. Öffnende Tags stehen in spitzen Klammern `< ...`
- `⑤` ① ...
 `② /// <summary></summary>` Tags
 `③ /// </summary>` Tags
 `④ /// <param name="args"></param>`
 `⑤ static void Main(string[] args)`
 {
 ...
 }
 ...
- `⑥` ①. Schließende Tags beinhalten zusätzlich vor dem Tag-Namen das Zeichen `/` ③.

```
① ...
    /// <summary></summary>
② ...
    /// </summary>
③ ...
    /// </param>
④ ...
    static void Main(string[] args)
{
    ...
}
...
```

- ①-③ Die Beschreibung für die Methode ⑤ schreiben Sie zwischen die Tags `<summary>` und `</summary>` in die Zeile ②.
- ④ Die Parameter der Methode können Sie mit den entsprechenden `param`-Tags beschreiben. Der Name des Parameters wird dazu im Attribut `Name` des Tags angegeben.



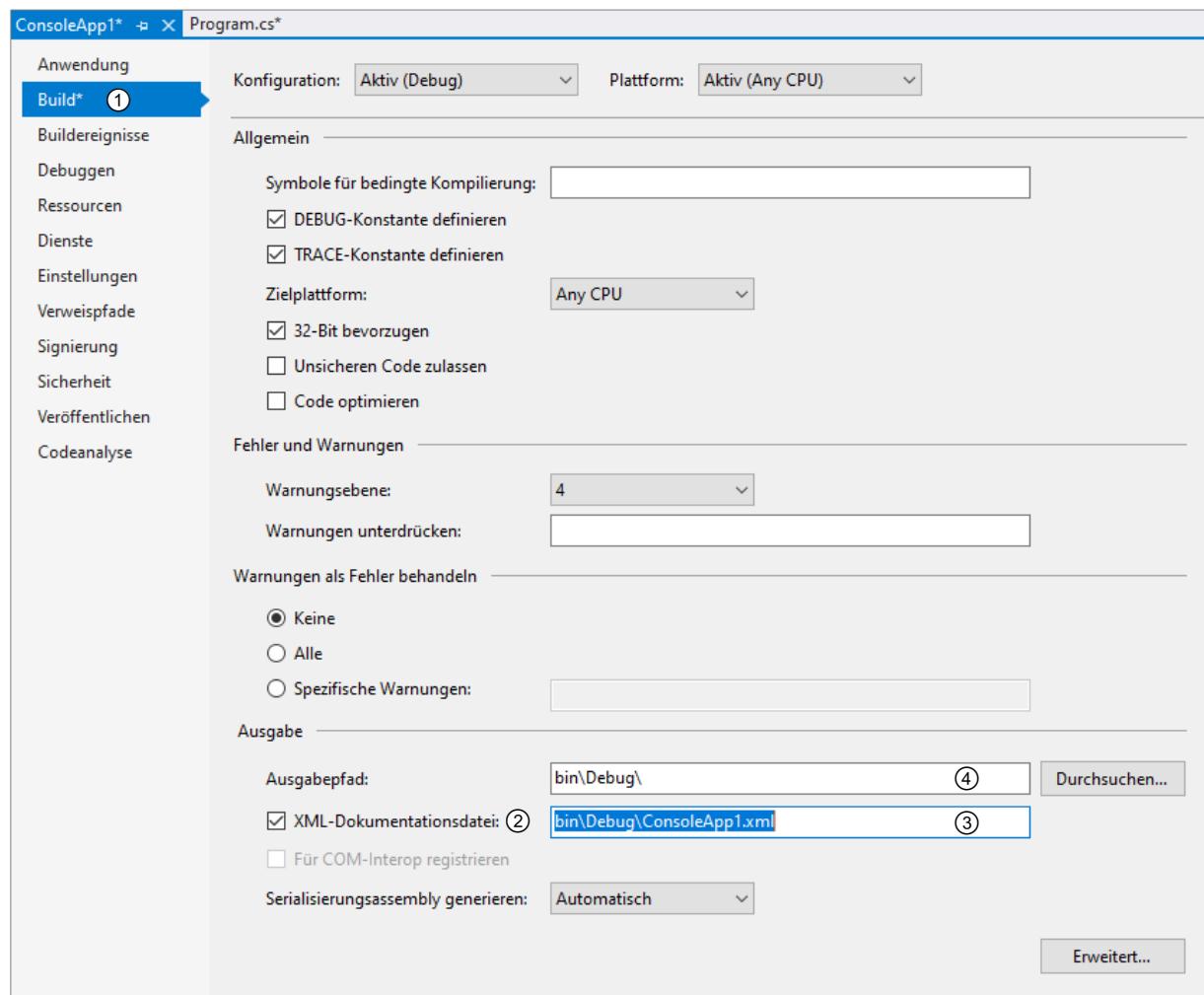
In Visual Studio und IntelliSense können Sie seit der Version 2015 zusätzlich auch JSDoc-Kommentare verwenden. JSDoc ist eine Auszeichnungssprache, die ursprünglich für JavaScript entwickelt wurde, um in JavaScript-Quellcode Kommentare zur automatisierten Dokumentation einzufügen zu können. Die universelle Verwendbarkeit in Visual Studio geht mit der stark ausgeprägten Cross-Over-Entwicklungsstrategie von Microsoft einher.

Erzeugung der Dokumentationsdatei aktivieren

Wenn Sie die Erstellung der Dokumentationsdatei aktiviert haben, wird diese beim Kompilieren automatisch generiert:

- Öffnen Sie im Projektmappen-Explorer das Kontextmenü des Projekts und wählen Sie den Kontextmenüpunkt *Eigenschaften*.

oder Markieren Sie im Projektmappen-Explorer das Projekt und klicken Sie auf .



- Wechseln Sie in das Register *Build* (in alten Versionen von Visual Studio auch *Erstellen*) ①.
 - Aktivieren Sie im unteren Bereich des Registers das Kontrollfeld ②.
- Geben Sie den Ordner und den Dateinamen für die Dokumentationsdatei an ③.

Standardmäßig wird die Dokumentationsdatei mit dem Namen des Projekts und der Dateinamenerweiterung *.xml* ③ im Unterordner *bin\Debug* ④ unterhalb des Projektordners gespeichert.

6.5 Anweisungen in C# erstellen

Was sind Anweisungen?

Anweisungen (Statements) beschreiben vom Programmierer erstellte Befehle zur Lösung einer Aufgabe. Ein Programm besteht aus einer Folge von Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden.

Syntax für Anweisungen

Die Syntax von Anweisungen ist durch folgende Regeln festgelegt:

- ✓ Eine einfache Anweisung wird durch ein Semikolon ; abgeschlossen.
- ✓ Eine Anweisung (Statement) besteht aus einer einfachen Anweisung oder aus einem Anweisungsblock.
- ✓ Sie können mehrere Anweisungen in eine Zeile schreiben. Aus Gründen der Übersichtlichkeit (und später auch beim Debuggen) sollten Sie diese Schreibweise jedoch **nicht verwenden**, sondern jede Anweisung in einer neuen Zeile beginnen.
- ✓ Um sehr lange Programmzeilen zu vermeiden, kann eine Anweisung in der nächsten Zeile fortgesetzt werden. Erst das Semikolon kennzeichnet das Anweisungsende.

Anweisungsblöcke

- ✓ Ein Anweisungsblock fasst mehrere einzelne Anweisungen in geschweiften Klammern { } zusammen.
- ✓ Die geschweiften Klammern müssen immer **paarweise** auftreten.
- ✓ Anweisungsblöcke können geschachtelt werden.

```
static void Main(string[] args)
{
    ...
    ...
    ...
}
```

Anweisungsblock

Die Programmzeilen eines Blocks werden jeweils einen Tabstopp eingerückt. Der Editor übernimmt diese Arbeit automatisch, wenn Sie das Semikolon am Anweisungsende eingeben oder einen Anweisungsblock mit } abschließen. Alternativ können Sie in den Visual-Studio-Einstellungen unter *Extras - Optionen, Text-Editor/C#/Tabstopps* auch Leerzeichen zum Einfügen einstellen.



Wenn Sie den Cursor vor eine öffnende bzw. hinter eine schließende geschweifte Klammer setzen, kennzeichnet der Editor das zusammengehörende Klammerpaar, indem er die entsprechenden Klammern farbig hinterlegt.

Programmcode manuell ein- bzw. ausrücken

Um den Programmcode manuell einzurücken, verwenden Sie die ↪-Taste oder markieren Sie die Programmzeilen und vergrößern bzw. verkleinern Sie den Einzug mit ⌘ ⌘ bzw. ⌘ ⌘ der Symbolleiste *Text-Editor* (diese Schaltflächen müssen Sie vorher manuell der Symbolleiste hinzufügen).

Die automatischen Formatierungen des Programmcodes können Sie beeinflussen, indem Sie den Menüpunkt *Extras - Optionen* aufrufen und in den Unterbereichen des Bereichs *Text-Editor/C#* die gewünschten Einstellungen vornehmen. Die gewöhnlich verwendeten Formatierungen sind voreingestellt.

Programmcode automatisch formatieren

Wenn Sie mit dem Cursor im Editor im Quelltext stehen, können Sie Programmcode automatisch formatieren lassen:

- Rufen Sie den Menüpunkt *Bearbeiten - Erweitert - Dokument formatieren* auf.

Alternative: **Strg** **E** und dann **D**



Die automatische Formatierung wird nicht durchgeführt, wenn der betreffende Programmcode fehlerhaft ist. Ebenso steht der Befehl nicht zur Verfügung, wenn Sie nicht mit dem Cursor im Editor im Quelltext stehen.

6.6 Einfache Datentypen

In den Anweisungen eines Programms arbeiten Sie mit Daten, die sich in ihrer Art unterscheiden:

- ✓ Zahlen (numerische Daten),
- ✓ boolesche (logische) Daten,
- ✓ Zeichen (alphanumerische Daten).

Um eine feste Basis an vordefinierten Datentypen bereitzustellen, besitzt C# sogenannte einfache (primitive) Datentypen. Diese unterscheiden sich nicht nur in der Art der Daten, sondern auch in dem zulässigen Wertebereich und in der Größe des dafür benötigten Speichers. Des Weiteren bestimmt der Datentyp die Operationen und Funktionen, die für diesen Datentyp angewendet werden können. Für jeden der einfachen Datentypen ist der jeweils benötigte Speicherplatz immer gleich.

Primitive Datentypen erkennen Sie daran, dass sie kleingeschrieben werden und Visual Studio kennzeichnet sie farblich besonders (analog anderer Schlüsselworte). Das Gegenstück zu primitiven Datentypen sind Referenztypen, auf die später zurückgekommen wird. Diese werden konsequent großgeschrieben und auch mit einer anderen Farbe in Visual Studio gekennzeichnet.

Alle primitiven Datentypen von C# entsprechen der Vorgabe aus dem CTS (Common Type System). Um ein Zusammenwirken der verschiedenen .NET-Programmiersprachen zu gewährleisten bzw. zu ermöglichen, sind in den CLS (Common Language Specifications) grundlegende Datentypen festgelegt, die alle .NET-Programmiersprachen unterstützen müssen. Einige der primitiven Datentypen von C# entsprechen zwar dem CTS, aber nicht den strengerem CLS. Diese Datentypen können **innerhalb** eines C#-Programms verwendet werden, können aber beim Zusammenspiel mit anderen Sprachen aus dem .NET-Framework Probleme bereiten. Wenn ein Zusammenwirken mit Komponenten erreicht werden soll, die in anderen .NET-Sprachen entwickelt wurden, dann sind in der Regel CLS-konforme Datentypen erforderlich.

Für alle primitiven Datentypen in C# gibt es äquivalente Referenztypen. Intern werden primitive Datentypen sogar als Referenztypen verwaltet.

(1) Numerische Datentypen

Die numerischen Datentypen werden dann benötigt, wenn im Programm mit Zahlenwerten gearbeitet werden soll. Sie werden beispielsweise für Berechnungen, Aufzählungen oder Nummerierungen eingesetzt. Es werden **Integer-Datentypen** (ganzzahlige Zahlenwerte) und **Gleitkomma-Datentypen** unterschieden.

Integer-Datentypen

Integer-Datentypen stellen ganze Zahlen (ohne Nachkommastellen) mit Vorzeichen dar. Die verschiedenen Varianten unterscheiden sich hinsichtlich des Wertebereichs. Bei einigen Datentypen wird das Vorzeichen nicht gespeichert; sie können daher lediglich positive Zahlen darstellen.

- ✓ Integer-Datentypen werden im Computer immer genau dargestellt.
- ✓ Es treten keine Rundungsfehler bei der Darstellung der Zahlen auf.

Die Integer-Datentypen von C# werden intern in die Datentypen der sogenannten .NET-Klassenbibliothek FCL (**Framework Class Library**) umgewandelt. Das sind besagte Referenztypen. Die Übersicht enthält die Namen dieser jeweils zugeordneten Referenzdatentypen in Klammern.

Datentyp (Datentyp der FCL)	Wertebereich	Speichergröße	
sbyte (SByte)	-128 ... 127	1 Byte	*
short (Int16)	-32768 ... 32767	2 Byte	
int (Int32)	-2.147.483.648 ... 2.147.483.647	4 Byte	
long (Int64)	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807	8 Byte	
byte (Byte)	0 ... 255	1 Byte	
ushort (UInt16)	0 ... 65.535	2 Byte	*
uint (UInt32)	0 ... 4.294.967.295	4 Byte	*
ulong (UInt64)	0 ... 18.446.744.073.709.551.615	8 Byte	*

Die mit (*) gekennzeichneten Datentypen sind **nicht CLS-konform**.

Üblicherweise werden Sie für ganzzahlige Werte den Datentyp `int (Int32)` verwenden, denn er bietet für die meisten Anwendungsfälle einen ausreichenden Wertebereich und gewährleistet auf 32-Bit-Computersystemen die optimale Ausführungsgeschwindigkeit.



Gleitkomma-Datentypen

Für reelle Zahlen werden Gleitkomma-Datentypen mit Vorzeichen verwendet. Der Computer kann jedoch nicht jede Zahl genau darstellen. Dies führt auch bei einfachen Berechnungen zu Rundungsfehlern. Je nach verwendetem Datentyp lassen sich Zahlen nur mit einer bestimmten Genauigkeit abbilden. Für eine höhere Genauigkeit wird aber auch mehr Speicherplatz benötigt.

Datentyp (Datentyp der FCL)	Wertebereich	Genauigkeit	Speichergröße
float (Single)	ca. +/- 1.4×10^{-45} ... 3.4×10^{38}	7–8 Stellen	4 Byte
double (Double)	ca. +/- 4.9×10^{-324} ... 1.8×10^{308}	15–16 Stellen	8 Byte

Zur Vermeidung von Rundungsfehlern sollten Sie Gleitkomma-Datentypen nur dann verwenden, wenn kein Integer-Datentyp für die zu lösende Aufgabe geeignet ist, beispielsweise wenn die Verwendung von Nachkommastellen erforderlich ist. Insbesondere sind Divisionsoperationen mit Gleitkomma-Datentypen in Hinsicht auf Rundungsprobleme sehr kritisch.



Der Datentyp `decimal`

Der Datentyp `decimal` (`Decimal`) besitzt eine hohe Genauigkeit (28 Nachkommastellen) und ist vor allem für Finanz- und Währungsberechnungen geeignet, da Rundungsfehler seltener auftreten als bei `double` oder `float`. Dennoch sind auch hier Rundungsprobleme nicht ausgeschlossen. Sie können ihn als ganze Zahl ① oder als Zahl mit **festen Nachkommastellen** ② einsetzen, was ihn hauptsächlich von `double` und `float` unterscheidet.

Datentyp (Datentyp der FCL)	Wertebereich	Genauigkeit	Speichergröße
decimal (Decimal)	① 0 ... +/- 79.228.162.514.264.337.593.543.950.335 ② +/- 1,0 x 10 ⁻²⁸ ... 7,9 x 10 ²⁸	28 Stellen	16 Byte

Mit dem .NET Framework 4.0 wurden zum Rechnen mit sehr großen sowie mit komplexen Zahlen die Typen `BigInteger` und `Complex` bereitgestellt. Allerdings handelt es sich hierbei nicht um primitive Datentypen.

(2) Zeichen-Datentypen

Zeichen-Datentypen in C# können beliebige Zeichen oder Zeichenketten des sogenannten Unicode-Zeichensatzes enthalten. Sie sind nicht auf Zahlen und Buchstaben begrenzt, sondern können auch Sonderzeichen wie ! "§\$%&/ sowie Buchstaben anderer Alphabete enthalten. Die Namen der Zeichen-Datentypen entsprechen den Namen der Datentypen in der FCL (Framework Class Library).

Datentyp (Datentyp der FCL)	Wertebereich	Speichergröße
char (Char)	Alle Unicode-Zeichen (0 ... 65535)	2 Bytes
string (String)	Alle Unicode-Zeichen (0 ... 65535)	maximal ca. 4 GB

(3) Boolescher (logischer) Datentyp

Zur Repräsentation von Wahrheitswerten (wahr bzw. falsch) dient der Datentyp `bool`. Ein Wahrheitswert kann nur die Literale `true` oder `false` als Wert annehmen.

Datentyp (Datentyp der FCL)	Wertebereich	Speichergröße
bool (Boolean)	<code>true</code> , <code>false</code>	1 Byte

6.7 Referenztypen

Es wurde oben angesprochen, dass es neben primitiven Datentypen in C# Referenztypen gibt. Für alle primitiven Datentypen gibt es solche Gegenstücke im FCL, die eine saubere Integration von Datentypen in die objekt-orientierte Welt erst gestatten. Auf die Details wird später genauer eingegangen. Hier soll nur noch ein weiterer Vertreter solcher Referenztypen neben den FCL-Typen vorgestellt werden, der einen hohen Bezug zu primitiven numerischen Datentypen hat, da er intern als eine ganze Zahl gespeichert wird.

(4) Datentyp `DateTime`

Datumsangaben lassen sich mit dem Datentyp `DateTime` speichern. Hier handelt es sich allerdings nicht mehr um einen primitiven Datentypen. Das erkennen Sie auch an der Großschreibung. Es gibt aber auch gar kein direktes primitives Gegenstück, wie etwa bei `Int32` mit `int`.

Datentyp	Wertebereich	Speichergröße
<code>DateTime</code>	1. Januar 1 bis 31. Dezember 9999, 0:00:00 bis 23:59:59	8 Byte

Die kleinste Einheit zum Messen der Zeit, die als **Tick** bezeichnet wird, entspricht 100 Nanosekunden ($100 * 10^{-9}$ s), ist aber prinzipiell betriebssystemabhängig.

6.8 Literale

Werte, die Sie im Programmcode eingeben (z. B. Zahlen), werden als **Literale** bezeichnet. Zum Beispiel sind Zahlen oder Texte Literale. Für ihre Schreibweise gelten entsprechende Regeln, damit der Datentyp ersichtlich ist.

Literale für numerische Datentypen

- ✓ Für numerische ganzzahlige Werte können Sie die Ziffern 0 ... 9 und die Vorzeichen + und - verwenden. Das Vorzeichen + kann entfallen. Die Verwendung von - ist nur möglich, wenn der Wertebereich negative Zahlen umfasst.
- ✓ Gleitkommazahlen enthalten einen Punkt . als Dezimaltrennzeichen.
- ✓ Für Fließkommawerte können Sie die Exponentialschreibweise verwenden. Den Wert $4,56 \cdot 10^{16}$ können Sie im Programmcode beispielsweise folgendermaßen eingeben: 4.56E16. Zahlen mit bis zu 15 Stellen vor dem Komma werden im Editor automatisch umgewandelt und ausgeschrieben.
- ✓ Zahlen mit Dezimalpunkt oder in Exponentialschreibweise werden standardmäßig als Zahlen vom Typ double interpretiert. Für Zahlen ohne Dezimalpunkt verwendet C# den Typ int.
- ✓ Wenn Sie explizit einen bestimmten Datentyp benötigen, ergänzen Sie ein sogenanntes Suffix. Verwenden Sie beispielsweise das F-Suffix (z. B. 12.6F bzw. 12.6f) für den Datentyp float.
- ✓ Für den vorzeichenlosen Datentyp uint verwenden Sie das Suffix U allein, bei ulong zusammen mit dem L-Suffix (z. B. UL oder UJ).

Suffix	Datentyp	Beispiel
L	long	56L
U	uint	74U
UL	ulong	23LU
M	decimal	12M
F	float	475F
D	double	77D

Wenn Sie beim Datentyp long das Suffix L kleinschreiben, erfolgt eine Compilerwarnung, da das l leicht mit der Ziffer 1 verwechselt wird. Sie sollten für das Suffix daher immer den Großbuchstaben L verwenden.

Literale für Zeichen

- ✓ Einzelne Zeichen werden in Apostrophe ' ... ' eingeschlossen.
- ✓ Sie können ein Zeichen auch als Unicode-Escape-Sequenz darstellen. Die Unicode-Repräsentation ist dann ebenfalls in Apostrophe zu setzen. Escape-Sequenzen beginnen mit einem Backslash \. Die nebenstehende Tabelle zeigt einige Beispiele.
- ✓ Mit der Escape-Sequenz \u können Sie den entsprechenden Zeichencode direkt eingeben.

Escape-Sequenz	Bedeutung
\u... Beispiel: \u0045	Ein spezielles Zeichen (im Beispiel das Zeichen E mit dem Code 0045)
\b	Backspace
\t	Tabulator
\n	Zeilenumbruch
\f	Seitenvorschub
\r	Wagenrücklauf
\"	Anführungszeichen
\\	Backslash

Literale für Zeichenketten

Für Zeichenketten bestehen zwei Möglichkeiten zur Formulierung der Literale.

- ✓ Regular (reguläres) String-Literal
- ✓ Verbatim (wörtliches) String-Literal

Reguläre String-Literale	Die Zeichenkette wird in Anführungszeichen „ „ ... „ „ eingeschlossen. Für die Eingabe einer leeren Zeichenkette verwenden Sie zwei direkt aufeinanderfolgende Anführungszeichen (" ") oder die Eigenschaft <code>Empty</code> der Klasse <code>String</code> (<code>String.Empty</code>). Für die Darstellung spezieller Zeichen nutzen Sie die Escape-Sequenzen: Geben Sie z. B. \\ ein, um einen Backslash \ darzustellen, oder schreiben Sie \", um das Anführungszeichen (") in der Zeichenkette zu verwenden.
Verbatim-String-Literale	Durch Voranstellen des Zeichens @ wird eine Zeichenkette als Verbatim-String-Literal deklariert. Escape-Sequenzen werden in Verbatim-String-Literalen nicht ausgewertet, sondern als Teil der Zeichenkette interpretiert. Das Zeichen \ wird somit als Zeichen des Textes verwendet. Verbatim-String-Literale sind dann sinnvoll, wenn Sie z. B. den Backslash \ für die Angabe von Ordnerstrukturen benötigen. Um bei dieser Schreibweise ein Anführungszeichen „ im Text darzustellen, schreiben Sie statt der Escape-Sequenz \" zwei direkt aufeinanderfolgende Anführungszeichen \"\".

Die folgende Tabelle zeigt Beispiele für reguläre String-Literale und die entsprechenden Verbatim-String-Literale:

Reguläres String-Literal	Verbatim-String-Literal
"c:\\uebung\\vcspnet2022	@"c:\\uebung\\vcspnet2022"
"Klicken Sie auf \"Start\""	@"Klicken Sie auf ""Start"""

Literale für boolesche Daten

Für die Eingabe eines logischen Wertes existieren die beiden Literale `true` (wahr) und `false` (falsch).

6.9 Mit Variablen arbeiten

Was sind Variablen?

Die Anweisungen eines Programms arbeiten mit Daten, die beispielsweise als Zwischenergebnis bei Berechnungen immer wieder verändert werden und somit variable Werte darstellen. Dazu werden sogenannte **Variablen** verwendet, für die der entsprechende Speicherplatz im Arbeitsspeicher reserviert wird. Im Programm greifen Sie über den **Variablennamen** bzw. **Bezeichner** auf diesen Speicherplatz zu.

In C# ist eine **explizite Deklaration** der Variablen vorgeschrieben. Das bedeutet, dass eine Variable vor ihrer ersten Verwendung deklariert (eingeführt) sein muss. Variablen werden in C# mit einem Namen und einem Datentyp deklariert. Eine Ausnahme sind die mit dem .Net Framework 3.5 neu eingeführten implizit typisierten lokalen Variablen, die später noch erläutert werden.

Gültigkeitsbereich lokaler Variablen

C# kennt verschiedene Arten von Variablen. Variablen, die innerhalb eines Anweisungsblocks einer Methode (z. B. `Main()`) definiert werden, werden als **lokale Variablen** bezeichnet und sind auch nur innerhalb der Methode gültig. Dieser Bereich wird **lokaler Gültigkeitsbereich** genannt. Die lokale Variable verliert mit dem Ende des Blocks ihre Gültigkeit.

- ✓ Ein Gültigkeitsbereich wird durch ein Paar geschweifte Klammern festgelegt. Ein Bezeichner ist dabei innerhalb des umgebenden Klammerpaars und aller darin verschachtelten Klammerpaare gültig.
- ✓ In einem solchen Bereich darf es immer nur einen gültigen Bezeichner mit demselben Namen geben. In einigen Fällen führen gleiche Namen zu einem Compilerfehler (z. B. Verwendung zweier lokaler Variablen mit dem gleichen Namen), in anderen überschreiben Sie einen Bezeichner im umgebenden Bereich (eine lokale Variable überschreibt eine Instanzvariable (das wird später noch genauer erläutert) mit dem gleichen Namen).

Die folgende Gegenüberstellung zeigt Variablendefinitionen, die aufgrund der Eindeutigkeit und der Gültigkeitsbereiche zulässig bzw. unzulässig sind:

Zulässige Variablendefinition	Unzulässige Variablendefinition
<pre>static void Main(string[] args) { int number = 5; ② }① // die Variable number ist hier nicht // mehr gültig { int number = 12; ③ // jetzt existiert eine neue lokale // Variable mit dem Namen number }</pre>	<pre>static void Main(string[] args) { int number = 5; ④ ... }⑤ ... int number = 12; ⑥ // diese Variablendefinition ist nicht // zulässig, da die Variable number // aus dem übergeordneten Block noch // gültig ist.</pre>

Zwei Beispiele für den Gültigkeitsbereich einer Variablen

- ✓ Nach dem Blockende ① (geschweifte schließende Klammer) ist die im Block definierte Variable ② nicht mehr gültig. Es kann eine neue Variable mit dem gleichen Namen definiert werden ③.
- ✓ Da Blöcke geschachtelt werden können, ist die Variable ④ auch im untergeordneten Block ⑤ gültig. Eine Variablendefinition mit demselben Namen ⑥ ist hier also **unzulässig**.

Syntax der Variablendefinition

- ✓ Die Deklaration von Variablen wird mit dem Namen des Datentyps eingeleitet.
- ✓ Anschließend folgt der Variablenname (identifier = Bezeichner).
- ✓ Mehrere Variablen desselben Typs können Sie in einer Anweisung definieren. Die Variablennamen werden dabei durch Kommata getrennt.
- ✓ Wie jede Anweisung wird die Deklaration einer Variablen mit einem Semikolon abgeschlossen.
- ✓ Die Namen der Variablen halten sich an die Vorgaben für Bezeichner, wobei lokale Variablen üblicherweise mit einem Kleinbuchstaben beginnen.

```
type1 identifier1[, identifier2...];
```

Beispiele für Variablendefinitionen: VariablenDefinition.sln

```
// --richtige Definition -----
int zahl;
double anzahl, preis, gewicht;
char zeichen;
// --falsche Definition -----
// int &menge;           // Name beginnt nicht mit _ oder a..Z
// double Menge der Waren; // Name enthält Leerzeichen
```



Sie erhalten eine Warnung des Compilers, wenn Sie eine Variable deklariert haben, die niemals verwendet wird. Auf diese Weise wird Ihnen das „Aufräumen“ des Programmcodes erleichtert.

Implizit typisierte lokale Variablen

Für lokale Variablen existiert seit dem .NET Framework 3.5 eine neue Deklarationsvariante: Es wird kein Datentyp mehr bei der Variablendeklaration angegeben, sondern der Compiler leitet den Datentyp aus dem der Variablen zugewiesenen Wert ab. Diese Erweiterung der Sprache C# wurde hauptsächlich für LINQ-Abfragen implementiert (die Beschreibung der Verwendung von LINQ ist nicht Bestandteil dieses Buchs). Die stark im Fokus stehende Cross-Over-Entwicklung mit Web-Technologien wie JavaScript wird zusätzlich durch implizit typisierte Variablen vereinfacht. Sie sollten aufgrund der fehlenden Typprüfung durch den Compiler implizit typisierte Variablen sparsam einsetzen bzw. auf sie verzichten, wenn diese nicht unbedingt notwendig sind.

Über das Schlüsselwort `var` beginnen Sie die Deklaration. Es folgen der Bezeichner der Variablen und eine Wertzuweisung. Anhand der Wertzuweisung ermittelt der Compiler den Typ der Variablen. Im Unterschied zu „normalen“ Variablen müssen Sie implizit typisierte Variablen immer einzeln deklarieren.

Zulässig	Unzulässig
<code>var nummer = 123;</code>	<code>var nummer1 = 123, nummer2 = 456;</code>

Sie können implizit typisierte Variablen in `for`- und `foreach`-Schleifen (vgl. Kapitel 7) sowie der `using`-Anweisung (vgl. Kapitel 9) einsetzen.

Beachten Sie, dass solche Variablen zwar implizit Ihren Datentypen zugewiesen bekommen, dann aber im Datentyp festgelegt sind und danach **nicht** wieder den Datentypen ändern können, was bei lose typisierten Sprachen der Fall ist.

Beispiele für die Deklaration implizit typisierter Variablen: *ImpTypVar.sln*

	<pre>static void Main(string[] args) { ① var nummer = 123; ② Console.WriteLine(numero.GetType().ToString()); ③ for(var i = 0; i < 5; i++) Console.WriteLine(i); }</pre>
--	--

- ① Die Variable `nummer` wird mit dem Wert 123 belegt und somit als `Integer`-Variable behandelt.
- ② Mittels der Methode `GetType()` wird der vom Compiler ermittelte Typ der Variablen `nummer` bestimmt und mittels `ToString()` in eine Zeichenkette umgewandelt. Das Ergebnis wird ausgegeben.
- ③ Implizit deklarierte Variablen können auch in Schleifenkonstruktionen verwendet werden, z. B. in einer `for`-Schleife. Diese Schleife wird fünfmal durchlaufen. Der Variablen `i` wird der Wert 1 zugewiesen. Dadurch wird vom Compiler ebenfalls der Datentyp `Integer` für diese Variable verwendet.
- ④ In jedem Schleifendurchlauf wird der Wert der Variablen `i` ausgegeben.

6.10 Werte zuweisen

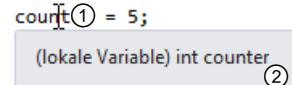
Wertzuweisung und Initialisierung von Variablen

Eine lokale Variable muss vor der ersten Verwendung initialisiert sein. Dazu wird der Variablen ein Wert zugewiesen. Nach der Initialisierung kann im weiteren Verlauf des Programms der Wert einer Variablen geändert werden; es wird ein neuer Wert zugewiesen. Die Wertzuweisung an Variablen erfolgt nach der folgenden Syntax:

Syntax für Wertzuweisungen an Variablen

- ✓ Die Wertzuweisung beginnt mit dem Bezeichner der Variablen. `identifier = expression;`
- ✓ Anschließend folgt als **Zuweisungsoperator** ein Gleichheitszeichen `=` und der Ausdruck (expression), der zugewiesen werden soll.
- ✓ Häufig ist der Ausdruck ein Wert, den Sie direkt, beispielsweise als Zahl (Literal), eingeben. Der Ausdruck kann aber auch eine andere Variable sein, sofern diese bereits einen Wert besitzt, oder ein komplexer Ausdruck. Komplexe Ausdrücke werden Sie im weiteren Verlauf dieses Kapitels kennenlernen.

Wenn Sie mit der Maus auf eine Variable zeigen ①, dann erhalten Sie in einer QuickInfo ② Informationen zu der Variablen. Sie erfahren beispielsweise, ob es sich um eine lokale Variable handelt und welchen Datentyp die Variable besitzt.



Tipps und Hinweise zur Schreibweise

Sie können die Definition und die Initialisierung einer Variablen in einer einzigen Anweisung vornehmen. Dabei fügen Sie die Wertzuweisung direkt an die Definition an.

`[Datentyp] identifier = value;`

Syntax für Definition und Initialisierung in einer Anweisung



6.11 Tipps zur Arbeit mit Variablen

Tippfehler vermeiden

Bei der Anwendung von Variablen, z. B. bei Wertzuweisungen, können Sie die IntelliSense nutzen. Die Namen aller an der aktuellen Programmcodeposition gültigen Variablen werden in der Memberliste zur Auswahl gestellt. So vermeiden Sie Tippfehler.

Zur Variablendefinition springen

- ▶ Klicken Sie im Programmcode mit der rechten Maustaste auf den Namen der Variablen.
- ▶ Wählen Sie den Kontextmenüpunkt *Gehe zu Definition*, um schnell zu der Programmzeile zu springen, in der die Variable definiert wurde.

Bezeichner ändern

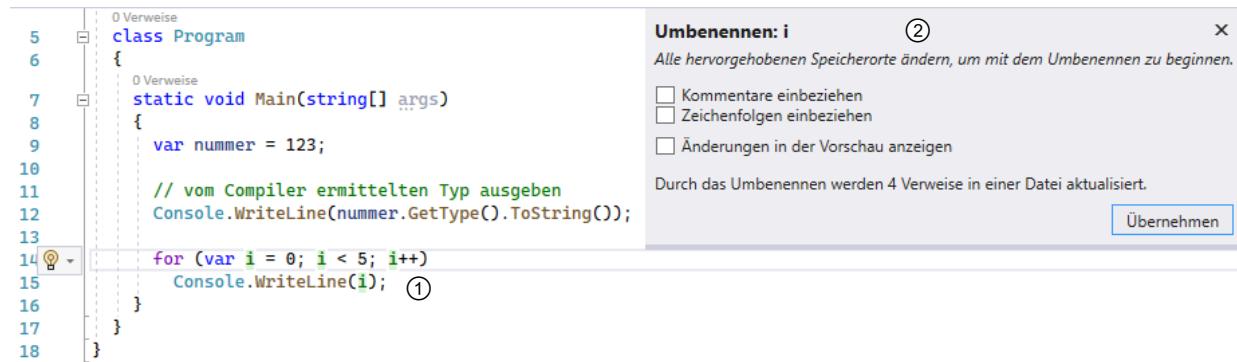
Visual Studio unterstützt Sie beim Umbenennen von Variablen und Konstanten. So ist es nicht erforderlich, den Namen an allen Verwendungspositionen manuell zu bearbeiten. Sie kann an einer beliebigen Stelle des Codes im Editor erfolgen.

1. Schritt: Neuen Variablennamen festlegen

- ▶ Klicken Sie einen Bezeichner im Quellcode an, um ihn auszuwählen.
- ▶ Wählen Sie im Kontextmenü des Variablenamens den Befehl *Umbenennen*. Alternativ können Sie im Menü *Bearbeiten* den Befehl *Umgestalten - Umbenennen* auswählen.
- ▶ Geben Sie den neuen Bezeichner im Editor an einer beliebigen Stelle ein ①. Alle weiteren Vorkommen des Bezeichners im Projekt werden automatisch geändert.

oder

- ▶ Wenn Sie den Variablenamen ändern, erhalten Sie einen Smarttag-Anzeiger ②.
- ▶ Zeigen Sie auf den Anzeiger, um das Smarttag einzublenden.
- ▶ Wählen Sie im Smarttagmenü den Befehl *Preview Änderungen* oder in manchen Versionen von Visual Studio komplett auf Deutsch, *Änderungen in der Vorschau ansehen*.

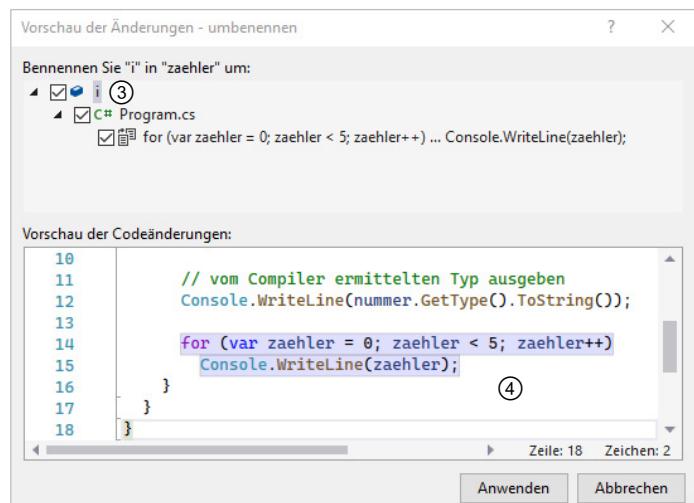


Anschließend wird ein Dialogfenster geöffnet, in dem Sie festlegen, welche Änderungen durchgeführt werden sollen.

2. Schritt: Änderungen bestätigen bzw. ablehnen

In dem Dialogfenster zur Vorschau der Änderungen werden automatisch alle Positionen ③ angezeigt, in denen die Variable auftritt. Außerdem erhalten Sie eine Vorschau auf den geänderten Programmcode ④.

- ▶ Lehnen Sie ggf. eine Änderung ab, indem Sie das entsprechende Häkchen ③ durch Anklicken entfernen. Standardmäßig sind alle Änderungen aktiviert.
- ▶ Bestätigen Sie mit *OK*, um die ausgewählten Änderungen durchzuführen. Andernfalls werden die Änderungen nicht durchgeführt.



6.12 Typkompatibilität und Typkonversion

Kompatible und nicht kompatible Datentypen

- Weisen Sie einer Variablen den Wert des Datentyps zu, den sie selbst besitzt oder den sie aufgrund des Wertebereichs umfasst, ist der Datentyp des Wertes **zuweisungskompatibel** zum Datentyp der Variablen. Zum Beispiel ist der Datentyp `int` **zuweisungskompatibel** zum Datentyp `double`, da der Datentyp `double` den Wertebereich des Datentyps `int` einschließt. Bei der folgenden Zuweisung gehen daher keine Informationen verloren:

```
double number = 2;
```

- Wird bei der Zuweisung der Wertebereich eingeschränkt, sind die Datentypen **nicht zuweisungskompatibel**. Beispielsweise ist der Datentyp `double` **nicht zuweisungskompatibel** zum Datentyp `int`, da der Datentyp `int` nur einen Teil des Wertebereichs des Datentyps `double` umfasst. Bei der folgenden Zuweisung gehen daher Informationen (die Nachkommastellen) verloren:

```
int counter = 2.45; // Diese Anweisung erzeugt eine Fehlermeldung
```

Implizite und explizite Typkonversion

Bei zuweisungskompatiblen Datentypen führt C# automatisch eine **implizite Typkonversion** durch. So können Sie beispielsweise eine Ganzzahl (Typ `int`) problemlos einer Variablen vom Typ `double` zuweisen.

Sind die Datentypen nicht zuweisungskompatibel, so können Sie eine **explizite Typkonversion** durchführen. Logische Werte (Typ `bool`) können nicht umgewandelt werden.

Syntax für die explizite Typkonversion

- Schreiben Sie in runden Klammern `()` den Datentyp, in den Sie den Wert bzw. Ausdruck umwandeln möchten.
- Anschließend folgt der umzuwandelnde Wert bzw. Ausdruck.

```
(Datentyp) expression;
```

Die explizite Typkonversion wird auch als **Casten** (engl. casting) bezeichnet.

Weitere Möglichkeiten zur expliziten Typkonversion

Die Klasse `Convert`

Sie können auch mit den Methoden der Klasse `System.Convert` Typen konvertieren. Die verfügbaren Konvertierungsmethoden werden Ihnen beispielsweise über die Memberliste der IntelliSense angezeigt.

```
int i;
string s = "6373";
i = System.Convert.ToInt32(s);
```

Die Methoden `ToString()` und `Parse()`

Für viele Standarddatentypen stehen die Methoden `ToString()` und `Parse()` zur Verfügung, mit denen auf einfache Weise Zahlen in Zeichenketten und umgekehrt konvertiert werden können.

```
string s;
int i = 10;
s = i.ToString();
i = Int32.Parse(s);
```

Beispiele für Typumwandlungen (Typkonversionen): *TypeCast.sln*

```

int number = 4567;
double size = 123.12;
char ch = 'K';
string txt;

① number = (int)size;
② size = 149;
③ txt = ch.ToString();
④ txt = number.ToString();
⑤ number = Convert.ToInt32("4567");
⑥ size = Double.Parse("4567,89");
⑦ size = Convert.ToDouble("4567,89");
⑧ ch = Convert.ToChar("k");

```

- ① Explizite Umwandlung einer Dezimalzahl in eine Ganzzahl: `number` erhält den Wert 123
- ② Implizite Umwandlung einer Ganzzahl in eine Dezimalzahl: `size` erhält den Wert 149.0
- ③ Explizite Umwandlung eines Zeichens in eine Zeichenkette (String): `txt` erhält den Text "K"
- ④ Explizite Umwandlung einer Ganzzahl in eine Zeichenkette: `txt` erhält den Text "123"
- ⑤ Explizite Umwandlung einer Zeichenkette in eine Ganzzahl: `number` erhält den Wert 4567
- ⑥ Explizite Umwandlung einer Zeichenkette in eine Dezimalzahl: `size` erhält den Wert 4567.89
- ⑦ Weitere Möglichkeit zur expliziten Umwandlung einer Zeichenkette in eine Dezimalzahl: `size` erhält den Wert 4567.89
- ⑧ Explizite Umwandlung eines Strings mit **einem Zeichen** in ein Zeichen (char): `ch` erhält den Wert 'k'

Datentypumwandlung – eine einfache Dateneingabe über die Konsole

Eine Möglichkeit zur **Datenausgabe** haben Sie bereits kennengelernt. Sie können dazu in einer Konsolenanwendung die Anweisung `Console.WriteLine(...)` verwenden.

Wenn eine **Dateneingabe** erforderlich ist, gibt der Anwender des Programms über die Tastatur immer eine Zeichenkette ein. Das Einlesen kann bei einer Konsolenanwendung mit der Anweisung `Console.ReadLine()` erfolgen. Anschließend muss die eingelesene Zeichenkette in den entsprechenden Datentyp umgewandelt werden.

Beispiel für Daten einlesen und Datentyp umwandeln: *IntegerInput.sln*

```

string s;
int i;
// Ausgabe eines Textes ohne Zeilenumbruch
① Console.Write("Geben Sie eine ganze Zahl ein: ");
② s = Console.ReadLine();
③ i = Int32.Parse(s);
④ Console.WriteLine("Die Zahl " + i + " wurde eingegeben");

```

- ① Der Anwender erhält die Aufforderung zur Eingabe einer Zahl. Statt `WriteLine` wird hier `Write` verwendet, damit kein Zeilenumbruch ausgeführt wird und die anschließende Eingabe in derselben Zeile erfolgt.
- ② Die Zeichenkette `s` wird eingelesen.

- ③ Die Zeichenkette wird in den Datentyp int umgewandelt.
- ④ Bei der Ausgabe können Sie Zeichenketten und Zahlenwerte mit dem Operator + verknüpfen. C# wandelt in diesem Fall Zahlenwerte dabei automatisch in einen String um.

Ausgabe mit Platzhaltern

- ✓ Sie können Platzhalter vorsehen ①/②, die später mit den Werten der darauf folgenden Variablen ③ gefüllt werden.
- ✓ Die Platzhalter werden in geschweifte Klammern eingeschlossen und geben mit Null beginnend die Position des einzusetzenden Wertes in der folgenden Werteliste an.
- ✓ Über die Platzhalter können Sie weitere Formatierungen vornehmen. Nach einem Komma lässt sich bestimmen, wie viel Platz (in Zeichen) für die entsprechende Variable zu verwenden ist ①. Nach einem Doppelpunkt können Formatbezeichnungen stehen (hier c für das Währungsformat), gefolgt von einer weiteren Zahl, die für die Anzahl der Dezimalstellen steht ②.

```
Console.WriteLine("{0,5};{1:c2}", s, i)
```

Auswahl der Formatierungszeichen

Zeichen	Beschreibung
0	Platzhalter für Ziffern: An dieser Stelle wird eine Ziffer oder ersatzweise eine Null angezeigt.
#	Platzhalter für Ziffern: An dieser Stelle wird eine Ziffer oder ersatzweise ein Leerzeichen angezeigt.
.	Platzhalter für Dezimaltrennzeichen
,	Platzhalter für Tausendertrennzeichen (Zifferngruppierung)
%	Platzhalter für das Prozentzeichen
+ -	Vorzeichen (Plus oder Minus)
C oder c	Dieses Währungsformat erzeugt eine Zahl (wenn nötig) mit Tausendertrennzeichen. Es werden zwei Dezimalstellen angezeigt.
E oder e	Wissenschaftliche Ausgabe (exponentielle Schreibweise)
F oder f	Zeigt mindestens eine Ziffer links und zwei Ziffern rechts vom Dezimaltrennzeichen an
G oder g	Allgemeine Ausgabe
N oder n	Ausgabe mit Tausendertrennzeichen
P oder p	Die Ausgabe erfolgt in Prozent mit zwei Dezimalstellen.
X oder x	Hexadezimale Ausgabe

Mit Datumsangaben arbeiten

Datumsangaben können Sie nicht direkt durch eine Wertzuweisung eingeben, sondern:

- ✓ Sie verwenden beispielsweise die Methode ToDateTime() der Klasse Convert
oder
- ✓ Sie nutzen die Methode Parse() der Klasse DateTime

```
DateTime datum;
...
datum = Convert.ToDateTime("01/01/20");
datum = Convert.ToDateTime("01.01.2020");
datum = Convert.ToDateTime("01. Januar 2020");
datum = Convert.ToDateTime("Januar 01, 2020");
datum = DateTime.Parse("01.01.2020");
```

Innerhalb der Zeichenkette können Sie das Datum in verschiedenen Formaten angeben.

Auswahl spezieller Formatierungszeichen für die Ausgabe von Datumswerten

Zeichen	Beschreibung
HH : mm : ss	Zeigt die Stunden, Minuten und Sekunden als Zahlen mit führender 0 an. Die Stunden werden im 24-Stunden-Format angezeigt, bei Verwendung von hh statt HH im 12-Stunden-Format.
dddd bzw. dd	Zeigt den Tag ausgeschrieben bzw. als zweistellige Zahl mit führender 0 an
MMMM bzw. MM	Zeigt den Monat ausgeschrieben bzw. als zweistellige Zahl mit führender 0 an
yyyy bzw. yy	Zeigt das Jahr als vierstellige Zahl bzw. als zweistellige Zahl mit führender 0 an
d	Zeigt einen Datumswert entsprechend dem kurzen Datumsformat des ausgewählten Gebietsschemas an
D	Zeigt einen Datumswert entsprechend dem langen Datumsformat des ausgewählten Gebietsschemas an
t	Verwendet für eine Anzeige von Stunden und Minuten das 24-Stunden-Format
T	Zeigt Stunden, Minuten und Sekunden entsprechend dem langen Zeitformat des ausgewählten Gebietsschemas an
f	Verwendet das lange Datumsformat und das kurze Zeitformat
F	Es werden das lange Datumsformat und das lange Zeitformat des in der Systemsteuerung des Betriebssystems ausgewählten Gebietsschemas benutzt.
g	Das kurze Datumsformat und das kurze Zeitformat werden verwendet.
G	Das kurze Datumsformat und das lange Zeitformat werden verwendet.

6.13 Konstanten

Konstanten nutzen

In einem Programm werden häufig Werte benötigt, die sich nicht mehr ändern, wie beispielsweise die Schallgeschwindigkeit, ein Mehrwertsteuersatz oder der Umrechnungsfaktor von mm in Inch. Diese Konstanten werden durch eine spezielle Schreibweise dargestellt. Bei der Definition der Konstanten erfolgt auch die Wertzuweisung, die endgültig ist. Sie können also keine zweite Wertzuweisung an eine Konstante vornehmen.

Konstanten vereinfachen die Lesbarkeit und vermeiden Fehler:

- ✓ Der Wert einer Konstanten wird nur einmal eingegeben.
- ✓ Im weiteren Verlauf des Programms arbeiten Sie nur noch mit dem Namen der Konstanten.
- ✓ Sofern der Wert der Konstanten geändert werden muss, ist nur an einer einzigen Stelle im Programmcode diese Änderung vorzunehmen.

Wie für eine lokale Variable wird auch für jede Konstante Speicherplatz im Arbeitsspeicher Ihres Computers reserviert. Im Programm greifen Sie auf diesen Bereich über den Namen der Konstanten zu. Eine Konstante ist wie eine Variable nur in dem Anweisungsblock gültig, in dem sie definiert wurde.

Syntax der Konstantendefinition und -initialisierung

- ✓ Die Konstantendefinition wird mit dem Schlüsselwort const eingeleitet. `const type identifier = expression;`
Die Ausgabe des Programms
- ✓ Anschließend geben Sie den Datentyp an.
- ✓ Es folgt der Name der Konstanten, der im jeweiligen Gültigkeitsbereich eindeutig sein muss.

- ✓ Anschließend geben Sie nach einem Gleichheitszeichen den Ausdruck (expression) an, der der Konstanten zugewiesen werden soll.
- ✓ Die Namen der Konstanten halten sich an die Vorgaben für Bezeichner. Üblicherweise werden Konstanten komplett großgeschrieben.

6.14 Arithmetische Operatoren sowie Vorzeichen- und Verkettungsoperatoren

Ausdrücke mit Operatoren bilden

Häufig werden Berechnungen benötigt, deren Ergebnisse anschließend in Variablen gespeichert werden. Mithilfe von **Operatoren** können Sie Ausdrücke bilden, die z. B. eine Formel zur Berechnung darstellen.

Ein **Ausdruck** (expression) ist allgemein eine Kombination aus Werten, Variablen, Konstanten und Operatoren.

- ✓ Der Ausdruck wird ausgewertet und der am Ende ermittelte Wert einer Variablen zugewiesen.

```
identifier = expression;
```

Einige Operatoren, wie die zur Addition $(+)$, Subtraktion $(-)$, Multiplikation $(*)$ und Division $(/)$, kennen Sie als Grundrechenarten. Für die Ausführungsreihenfolge in einem Ausdruck werden Operatoren nach Prioritäten eingeteilt. Die Ausführungsreihenfolge kann jedoch auch durch das Setzen von runden Klammern $()$ festgelegt werden, die die höchste Priorität besitzen. Der Inhalt der Klammern wird immer zuerst ausgewertet.

Folgende Operatoren können Sie in C# verwenden:

- ✓ **Arithmetische Operatoren**, z. B. für die einfachen Grundrechenarten
- ✓ **Vorzeichenoperatoren** für die Kennzeichnung positiver bzw. negativer Zahlen
- ✓ **Verkettungsoperatoren** zur Verbindung von Zeichenketten
- ✓ **Inkrementierungs-/Dekrementierungsoperatoren** zur Erhöhung bzw. Verringerung eines Wertes um den Betrag 1
- ✓ **Vergleichsoperatoren** zum Vergleich zweier Werte, z. B. auf Gleichheit oder Ungleichheit
- ✓ **Verknüpfungsoperatoren** zur logischen Verknüpfung boolescher Ausdrücke bzw. zur binären Verknüpfung binärer Werte
- ✓ **Zuweisungsoperatoren** für die verkürzte Schreibweise von Berechnungen mit **arithmetischen Operatoren** und anschließender **Wertzuweisung**

Arithmetische Operatoren

Zu den arithmetischen Operatoren gehören in C#:

- ✓ Die Operatoren für die Grundrechenarten ($(+)$, $(-)$, $(*)$, $(/)$)
- ✓ Der Modulo-Operator $(\%)$, der den Rest bei einer Division bestimmt

Diese Operatoren benötigen jeweils **zwei Operanden**, die mit dem Operator verknüpft werden. Sie werden daher als **binäre Operatoren** bezeichnet.

Die Grundrechenarten können auf Integer- und auf Gleitkommawerte angewendet werden. Bei der Anwendung der Operatoren gilt wie in der Mathematik die Regel „Punktrechnung geht vor Strichrechnung“. Das bedeutet, dass die Operatoren $(*)$ und $(/)$ eine höhere Priorität besitzen als die Operatoren $(+)$ und $(-)$.

Neben der Priorität ist die sogenannte **Assoziativität (Bindung)** für die Auswertung entscheidend. Sie legt fest, in welcher Richtung die Auswertung erfolgt. Alle Operatoren mit der gleichen Priorität besitzen auch die gleiche Assoziativität. Die arithmetischen (binären) Operatoren besitzen eine Links-Assoziativität, sie sind **links-bindend**.

Beispiel

<ul style="list-style-type: none"> ① Nach der Priorität wird zunächst der Ausdruck $4 * 5$ ausgewertet. ② Alle Operatoren haben jetzt die gleiche Priorität und sind links-bindend. ③ Die Auswertung erfolgt daher von links nach rechts. (Die Klammern sind hier nicht erforderlich, sondern dienen nur zur Veranschaulichung.) 	$ \begin{aligned} & 3 + 4 * 5 + 6 + 7 \\ & \textcircled{1} = 3 + 20 + 6 + 7 \\ & \textcircled{2} = (3 + 20) + 6 + 7 \\ & \textcircled{3} = (23 + 6) + 7 \\ & = 29 + 7 \\ & = 36 \end{aligned} $
--	--

Auswertung eines Ausdrucks

Beispiele für den Einsatz der Grundrechenarten: Operators.sln

Den Beispielen liegen die folgenden beiden Variablen-deklarationen zugrunde:

```
int i;
double x;
```

- ✓ Geklammerte Ausdrücke werden immer zuerst ausgewertet und „Punktrechnung geht vor Strichrechnung“.


```
i = 3 + 4 * 5;           // Die Variable i erhält hier den Wert 23,
                           // da der Ausdruck 4 * 5 zuerst ausgewertet wird.
```

```
i = (3 + 4) * 5;         // Dieser Ausdruck weist der Variablen i den Wert 35 zu,
                           // da geklammerte Ausdrücke zuerst ausgewertet werden.
```
- ✓ Die Auswertung eines Operators richtet sich nach den Datentypen der verwendeten Operatoren: Die Division mit Integer-Werten liefert als Ergebnis ebenfalls einen Wert vom Typ int. Kommastellen entfallen:


```
i = 13 / 5;            // Die beiden integer-Werte werden ohne Rest dividiert,
                           // i erhält den Wert 2
```

```
x = 13 / 5;            // Auch bei der Wertzuweisung an eine Variable vom Typ
                           // double entsteht zunächst der integer-Wert.
                           // x erhält den Wert 2.0
```
- ✓ Die Division zweier Zahlen vom Typ double liefert auch ein Ergebnis vom Typ double.

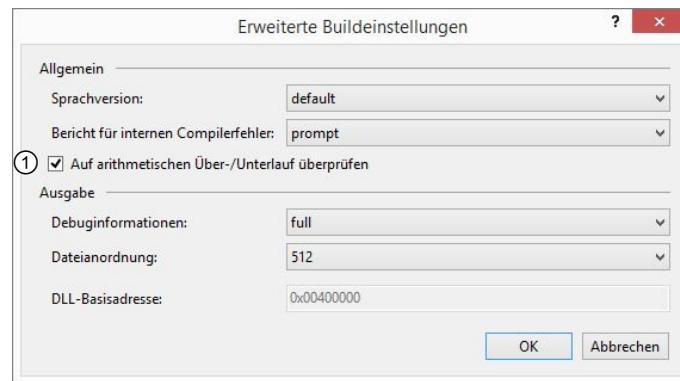

```
x = 13.0 / 5.0;        // x erhält den Wert 2.6
```
- ✓ Der Modulo-Operator % bestimmt den Rest bei einer Division von Integer- oder Gleitkommazahlen, wobei die Teilung selbst ganzzahlig erfolgt.


```
i = 13 % 5;            // i erhält den Wert 3, denn 13 / 5 = 2, Rest = 3
x = 13.0 % 5.0;          // x erhält den Wert 3.0
x = 5 % 2.2;             // x erhält den Wert 0.6, denn 5 / 2.2 = 2, Rest = 0.6
```

Überlauf (Overflow)

Bei arithmetischen Operationen und bei Typkonvertierungen von Ganzzahl-Datentypen kann das Ergebnis die Grenzen des Wertebereichs des Ergebnisdatentyps überschreiten (Überlauf; engl. Overflow).

Sie können selbst bestimmen, ob im Fall eines Überlaufs eine Fehlermeldung generiert oder der Überlauf ignoriert werden soll. Ignorieren Sie einen Überlauf, geht die Information verloren, die nicht mehr von dem verwendeten Datentyp erfasst wird. Das Verhalten Ihres Programms in Bezug auf arithmetische Überläufe können Sie im Dialogfenster *Erweiterte Buildeinstellungen* einstellen.



Diese Eigenschaften erreichen Sie, indem Sie über das Kontextmenü der Projektdatei im Projektmappen-Explorer oder mit dem Menü *Projekt - [Projektname]-Eigenschaften* die Projekteigenschaften öffnen. Anschließend wechseln Sie in das Register *Build* (früher *Erstellen*) und klicken in diesem Register auf die Schaltfläche *Erweitert*.

Standardmäßig erfolgt keine Überlaufprüfung. Das entsprechende Kontrollfeld ist deaktiviert ①. Dadurch entsteht die Gefahr, dass Überlauffehler unentdeckt bleiben und zu falschen Ergebnissen führen. Wenn Sie das Kontrollfeld aktivieren, erzeugt das Programm bei einer Über- oder Unterschreitung von Zahlengrenzen eine sogenannte Ausnahme (`OverflowException`), durch die das Programm abgebrochen und eine Fehlermeldung ausgegeben wird. Im Kapitel über Fehlerbehandlung erfahren Sie, wie Sie Ausnahmen abfangen und verarbeiten können.

Es ist außerdem möglich, das Verhalten bei Überläufen direkt im Programmcode zu steuern. Diesem Zweck dienen die Schlüsselwörter `checked` (Überlaufprüfung) und `unchecked` (keine Überlaufprüfung), die Sie direkt vor den Ausdruck oder vor den Anweisungsblock schreiben können, für den sie gültig sein sollen.

Beispiel: *Check.sln*

Das Programm soll die Summe von zwei Zahlen berechnen und auf dem Bildschirm ausgeben. Da der Datentyp `sbyte` nur Werte im Bereich von -128 bis 127 annehmen kann, wird dieser Bereich bei der Addition überschritten. Je nachdem, wie die Überlaufprüfung eingestellt ist, kommt es zu einem unterschiedlichen Verhalten des Programms.

```
class Program
{
    static void Main(string[] args)
    {
        ① sbyte a = 127, b = 12, c;
        ② c = checked((sbyte)(a + b));
        ③ Console.WriteLine(c);
    }
}
```

- ① Drei Variablen vom Ganzzahl-Datentyp `sbyte` werden deklariert. Die Variablen `a` und `b` werden mit dem Wert 127 bzw. 12 initialisiert.

- ② Die Addition der beiden Werte vom Datentyp `sbyte` liefert einen Wert des Datentyps `int` zurück. Durch eine explizite Konvertierung (`sbyte`) wird der Wert in den Typ `sbyte` konvertiert. Durch die Angabe des Schlüsselworts `checked` wird beim Überlauf eine Fehlermeldung generiert. Wenn Sie stattdessen `unchecked` eingeben, wird keine Prüfung vorgenommen und das Ergebnis `-117` ausgegeben. Der Wert entsteht dadurch, dass der Wertebereich von `sbyte` von `-128` bis `+127` reicht. Der Überlaufbetrag von $(12 - 1=11)$ wird zur unteren Grenze addiert ($-128 + 11=-117$).
- ③ Das Ergebnis wird ausgegeben.

Vorzeichenoperatoren

Bei negativen Zahlen kennen Sie die Kennzeichnung mit einem vorangestellten `-`. Bei positiven Zahlen können Sie ein `+` voranstellen, das aber entfallen kann. Diese Vorzeichenoperatoren werden als **unäre Operatoren** bezeichnet, denn sie besitzen nur einen Operanden.

- ✓ Die unären Vorzeichenoperatoren können nicht nur auf Zahlen, sondern auch auf Variablen bzw. auf ganze Ausdrücke angewendet werden.
- ✓ Die unären Vorzeichenoperatoren besitzen eine höhere Priorität als die binären (arithmetischen) Operatoren.
- ✓ Die Auswertung der unären Vorzeichenoperatoren erfolgt von rechts nach links. Das heißt, unäre Vorzeichenoperatoren besitzen eine **Rechts-Assoziativität** (Rechts-Bindung).
- ✓ Die folgenden Beispiele zeigen die Verwendung der unären Vorzeichenoperatoren:

Beispiel Vorzeichenoperatoren anwenden: *Sign.sln*

```
int i = 3;
int result = 0;

① result = -i;           // result erhält den Wert -3
② result = -(i - 5);    // result erhält den Wert 2
③ result = -(-3);      // result erhält den Wert 3
```

- ① `result` erhält den Wert `-3`.
- ② Zuerst wird die Klammer ausgewertet und liefert das Ergebnis `-2`. Anschließend wird durch den Vorzeichenoperator `-` das Vorzeichen gewechselt; `result` erhält den Wert `2`.
- ③ `result` erhält den Wert `3` (doppelte Vorzeichenumkehrung).

Verkettungsoperatoren

Mehrere Zeichenketten können Sie zu einer Zeichenkette zusammenfügen. Dazu verwenden Sie den Verkettungsoperator `+`.

```
string s1, s2;
s1 = "Hal" + "lo"; // s1 erhält den Text "Hallo"
s2 = 123 + " km"; // s2 erhält den Text "123 km"
```

Verkettungsoperatoren anwenden (*JoinStrings.sln*)

Wie das zweite Beispiel zeigt, wird eine Zahl bei der Verkettung mit einer Zeichenkette automatisch in einen `string` umgewandelt.

Inkrementierung und Dekrementierung

Operator	Operandentyp	Beispiel	Beschreibung
<code>++</code>	numerische Variablen	<code>int i = 1;</code> <code>i++; // oder ++i;</code>	Die numerische Variable wird um den Wert 1 erhöht (inkrementiert).
<code>--</code>	numerische Variablen	<code>int i = 1;</code> <code>i--; // oder --i;</code>	Die numerische Variable wird um den Wert 1 vermindert (dekrementiert).

Bei den Operatoren `++` und `--` wird zwischen Postfix- und Präfix-Notation unterschieden. Die Operatoren stehen entsprechend hinter bzw. vor der Variablen. Bei der Postfix-Notation wird die Variable nach dem Auswerten des Ausdrucks inkrementiert bzw. dekrementiert, bei der Präfix-Notation davor.



```

int i=5, j=5;
i++;           // entspricht ++i i erhält den Wert 6
j--;           // entspricht --j j erhält den Wert 4
① int result1 = 2 * ++i; // result1 erhält den Wert 14
② int result2 = 2 * j++; // result2 erhält den Wert 8
    
```

Inkrementierung und Dekrementierung mit Prä- und Postfix-Notation (PrefixPostfix.sln)

- ① Die Berechnung erfolgt in zwei Schritten:

1. Schritt: `i = i + 1;` // i erhält den Wert 7
2. Schritt: `result1 = 2 * i;` // result1 erhält den Wert 14

- ② Die Berechnung erfolgt auch hier in zwei Schritten:

1. Schritt: `result2 = 2 + j;` // result1 erhält den Wert 8
2. Schritt: `j = j + 1;` // j erhält den Wert 5

6.15 Logische Operatoren

Vergleichsoperatoren

C# besitzt für Wahrheitswerte den Datentyp `bool`.

```

bool isResultValid = true;
bool isValueOutOfRange = false;
    
```

Mithilfe von **Vergleichsoperatoren** können Sie Ausdrücke formulieren, die einen Wert vom Typ `bool` liefern. Diese Ausdrücke lassen sich sprachlich entsprechend den folgenden Beispielen wie eine Behauptung formulieren, die mit wahr (`true`) oder falsch (`false`) beurteilt werden kann:

- ✓ Ein Wert ist mit einem anderen Wert **identisch**.
- ✓ Ein Wert ist **größer als** ein anderer Wert.
- ✓ Der Wert eines Ausdrucks ist **kleiner als** der Wert eines anderen Ausdrucks.

In C# können alle mathematischen Vergleiche mithilfe einfacher Zeichen dargestellt werden. Vergleichsoperatoren können auf fast alle Datentypen angewendet werden. Ist ein Ausdruck nicht wahr, wird `false` geliefert, sonst `true`. Es werden 6 Vergleichsoperatoren unterschieden:

<code>==</code>	Überprüft zwei Ausdrücke auf Gleichheit (alle primitiven Datentypen)
<code>!=</code>	Überprüft zwei Ausdrücke auf Ungleichheit (alle primitiven Datentypen)
<code>></code>	Liefert <code>true</code> , wenn der erste Ausdruck größer als der zweite ist (alle außer <code>bool</code>)
<code><</code>	Liefert <code>true</code> , wenn der erste Operand kleiner als der zweite ist (alle außer <code>bool</code>)
<code>>=</code>	Liefert <code>true</code> , wenn der erste Operand größer oder gleich dem zweiten ist (alle außer <code>bool</code>)
<code><=</code>	Liefert <code>true</code> , wenn der erste Ausdruck kleiner oder gleich dem zweiten ist (alle außer <code>bool</code>)

Beispiel: *CompareOperators.sln*

```
int i = 10;
int j = 15;
bool b;
b = i > j;           // b erhält den Wert false
b = i != j;          // b erhält den Wert true
b = i == j;          // b erhält den Wert false
```

Verknüpfungsoperatoren

Verknüpfungsoperatoren dienen dazu, zwei Ausdrücke, die einen logischen Rückgabewert (`true`, `false`) liefern, entsprechend einer Vorschrift miteinander zu verknüpfen. Je nach Verknüpfungsvorschrift ist das Ergebnis ebenfalls `true` oder `false`. Der Operator `!` zur Negierung eines Ausdrucks besitzt nur **einen** Operanden.

Operator	Bedeutung
<code>!</code>	Negation: Bei booleschen Datentypen wird die Negation des Wertes als Ergebnis zurückgeliefert. <code>true</code> wird zu <code>false</code> und umgekehrt. Bei Ganzzahl-Datentypen wird eine logische Negation durchgeführt. Aus jeder 0 wird eine 1 und umgekehrt.
<code>&</code>	Und-Verknüpfung: Bei booleschen Datentypen ergibt die Verknüpfung mit dem Operator <code>&</code> nur dann <code>true</code> , wenn beide Ausdrücke <code>true</code> liefern. Bei Ganzzahl-Datentypen wird eine bitweise Und-Verknüpfung durchgeführt. Als Ergebnis ergibt sich 1, wenn beide Zahlen an derselben Stelle eine 1 besitzen.
<code>&&</code>	Bei booleschen Datentypen einsetzbar: Wie <code>&</code> , jedoch wird der zweite Ausdruck nur ausgewertet, wenn der erste Ausdruck <code>true</code> liefert.
<code> </code>	Oder-Verknüpfung: Bei booleschen Datentypen ergibt die Verknüpfung mit dem Operator <code> </code> den Wert <code>true</code> , wenn mindestens einer der Ausdrücke <code>true</code> liefert. Bei Ganzzahl-Datentypen wird eine bitweise Oder-Verknüpfung durchgeführt. Dabei ergibt sich 1, wenn mindestens eine der beiden Zahlen an der gleichen Stelle eine 1 besitzt.
<code> </code>	Bei booleschen Datentypen einsetzbar: Wie <code> </code> , jedoch wird der zweite Ausdruck nur ausgewertet, wenn der erste Ausdruck <code>false</code> liefert.
<code>^</code>	Exklusiv-Oder-Verknüpfung: Bei booleschen Datentypen ergibt die Verknüpfung mit dem Operator <code>^</code> den Wert <code>true</code> , wenn genau einer der Ausdrücke <code>true</code> liefert. Bei Ganzzahl-Datentypen wird eine bitweise exklusive Oder-Verknüpfung durchgeführt. Dabei ergibt sich 1, wenn beide Zahlen an derselben Stelle einen unterschiedlichen Wert besitzen.

Der Operator `!` ist wie der Vorzeichenoperator rechts-bindend. Die Auswertung erfolgt von rechts nach links. Die anderen logischen Operatoren sind links-bindend, d. h., die Auswertung erfolgt von links nach rechts.

Außerdem existieren in C# noch Verschiebeoperatoren `<<` und `>>`, die in diesem Buch nicht näher erläutert werden. Sie dienen dazu, eine Zahl bitweise um eine bestimmte Anzahl von Stellen nach links (`<<`) bzw. nach rechts (`>>`) zu schieben.

Beispiel: *LogicalOperators.sln*

```

int k = 0;
int i = 1;
bool b = false;

b = (k > i) & (k >= 0); // b erhält den Wert false, da k nicht größer als i
b = (k > i) | (i > k); // b erhält den Wert true, da i größer als k
b = !b; // b erhält den Wert false, da b zuvor true war
b = (k > i) && (k >= 0); // Da k > i nicht zutrifft, kann auch der gesamte
                           // Ausdruck nicht wahr sein. Der zweite Ausdruck
                           // k >= 0 wird daher nicht mehr ausgewertet.
b = (i > k) || (k > i); // k > i wird nicht ausgewertet, da bereits i > k
                           // den Wert true liefert

```

6.16 Zuweisungsoperatoren für eine verkürzte Schreibweise verwenden

Häufig treten Anweisungen auf, mit denen der Wert einer Variablen ausgelesen, verändert und anschließend wieder in derselben Variablen gespeichert wird.

variable = variable operator expression;
Häufige Anweisung

Für diese Anweisungen gibt es in C# verkürzte Schreibweisen:

Beispiel: *ReducedNotation.sln*

```

int i = 15;
int k = 3;
bool b = true;
string s = "Programm";
i += 5;           // ausführlich: i = i + 5;      -> i erhält den Wert 20
s += "ierung";   // ausführlich: s = s + "ierung";
                  // -> s erhält den Wert "Programmierung"
i += k;          // ausführlich: i = i + k;      -> i erhält den Wert 23
i -= 7;          // ausführlich: i = i - 7;      -> i erhält den Wert 16
i /= k;          // ausführlich: i = i / k;      -> i erhält den Wert 5
i *= 7;          // ausführlich: i = i * 7;      -> i erhält den Wert 35
i %= 8;          // ausführlich: i = i % 8;      -> i erhält den Wert 3
b ^= true;        // ausführlich: b = b ^ false; -> b erhält den Wert false
b |= true;        // ausführlich: b = b | true;   -> b erhält den Wert true
b &= true;        // ausführlich: b = b & true;   -> b erhält den Wert true

```

Auch für die Schiebeoperatoren << und >> kann die verkürzte Schreibweise mit dem Zuweisungsoperatoren <<= bzw. >>= verwendet werden.

6.17 Der Operator `nameof`

Der Operator `nameof` gibt **typsicher** den Namen von einem Parameter, Member oder Typ als String zurück – auch nach einer Refrakturierung. Zur Anwendung übergeben Sie dem Operator `nameof` einen Operanden und der Operator liefert typsicher den Namen des Operanden. Erlaubt als Operand sind:

- ✓ Typnamen
- ✓ Lokale Variablen
- ✓ Klassenmitglieder
- ✓ Namensräume
- ✓ Parameter
- ✓ Typparameter

Dabei wird von `nameof()` immer nur der relative Name geliefert. Übergeordnete Namensräume bzw. Typnamen bleiben unberücksichtigt:

Beispiel: *NameOfTest.cs*

```
int a = 42;
static readonly String b = "Die Antwort ist";
DateTime c = new DateTime();
static void Main(string[] args)
{
    bool d;
    Console.WriteLine(nameof(a));
    Console.WriteLine(nameof(b));
    Console.WriteLine(nameof(c));
    Console.WriteLine(nameof(d));
}
```

In diesem einfachen Beispiel erhalten Sie als Ergebnis die deklarierten Variablennamen `a`, `b`, `c` und `d`.

6.18 Übungen

Übung 1: Werte ein- und ausgeben

Übungsdatei: --

Ergebnisdatei: *InputConsole.sln*

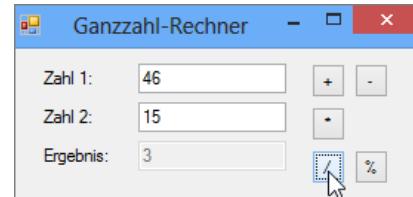
1. Fordern Sie in der Anwendung den Anwender dazu auf, eine ganze Zahl einzugeben, und speichern Sie den Wert in einer Variablen vom Typ `int`.
2. Lassen Sie auch einen `double`- und einen `string`-Wert mithilfe der Tastatur eingeben und speichern Sie die Werte ebenfalls in geeigneten Variablen.
3. Geben Sie die Werte der drei Variablen abschließend auf dem Bildschirm aus.

Übung 2: Ein einfacher Rechner für ganze Zahlen

Übungsdatei: --

Ergebnisdatei: *IntegerCalc.sln*

1. Erstellen Sie ein neues Projekt als Windows-Anwendung.
2. Platzieren Sie auf dem Formular:
 - ✓ drei TextBox-Steuerelemente (zwei zur Eingabe und eines zur Ausgabe der Werte),
 - ✓ drei Label-Steuerelemente zur Beschriftung,
 - ✓ fünf Button-Steuerelemente.
3. Setzen Sie die Eigenschaften `Enabled` und `ReadOnly` des Ergebnis-Ausgabefelds (TextBox-Steuerelement) auf den Wert `false`.
4. Erzeugen Sie durch Doppelklick auf die Schaltfläche das Gerüst für eine entsprechende Ereignisbehandlungsmethode. Mit der Ereignisbehandlungsmethode sollen zwei eingegebene Zahlen (`Zahl1` und `Zahl2`) addiert werden.
5. Lesen Sie die Eigenschaft `Text` des ersten Felds aus und speichern Sie die Zeichenkette in einer `string`-Variablen.



Zum Auslesen der Eigenschaft `Text` schreiben Sie hinter den Namen des TextBox-Steuerelements (hier `txtZahl1`) durch einen Punkt getrennt den Eigenschaftsnamen `Text` und weisen diesen Ausdruck einer Variablen vom Typ `string` zu.

```
string txt;
txt = txtZahl1.Text;
...
```

6. Wandeln Sie die Eingabe in den gewünschten Datentyp (`int`) um.
7. Speichern Sie auch die zweite eingegebene Zahl in einer geeigneten Variablen.
8. Berechnen Sie das Ergebnis in einer Variablen vom Typ `int`.
9. Wandeln Sie das Ergebnis wieder in einen `String` um und weisen Sie diesen der Eigenschaft `Text` des Ausgabefeldes zu.
10. Programmieren Sie auf die gleiche Weise die Ereignisbehandlungsmethoden für die anderen Schaltflächen. Neben der bereits programmierten Addition sollen Subtraktion, Multiplikation, Division und Modulo-Berechnung (Divisionsrest) möglich sein.

```
...
txtErgebnis.Text = txt;
```

7 Kontrollstrukturen

7.1 Was Kontrollstrukturen einsetzen

Kontrollstrukturen im Überblick

Die Programme, die Sie bisher kennengelernt haben, bestanden aus einer Folge von Anweisungen, die einmal sequenziell (der Reihe nach) abgearbeitet wurden. Oft ist es erforderlich, dass Programmteile mehrmals oder gar nicht ausgeführt werden. Die C#-Sprachelemente, mit denen der Programmablauf gesteuert werden kann, werden **Kontrollstrukturen** genannt. Die Entscheidung, nach welchen Kriterien der Ablauf gesteuert wird, wird in **Bedingungen** formuliert.

Es werden zwei Gruppen von Kontrollstrukturen unterschieden:

- ✓ **Verzweigungen:**
Es werden alternative Programmteile angeboten, in die – abhängig von einer Bedingung – beim Programmablauf verzweigt wird.
- ✓ **Schleifen (engl. loops):**
Ein Programmteil kann – abhängig von einer Bedingung – mehrmals durchlaufen werden.

Verzweigungen

Zwei Grundformen der Verzweigungen sind in C# enthalten:

- ✓ **Bedingte Auswahl:**
Je nachdem, ob eine Bedingung erfüllt ist oder nicht, wird ein Programmteil ausgeführt oder übersprungen bzw. ein alternativer Programmteil ausgeführt.
- ✓ **Fallauswahl:**
Es wird ein Ausdruck ausgewertet. Je nachdem, welchem Wert der Ausdruck entspricht, verzweigt das Programm fallweise (**case**) in einen entsprechenden Programmteil. Eine **Fallauswahl** wird auch **mehrseitige Auswahl** genannt, da sie mehrere Alternativen bereitstellen kann.

Schleifen (Wiederholungen)

Über eine **Schleife**, auch **Wiederholung** oder **Iteration** genannt, können Anweisungen und Anweisungsblöcke mehrfach ausgeführt werden.

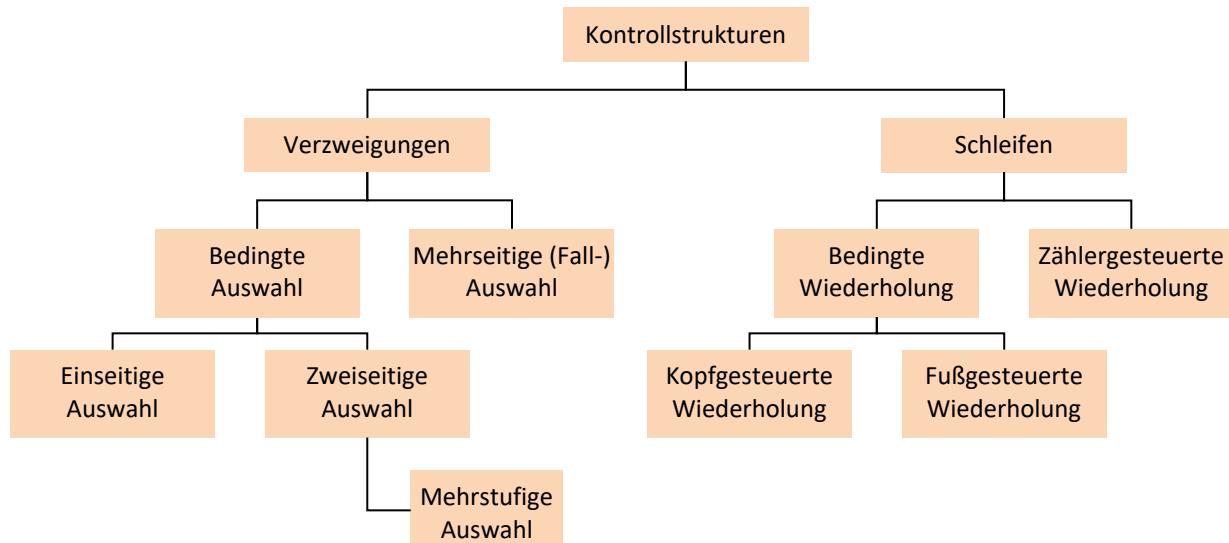
Zwei Grundformen von Schleifen-Strukturen sind in C# enthalten:

- ✓ **Bedingte Wiederholung:**
Die Wiederholung des in der Schleife eingeschlossenen Programmteils ist an eine Bedingung geknüpft. Die Bedingung kann **vor** oder **nach** der Ausführung des eingeschlossenen Programmteils erfolgen:
Bei der **kopfgesteuerten Wiederholung** erfolgt die Überprüfung der Bedingung **vor** der Ausführung des eingeschlossenen Programmteils. Solange eine Bedingung erfüllt ist, wird der im Anweisungsblock der Struktur eingeschlossene Programmteil ausgeführt. Anschließend wird das Programm hinter der Schleifenstruktur fortgesetzt.
Bei der **fußgesteuerten Wiederholung** wird zunächst der eingeschlossene Programmteil ausgeführt. **Anschließend** wird die Bedingung geprüft. Ist die Bedingung erfüllt, wird der Programmteil erneut ausgeführt. Andernfalls wird das Programm hinter der Schleifenstruktur fortgesetzt.

✓ **Zählergesteuerte Wiederholung:**

Ein Zähler bestimmt die Anzahl der Wiederholungen. Die Überprüfung, ob der Zähler den gewünschten Endwert erreicht hat, erfolgt **vor** der Ausführung des eingeschlossenen Programmteils.

Übersicht über die Kontrollstrukturen in C#

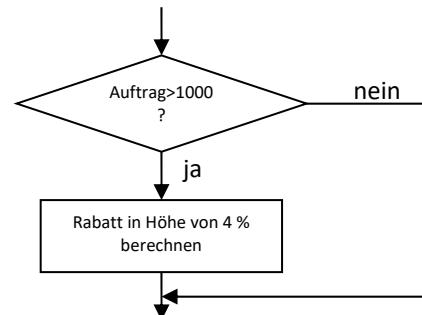


7.2 Einseitige Auswahl

Bei vielen Problemstellungen ist die Verarbeitung von Anweisungen von **Bedingungen** abhängig. Nur wenn die Bedingung erfüllt ist, wird die betreffende Anweisung (bzw. der Anweisungsblock) ausgeführt. Andernfalls wird die Anweisung übersprungen. Für die Steuerung eines solchen Programmablaufs stellt C# die **einseitige Auswahl** (**if**-Anweisung) zur Verfügung.

Beispiel für die Verwendung einer einseitigen Auswahl

- ✓ Wenn (**if**) ein Kunde einen Auftrag über 1000,- EUR erteilt, dann erhält er 4 % Rabatt. Bei Aufträgen bis 1000,- EUR wird kein Rabatt gewährt, d. h., die Berechnung des Rabatts wird nicht ausgeführt.
- ✓ Die Überprüfung der Bedingung kann nur die beiden Ergebnisse "ja" oder "nein" ergeben, in C# entsprechend **true** oder **false**.



Syntax der einseitigen Auswahl (**if**-Anweisung)

- ✓ Die einseitige Auswahl beginnt mit dem Schlüsselwort **if**.
- ✓ Dahinter wird – eingeschlossen in runde Klammern **()** – eine Bedingung (**condition**) formuliert. Die Bedingung ist ein Ausdruck (**expression**) und liefert als Ergebnis einen Wert vom Typ **bool** zurück.
- ✓ Anschließend folgt die Anweisung **①**, die ausgeführt wird, wenn die Auswertung der Bedingung den Wert **true** (wahr) ergibt.
- ✓ Die Anweisung **①** kann aus einer einzelnen Anweisung oder einem Anweisungsblock in geschweiften Klammern **{ }** bestehen.
- ✓ Liefert die Bedingung den Wert **false** (falsch), wird die Anweisung **①** bzw. der Anweisungsblock übersprungen.

if (**condition**)
 statement; ①

Beachten Sie, dass Anweisungen ein abschließendes Semikolon enthalten.

Schnell können Sie eine einseitige if-Anweisung mithilfe eines Codeausschnitts schreiben. Geben Sie dazu if ein und betätigen Sie zweimal die -Taste. In den Platzhalter ① tragen Sie die Bedingung ein.

```
if (true)
{
    ①
}
```

Beispiel: *Discount.sln*

Für einen Rechnungsbetrag (`invoiceAmount`) soll abhängig von seiner Höhe ein Rabatt berechnet werden. Anhand des Wertes der Variablen `invoiceAmount` wird entschieden, ob ein Rabatt gewährt wird. Dieser wird ggf. berechnet und vom Rechnungsbetrag subtrahiert. Der neue Rechnungsbetrag wird (abzüglich des Rabatts) ausgegeben. Währungsangaben werden hier nicht berücksichtigt.

```
...
static void Main(string[] args)
{
    double invoiceAmount = 0.0; // Rechnungsbetrag
    string txt;
    Console.WriteLine("Bitte geben Sie den Rechnungsbetrag ein: ");
    ①
    ②
    invoiceAmount = Convert.ToDouble(txt);
    ③
    if (invoiceAmount > 1000)
    {
        ④
        invoiceAmount -= invoiceAmount * 0.04;
        Console.WriteLine("Es wird ein Rabatt gewährt.");
    }
    ⑤
    Console.WriteLine("Gesamtbetrag: " + invoiceAmount);
}
...
```

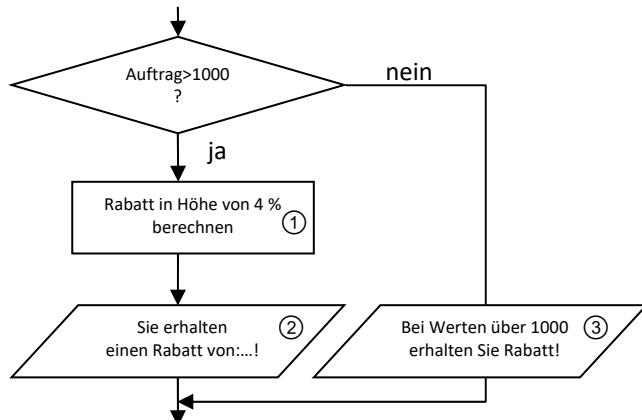
- ① Die eingegebene Zeichenkette wird eingelesen.
- ② Der String wird in eine Zahl vom Typ `double` konvertiert und einer Variablen zugewiesen.
- ③ Der Wert dieser Variablen wird mit dem Wert 1000 verglichen.
- ④ Ist der Rechnungsbetrag `invoiceAmount` größer als 1000, liefert die Bedingung den Wert `true`; der Rabatt wird berechnet und ein Text als Information ausgegeben.
- ⑤ Der Rechnungsbetrag wird ausgegeben. Wenn kein Rabatt gewährt wird, wird der Anweisungsblock ④ übersprungen und der Rechnungsbetrag unverändert ausgegeben.

7.3 Zweiseitige Auswahl

Die **zweiseitige Auswahl** ist dadurch gekennzeichnet, dass einer von **zwei** Anweisungsblöcken in Abhängigkeit von einer Bedingung ausgeführt wird. Bestimmte Anweisungen werden durchgeführt, falls die Bedingung erfüllt ist. Falls die Bedingung nicht erfüllt ist, werden andere Anweisungen ausgeführt.

Beispiel für den Einsatz einer zweiseitigen Auswahl

- ✓ Wenn (**if**) ein Kunde einen Auftrag über 1000,- EUR erteilt, dann erhält er 4 % Rabatt
①. Eine Meldung informiert darüber ②.
- ✓ Sonst (**else**), d. h. bei Aufträgen bis 1000,- EUR, wird kein Rabatt gewährt und eine entsprechende Meldung ③ ausgegeben.



Syntax der zweiseitigen Auswahl

- ✓ Nach dem Schlüsselwort **if** steht in runde Klammern **()** eingeschlossen die Bedingung ①.
- ✓ Anschließend folgt die Anweisung ② oder der Anweisungsblock, die bzw. der ausgeführt werden soll, wenn die Bedingung erfüllt ist.
- ✓ Nach **else** schließt sich die Anweisung ③ oder der Anweisungsblock an, die bzw. der ausgeführt wird, wenn die Bedingung in der **if**-Anweisung **nicht** zutrifft (Ausdruck liefert **false**). Die Anweisung ② bleibt unberücksichtigt.

```

if (condition) ①
  statement1 ②
else
  statement2 ③
  
```

Aus Platzgründen wird in diesem Buch immer eine kompakte Schreibweise der Anweisungen verwendet. Sie sollten allerdings zukünftig auch im Fall nur einer einzelnen Anweisung innerhalb der **if-else**-Anweisung diese immer in Anweisungsblöcke einschließen. Dadurch ist besser erkennbar, welche Anweisungen im **if**- oder **else**-Zweig ausgeführt werden, und bei Erweiterungen kann es nicht passieren, dass die Klammern versehentlich vergessen werden. Gerade Letzteres muss nicht zwangsläufig zu einem Fehler führen, da der Code trotzdem eine korrekte Struktur aufweisen kann, auch wenn seine Ausführung so nicht gewollt war.

```

if (condition)
{
  statement1
}
else
{
  statement2
}
  
```

Der Bedingungsoperator **:** :

Möchten Sie in Abhängigkeit von einer Bedingung einen bestimmten Wert zurückgeben, können Sie statt einer **if**-Anweisung einen Ausdruck mithilfe des Bedingungsoperators **? :** formulieren:

- ✓ Der Ausdruck beginnt mit der Formulierung der Bedingung (hier: **i < 5**), die einen logischen Wert liefert.
- ✓ Anschließend folgt ein Fragezeichen **?**.
- ✓ Ist die Bedingung erfüllt, wird der Ausdruck **expression1** (hier: **j + 6**) verwendet, andernfalls der Ausdruck **expression2** (hier: **j - 7**).
- ✓ Die beiden alternativen Ausdrücke werden durch einen Doppelpunkt **:** getrennt.

```
condition ? expression1 : expression2
```

```

int i = 3;
int j = 10;
int k;

k = i < 5 ? j + 6 : j - 7;

// dies entspricht:
if (i < 5)
  k = j + 6;
else
  k = j - 7;
  
```

Ein solcher Bedingungsausdruck liefert als Ergebnis immer einen Ausdruck und kann somit beispielsweise für eine Wertzuweisung (wie hier an Variable **k**) verwendet werden.

Beispiel: *Discount2.sln*

Für einen Rechnungsbetrag (`invoiceAmount`) soll abhängig von seiner Höhe ein Rabatt (`discountAmount`) berechnet werden. Wird ein Rabatt gewährt, wird dieser berechnet und der Rabattbetrag ausgegeben. Sofern kein Rabatt gewährt wird, erfolgt die Ausgabe einer entsprechenden Information.

```

...
static void Main(string[] args)
{
    ① double invoiceAmount = 0.0;                      // Rechnungsbetrag
    const double DISCOUNTRATE = 0.04;                  // Rabatt 4%, als Konstante
    double discountAmount = 0.0;                        // Rabattbetrag
    string txt;
    Console.WriteLine("Bitte geben Sie den Rechnungsbetrag ein: ");
    txt = Console.ReadLine();
    invoiceAmount = Convert.ToDouble(txt);
    ② if (invoiceAmount > 1000)
    {
        ③ discountAmount = invoiceAmount * DISCOUNTRATE;
        invoiceAmount -= discountAmount;
        Console.WriteLine("Sie erhalten einen Rabatt von: "
            + discountAmount);
    }
    ④ else
        Console.WriteLine("Bei Werten über 1000 erhalten Sie Rabatt!");
        Console.WriteLine("Gesamtbetrag: " + invoiceAmount);
    }
}
...

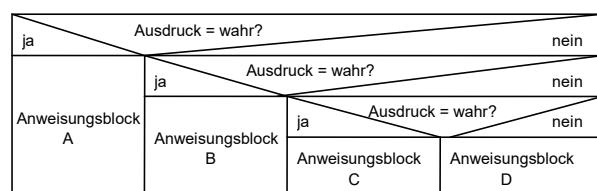
```

- ① Die Variablen und eine Konstante werden definiert und initialisiert. Beachten Sie, dass Konstanten per Konvention immer vollständig großgeschrieben werden.
- ② Es wird ein Rabatt bei einem Rechnungsbetrag über 1000 gewährt.
- ③ Ist die Bedingung erfüllt, wird der Rabatt berechnet und ausgegeben.
- ④ Ist der eingegebene Wert kleiner oder gleich 1000, wird kein Rabatt gewährt. Sie erhalten jedoch eine Meldung, ab wann Sie Rabatt erhalten.

7.4 Mehrstufige Auswahl

Was ist eine mehrstufige Auswahl?

Mehrstufige Auswahl bedeutet, dass einer von mehreren Anweisungsblöcken in Abhängigkeit von Bedingungen ausgeführt wird. **Wenn** (`if`) die erste Bedingung erfüllt ist, dann wird Anweisungsblock A ausgeführt, **sonst** wird die nächste Bedingung ausgewertet.



Anders als die einseitige und die zweiseitige Auswahl ist die mehrstufige Auswahl hier nicht als **Programmablaufplan** (PAP), sondern als **Struktogramm** dargestellt. Ein Struktogramm (auch Nassi-Schneidermann-Diagramm genannt) ist eine sehr kompakte Darstellungsform für Programmabläufe.

Geschachtelte **if**-Anweisungen

Eine mehrstufige Auswahl erreichen Sie in C#, indem Sie mehrere **if**- bzw. **if-else**-Anweisungen schachteln. Dazu schreiben Sie innerhalb eines Anweisungsblocks einer **if**- bzw. **if-else**-Anweisung eine weitere **if**- bzw. **if-else**-Anweisung.

Zu welcher if -Anweisung gehört die else -Alternative?	
<pre> if (condition) if (condition2) ② statement1 else ① statement2 </pre>	<pre> if (condition) ④ { if (condition2) statement1 } else ③ statement2 </pre>
Die mit else eingeleitete Alternative ① gehört zur jeweils vorherigen if -Anweisung ②.	Soll der else -Zweig ③ zu einer weiter entfernten if -Anweisung ④ gehören, müssen Sie dies durch die Verwendung von Anweisungsblöcken ⑤ sicherstellen.

Beispiel: *Discount3.cs*

Im Unterschied zu den beiden letzten Projekten soll hier bereits bei einem niedrigeren Rechnungsbetrag ein Rabatt in Höhe von 2 % gewährt werden.

```

...
static void Main(string[] args)
{
    double invoiceAmount = 0.0;          // Rechnungsbetrag
    string txt;
    Console.WriteLine("Bitte geben Sie den Rechnungsbetrag ein: ");
    txt = Console.ReadLine();
    invoiceAmount = Convert.ToDouble(txt);

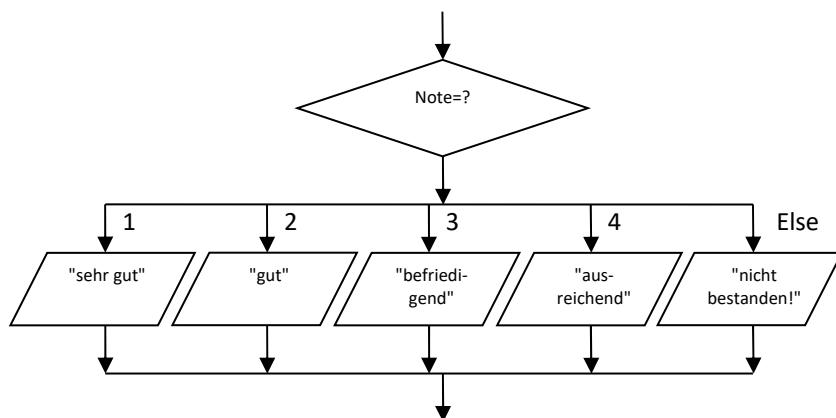
①    if (invoiceAmount > 1000)
    {
②        {
            invoiceAmount -= invoiceAmount * 0.04;
            Console.WriteLine("Sie erhalten 4 % Rabatt");
        }
    }
    else
    {
③        if (invoiceAmount > 500)
        {
④            invoiceAmount -= invoiceAmount * 0.02;
            Console.WriteLine("Sie erhalten 2 % Rabatt");
        }
    }
    else
⑤        Console.WriteLine("Bei Werten über 500 erhalten Sie Rabatt!");
    }
⑥    Console.WriteLine("Gesamtbetrag: " + invoiceAmount);
}
...
  
```

- ① Zunächst wird geprüft, ob der Rechnungsbetrag 1000 übersteigt. Wenn dies der Fall ist, werden 4 % Rabatt gewährt ②.

- ③ Andernfalls wird geprüft, ob der Rechnungsbetrag 500 übersteigt. Wenn dies der Fall ist, werden 2 % Rabatt gewährt ④.
- ⑤ In allen anderen Fällen erscheint ein Hinweis.
- ⑥ Der endgültige Rechnungsbetrag wird ausgegeben.

7.5 Mehrseitige Auswahl (Fallauswahl)

Bei `if`-Konstruktionen lassen sich immer nur **zwei** alternative Programmteile ausführen, da eine Bedingung als logischer Ausdruck nur die Werte `true` bzw. `false` ergeben kann. Bei einer **mehrseitigen Auswahl** testen Sie den Wert einer Variablen oder eines komplexen Ausdrucks. Diese Variable bzw. dieser Ausdruck wird als **Selektor** bezeichnet. In Abhängigkeit vom Wert des Selektors können fallweise (case) verschiedene Anweisungsblöcke ausgeführt werden. Die mehrseitige Auswahl wird daher auch Selektion genannt.



Eine solche mehrfache Auswahl können Sie ohne Weiteres mit `if` und passenden `else-if`-Zweigen umsetzen. In manchen Fällen eignet sich jedoch eine eigene Syntaxstruktur besser.

Syntax der Fallauswahl

Syntax für die `switch`-Anweisung

- ✓ Das Schlüsselwort `switch` leitet die **mehrseitige** Auswahl (Verzweigung) ein.
- ✓ Danach folgt der **Selektor**, der ein Ausdruck oder eine Variable sein kann. Der Wert des Selektors bestimmt, welche Anweisung als Nächstes ausgeführt wird. Der Selektor muss von einem ganzzahligen Datentyp oder vom Datentyp `string` sein.
- ✓ Auf den Selektor folgt ein Anweisungsblock ① mit sogenannten **case-Anweisungen**.
- ✓ Jede `case`-Anweisung beginnt mit dem Schlüsselwort `case`. Für jeden zu betrachtenden Wert, den der Selektor annehmen kann, wird eine `case`-Anweisung formuliert.
- ✓ Der Wert (`value`) muss ein konstanter Wert sein und einen dem Selektor entsprechenden Datentyp besitzen.

```

switch (selector)
{
    case value1:
        statement1
        break;
    case value2:
        statement2
        break;
    ...
    default:
        statement3 ③
        break;]
} ②
  
```

- ✓ Wird eine Bedingung erfüllt (der Wert des Selektors entspricht einem der aufgeführten Werte), dann wird das Programm mit dem Anweisungsblock unmittelbar hinter der entsprechenden case-Anweisung fortgesetzt. Mit der break-Anweisung wird die switch-Anweisung verlassen und das Programm hinter der switch-Anweisung ② fortgesetzt. Im Gegensatz zu vielen anderen Sprachen ist die break-Anweisung in C# zwingend. Genaugenommen muss dort zwingend eine Sprunganweisung stehen und damit kann alternativ auch eine andere Sprunganweisung notiert werden (etwa return, wenn man eine Methode verlassen will), aber der Regelfall ist break.
- ✓ Wird keine der Bedingungen erfüllt, wird der Anweisungsblock hinter der default-Anweisung ausgeführt ③, sofern diese Anweisung vorhanden ist. Auch hier ist die break-Anweisung oder eine andere Sprunganweisung in C# zwingend.

Verwenden Sie eine default-Anweisung, um sicherzustellen, dass immer mindestens einer der Anweisungsblöcke abgearbeitet wird.



Beispiel: Zahlen.sln

```

...
    static void Main(string[] args)
    {
        int zahl = 0;
        string txt;
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3 ein: ");
        ①
        txt = Console.ReadLine();
        zahl = Convert.ToInt32(txt);
        ②
        switch (zahl)
        {
            ③
            case 1: Console.WriteLine("Eins");
            ④
            break;
            ⑤
            case 2: Console.WriteLine("Zwei");
            ⑥
            break;
            ⑦
            case 3: Console.WriteLine("Drei");
            ⑧
            break;
            ⑨
            default:
                Console.WriteLine("ungültige Zahl");
                ⑩
                break;
        }
        ⑪
        Console.WriteLine("Eingegeben wurde: " + zahl);
    }
...

```

- ① Der eingegebene Text txt wird in den Datentyp int konvertiert.
- ② Der Wert der Variablen zahl wird als Selektor der switch-Anweisung verwendet.
- ③ Entspricht der Wert des Selektors dem nach case angegebenen Wert, dann wird der Wert als Text ausgegeben.
- ④ Mit der anschließenden break-Anweisung wird die switch-Anweisung beendet und die Anweisung ⑪ wird ausgeführt.
- ⑤ Falls keiner der Werte der case-Anweisungen dem Wert des Selektors entspricht, werden die Anweisungen nach default ausgeführt.

Werte Zusammenfassen: *Monate.sln*

Möglicherweise möchten Sie nicht für jeden einzelnen Fall (Wert) spezielle Anweisungen schreiben, sondern bei mehreren Werten sollen dieselben Anweisungen ausgeführt werden. Dazu schreiben Sie für diese Fälle die entsprechenden case-Anweisungen direkt untereinander. Für jeden Fall ist eine case-Anweisung anzugeben. Erst der letzte dieser Fälle enthält die auszuführenden Anweisungen.

```
...
    static void Main(string[] args)
    {
        string name = "";
        Console.Write("Geben Sie einen Monatsnamen ein: ");
        name = Console.ReadLine();
        ① switch (name)
        {
            ② case "Januar":
            ③ case "Februar":
            ④ case "März":
                Console.WriteLine("1. Quartal");
                break;
            ⑤ case "April":
                ...
            ⑥ default:
                Console.WriteLine("ungültiger Monatsname");
                break;
        }
    }
}
```

- ① Die string-Variable name, die den eingegebenen Monatsnamen enthält, wird als Selektor genutzt.
- ② Die case-Anweisung mit dem Wert "Januar" enthält keine Anweisung (auch keine break-Anweisung). Der Fall wird automatisch zum nächsten Fall weitergeleitet. Man spricht auch von "Durchfallen zum nächsten Fall".
- ③ Auch der Fall "Februar" fällt durch (zum Fall "März" ④).
- ④ Der Fall "März" enthält eine Anweisung.
- ⑤ Diese Anweisung wird für alle drei Fälle ("Januar", "Februar" und "März") ausgeführt.
- ⑥ Die break-Anweisung beendet den case-Zweig und damit die switch-Anweisung.

Die break-Anweisung darf nur dann entfallen, wenn der in der case-Anweisung beschriebene Fall durchfallen soll. Die case-Anweisung darf dann **keine** Anweisungen besitzen. Andernfalls erhalten Sie eine entsprechende Fehlermeldung.

7.6 Schleifen (Wiederholungen)

Schleifen verwenden

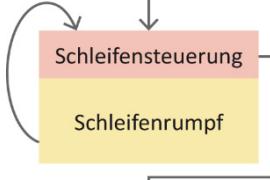
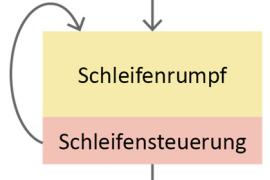
Um einen bestimmten Programmteil mehrfach auszuführen bzw. zu wiederholen, werden Schleifen (oft **Iterationen** genannt) verwendet: Sie können variabel (auch erst zur Laufzeit des Programms) festlegen, wie oft oder bis zum Eintreffen welcher Bedingung die Wiederholung durchlaufen werden soll.

Durch die Verwendung von Schleifen können Sie denselben Programmcode sehr kompakt mehrfach ausführen.

Aufbau einer Schleife

Eine Schleife besteht aus der **Schleifensteuerung** und dem **Schleifenrumpf**. Der Schleifenrumpf umfasst den Programmteil, der wiederholt werden soll. Die Schleifensteuerung legt fest, wie oft die Anweisungen im Schleifenrumpf wiederholt werden sollen bzw. nach welchen Kriterien entschieden wird, ob eine Wiederholung erfolgen soll.

Es werden zwei Steuerungsarten unterschieden:

Kopfgesteuerte Schleife	Fußgesteuerte Schleife
 <ul style="list-style-type: none"> ✓ Die Prüfung, ob die Anweisungen im Schleifenrumpf ausgeführt werden sollen, erfolgt gleich zu Beginn. ✓ Ist das Kriterium erfüllt, wird der Schleifenrumpf durchlaufen und anschließend erfolgt eine erneute Prüfung. ✓ Falls das Kriterium zu Beginn bereits nicht erfüllt ist, wird der Schleifenrumpf gar nicht ausgeführt. 	 <ul style="list-style-type: none"> ✓ Zuerst werden die Anweisungen im Schleifenrumpf ausgeführt. ✓ Dann erfolgt die Prüfung, ob ein weiterer Durchlauf erfolgen soll. ✓ Der Schleifenrumpf wird also immer mindestens einmal ausgeführt.

7.7 Kopfgesteuerte `while`-Anweisung

Bei der `while`-Anweisung handelt es sich um eine **kopfgesteuerte** Schleifenstruktur. Die Ausführung der Anweisungen im Schleifenrumpf ist von der Gültigkeit einer Bedingung abhängig, die gleich zu **Beginn** der Anweisung überprüft wird. **Solange** (`while`) die Bedingung erfüllt ist, werden die folgenden Anweisungen (Schleifenrumpf) ausgeführt. Anschließend folgt der nächste Durchlauf der Schleife und die Bedingung wird erneut geprüft. Ist die Bedingung **nicht erfüllt**, wird das Programm hinter der `while`-Anweisung fortgesetzt.

Beispiel

Solange eine Zahl kleiner als 100 ist, soll zu dieser Zahl der Wert 10 addiert werden: Es wird geprüft, ob die Zahl kleiner als 100 ist. Ist die Zahl kleiner als 100, wird zu der Zahl wieder der Wert 10 addiert. Anschließend erfolgt erneut die Prüfung der Bedingung.

Syntax der `while`-Anweisung

- ✓ Das Schlüsselwort `while` leitet die Anweisung ein.
- ✓ Es folgt in runden Klammern `()` die Formulierung der Bedingung, die einen Wert vom Typ `bool` liefern muss.
- ✓ Anschließend folgt der Schleifenrumpf mit der Anweisung oder einem Anweisungsblock (statement).

```
while (condition)
    statement;
```

- ✓ Ist die Bedingung erfüllt (`true`), wird der Anweisungsblock ausgeführt. Anschließend erfolgt eine erneute Prüfung der Bedingung. Solange die Bedingung erfüllt ist, wird die Ausführung der Anweisung(en) wiederholt.
- ✓ Ist die Bedingung nicht erfüllt (`false`), wird der Schleifenkörper übersprungen und das Programm nach der `while`-Anweisung fortgesetzt.



Achten Sie bei der Arbeit mit Schleifen immer darauf, dass Sie das Abbruchkriterium so festlegen, dass es **mit Sicherheit eintritt**. Andernfalls wird der Schleifenkörper unendlich oft ausgeführt (**Endlosschleife**) oder so lange, bis ein Fehler bei der Zuweisung entsteht. Dies kann z. B. der Fall sein, wenn Sie zu einer Zahl einen konstanten Wert addieren. Nach einer entsprechend großen Anzahl von Schleifendurchläufen wird der Wertebereich dieser Zahl überschritten.

Beispiel: *ControlWhile.sln*

Das folgende Programm gibt die Zahlen 1, 3, 5, 7, 9 aus. Dazu wird beim Startwert 1 begonnen und in jedem Schleifendurchlauf der Wert 2 addiert, solange der Wert der Variablen `i` (die Zählvariable) kleiner als 10 ist.

```
static void Main(string[] args)
{
    ① int i = 1;
    ② while (i < 10)
    {
        ④ Console.WriteLine(i);
        ⑤ i += 1;
    }
}
```



- ① Der Startwert der Schleife wird mit 1 festgelegt.
- ② Die Bedingung der Schleife legt fest, dass der Schleifenrumpf ③ ausgeführt wird, solange der Wert der Variablen `i` kleiner als 10 ist.
- ④ Der Wert der Variablen `i` wird ausgegeben.
- ⑤ Die Zählvariable `i` wird um 1 erhöht, anschließend wird wieder Schritt ② ausgeführt.

Beachten Sie, dass die Namen von Zählvariablen bei zählergesteuerten Schleifen im Grunde frei zu wählen sind (im Rahmen der verbindlichen Regeln für die Syntax). Es hat sich aber etabliert, dass die Zählvariable einer Schleife **immer i** genannt wird. Wenn Schleifen verschachtelt werden, bekommt die Zählvariable der inneren Schleife den Bezeichner `j`, eine weitere Schleife darin den Bezeichner `k` usw. Damit ist es für erfahrene Programmierer immer sofort klar, ob es sich um eine äußere Schleife oder eine innere Schleife handelt. Ebenso ist sofort und ohne weitere Erläuterungen ersichtlich, dass etwa `i` eine Zählvariable ist. Ebenso ist es Konvention, dass Zählvariablen mit dem Wert 0 initialisiert und immer um den Wert 1 erhöht werden.

Solche Konventionen tragen erheblich zu einer besseren Softwarequalität bei und sollten nur in Ausnahmefällen und bei zwingenden Gründen gebrochen werden.

7.8 Fußgesteuerte do-while-Anweisung

Die do-while-Anweisung ist eine **fußgesteuerte** Schleifenstruktur. Die Bedingung wird erst **nach** dem Durchlauf des Schleifenrumpfes ausgewertet. Dadurch wird der Schleifenkörper **mindestens einmal** ausgeführt. Wie bei der kopfgesteuerten while-Anweisung wird der Schleifenrumpf ausgeführt, solange die Bedingung erfüllt ist.

Beispiel

Zu einer Zahl wird mindestens einmal der Wert 10 addiert. Ist die Zahl dann immer noch kleiner als 100, addieren Sie erneut den Wert 10. Brechen Sie ab, wenn die Zahl größer oder gleich 100 ist.

Syntax do-while-Anweisung

- ✓ Das Schlüsselwort **do** leitet die Anweisung ein.
- ✓ Anschließend folgt der Schleifenrumpf mit der Anweisung oder mehreren Anweisungen innerhalb eines Anweisungsblocks in geschweiften Klammern (**statement**).
- ✓ Zur Einleitung der Schleifensteuerung folgt das Schlüsselwort **while**.
- ✓ Es folgt in runden Klammern **()** die Formulierung der Bedingung, die einen Wert vom Typ **bool** liefern muss.
- ✓ Abschließend folgt das Semikolon.
- ✓ Ist die Bedingung erfüllt (**true**), wird der Schleifenrumpf erneut durchlaufen. Ist die Bedingung nicht erfüllt (**false**), wird die do-while-Anweisung beendet und das Programm fortgesetzt.

```
do
  statement;
while (condition);
```

Beispiel: ControlDoWhile.sln

Das Programm berechnet die Anzahl der Jahre, die benötigt werden, um – beginnend mit einem Startkapital von 1000,- EUR – nur über die Zinsen (4,5 %) einen Endbetrag von 10000,- EUR zu erreichen. Beachten Sie, dass hier zu Demonstrationszwecken von der üblichen Benennung der Zählvariablen abgewichen wird. Das kann in Ausnahmefällen sinnvoll sein, wenn ein sprechender Bezeichner für eine gewisse Klarheit sorgt.

```
static void Main(string[] args)
{
    ①   double presentValue = 1000.0;
    L   double futureValue = 10000.0;
    ②   const double INTERESTRATE = 4.5;
    ③   int year = 0;
    do
    {
        ④   presentValue *= 1.0 + INTERESTRATE / 100; // Startwert um Zinsen erhöhen
        L   year++;                                // Jahr um 1 erhöhen
        // Alternative: year++;
    }
    ⑤   while (presentValue < futureValue);
    //solange das gewünschte Kapital noch nicht erreicht ist
    ⑥   Console.WriteLine("Laufzeit in Jahren: " + year);
}
```

- ① Die Variablen enthalten die Werte für das Startkapital (**presentValue**) und das zu erreichende Zielkapital (**futureValue**). Da für die Zinsberechnung Kommastellen notwendig sind, werden Gleitkommazahlen (**double**) verwendet.

- ② Der Zinssatz wird mit 4,5 % als Konstante festgelegt.
- ③ Die Variable `year` enthält als Zähler die Anzahl der Jahre, die vergehen, bis das gewünschte Kapital erreicht ist.
- ④ Im Schleifenrumpf werden die Zinsen berechnet und zum aktuellen Kapital addiert. Der Jahreszähler (`year`) wird jeweils um 1 erhöht.
- ⑤ Solange das gewünschte Kapital nicht erreicht wurde, wird der Schleifenkörper erneut ausgeführt.
- ⑥ Ist das Kapital erreicht, wird die Anzahl der Jahre, die zum Erreichen des Endbetrags notwendig sind, ausgegeben.

7.9 Zählergesteuerte Wiederholung

Jede Schleifenform kann man als zählergesteuerte Iteration aufbauen. Eine Schleife ist aber durch ihren Aufbau besonders für die Variante geeignet. Die `for`-Anweisung, eine zählergesteuerte Wiederholung, ist wie die `while`-Anweisung ebenfalls eine **kopfgesteuerte** Schleife. Die `for`-Anweisung zeichnet sich durch eine kompakte Schreibweise aus. Die Anzahl der Schleifendurchläufe wird durch einen Anfangswert, einen Endwert und die Schrittweite festgelegt. Der Schleifenrumpf wird mithilfe eines Zählers in der gewünschten Anzahl wiederholt ausgeführt.

Syntax der `for`-Anweisung

- ✓ Das Schlüsselwort `for` leitet die zählergesteuerte Wiederholung ein.
- ```
for ([type] counter = start; condition; nextStatement)
 statement;
```
- ✓ Anschließend folgt in runden Klammern `( )` die Schleifensteuerung. Diese beginnt mit der Festlegung der Zählervariablen (`counter`) auf den Startwert (`start`). Die Zählervariable kann auch hier deklariert werden und ist dann nur in dem Schleifenrumpf gültig. Die Zählervariable muss einen numerischen Datentyp besitzen.
- ✓ Abgetrennt durch ein Semikolon folgt die Bedingung (`condition`), die erfüllt sein muss, damit die Schleife erneut ausgeführt wird.
- ✓ Nach einem weiteren Semikolon folgt der Aktualisierungsteil (`nextStatement`). Als Anweisung beschreiben Sie hier beispielsweise, wie der Wert der Zählvariablen bei jedem Schleifendurchlauf verändert wird (z. B. Inkrementieren oder Dekrementieren). Die im Aktualisierungsteil aufgeführte Anweisung wird für jeden Schleifendurchlauf nur einmal ausgeführt.
- ✓ Für jeden Schleifendurchlauf wird die Anweisung bzw. der Anweisungsblock (`statement`) ausgeführt.

### Beispiel: `ControlFor.sln`

Mit diesem Programm können Sie die Fakultät einer natürlichen Zahl im Bereich zwischen 1 und 15 berechnen. Zum Beispiel ist die Fakultät der Zahl 4 gleich 24 (Berechnung:  $4! = 1 * 2 * 3 * 4$ ).

Achten Sie darauf, dass die Fakultät auch bei kleinen Zahlen sehr schnell wächst, weshalb hier die Berechnung der Fakultät auf Werte zwischen 1 und 15 beschränkt ist.

Beachten Sie weiter, dass in dem Beispiel von der Konvention abgewichen wird, eine Zählvariable immer mit dem Wert 0 zu initialisieren. Das kann in Ausnahmefällen sinnvoll sein, wenn damit der weitere Code vereinfacht wird.

```

static void Main(string[] args)
{
 ① int result = 1;
 ② int number = 0;
 Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 15 ein: ");
 string txt = Console.ReadLine();
 ③ number = Convert.ToInt32(txt);
 ④ if (number >= 1 && number <= 15)
 {
 ⑤ for (int i = 1; i <= number; i++)
 {
 ⑥ result *= i;
 }
 ⑦ Console.WriteLine($"{0}! = {1}", number, result);
 }
 else
 {
 ⑧ Console.WriteLine("Die Zahl liegt nicht im gültigen Bereich");
 }
}

```

- ① Die Variable `result` speichert die berechnete Fakultät.
- ② Die Variable `number` speichert den Zahlenwert, für den die Fakultät berechnet werden soll.
- ③ Die übergebene Textzeile wird in eine Zahl vom Datentyp `int` umgewandelt und der Variablen `number` zugewiesen.
- ④ Für Zahlen kleiner als 1, für die die Fakultät nicht berechnet werden kann, und für Zahlen größer als 15 erfolgt eine entsprechende Ausgabe ⑧ und das Programm wird beendet.
- ⑤ Innerhalb der `for`-Anweisung wird die Variable `i` definiert und mit dem Wert 1 initialisiert. Die Schleife soll ausgeführt werden, solange der Zähler `i` kleiner oder gleich dem Wert `number` ist. Die Variable `i` wird nach jedem Schleifendurchlauf inkrementiert (um den Wert 1 erhöht).
- ⑥ Die Fakultät wird berechnet. Durch die Wiederholungen des Schleifenrumpfs entsteht eine Formel der Art:  

$$\text{result} = 1 * 2 * 3 * \dots * \text{number}$$
- ⑦ Das Ergebnis der Berechnung wird ausgegeben.

### Tipps und häufige Verwendungsformen der `for`-Anweisung



Sofern die Zählervariable bereits definiert ist, können Sie sie in der `for`-Anweisung direkt verwenden.

```

int i;
...
for (i = 0; i <= 10; i++)
...

```

Sie können `for`-Anweisungen beliebig verschachteln. In diesem Beispiel wird die äußere Schleife 10-mal durchlaufen und in jedem dieser Durchläufe wird die innere Schleife 20-mal durchlaufen.

```

for (int i = 0; i < 10;
i++)
{
 for (int j = 0; j < 20;
j++)
 {
 ...
 }
}

```

## 7.10 Weitere Kontrollstrukturen

### Schleifensteuerung mit **break** und **continue**

Zusätzlich zur Schleifensteuerung haben Sie mit einer `break`- bzw. einer `continue`-Anweisung die Möglichkeit, den Ablauf des Programms zu beeinflussen.

- ✓ Wie in der `switch`-Anweisung kann die `break`-Anweisung auch innerhalb von Schleifen angewendet werden, um die jeweilige Kontrollstruktur zu beenden.
- ✓ Die `continue`-Anweisung können Sie innerhalb von Schleifen einsetzen. Bei einer bedingten Wiederholung wird sofort die Testbedingung neu ausgewertet. Bei der zählergesteuerten Wiederholung (`for`) wird sofort der nächste Zählerwert verwendet und danach die Testbedingung neu ausgewertet.

Bei geschachtelten Kontrollstrukturen wirkt sich der Aufruf von `continue` bzw. `break` nur für die jeweils innere Schleife aus.

### Beispiel für die Anwendung von **break** und **continue**

- ① Hier beginnt eine `while`-Anweisung.
- ② Innerhalb der `while`-Anweisung wird eine Bedingung durch eine `if`-Anweisung geprüft.
- ③ Wenn die Bedingung bei ② erfüllt ist, wird die `while`-Anweisung durch die `break`-Anweisung verlassen. Die Anweisungen nach der `while`-Anweisung ⑥ werden als Nächstes ausgeführt.
- ④ Es wird eine weitere Bedingung durch eine `if`-Anweisung geprüft.
- ⑤ Wenn diese Bedingung erfüllt ist, wird die `continue`-Anweisung ausgeführt. Die Schleifenprüfung beginnt wieder bei ①. Die Anweisungen nach der `continue`-Anweisung werden in diesem Schleifendurchlauf nicht mehr ausgeführt.

```

① while (i < 20)
 {
 ...
 ② if (i == 5)
 {
 break;
 }
 ...
 ④ if (i == 10)
 {
 ⑤ continue;
 }
 ...
 ⑥ }
 ...

```

*Schleifensteuerung mit break und continue*

### Sprunganweisung mit **goto**

Eine `goto`-Anweisung veranlasst das Programm, von einer Stelle zu einer anderen Stelle zu springen. Sie können die `goto`-Anweisung nur innerhalb des gleichen Gültigkeitsbereichs verwenden, z. B. innerhalb einer Methode.

- ① Hinter dem Schlüsselwort `goto` folgt eine Sprungmarke, zu der das Programm springen soll, falls die Bedingung erfüllt ist.
- ② Das Programm springt bei erfüllter Bedingung an diese Sprungmarke. Anschließend werden die Anweisungen nach dem Doppelpunkt abgearbeitet. Alle Anweisungen bis zur Anweisung ② bleiben bei Ausführung der Sprunganweisung unberücksichtigt.

```

...
if (wert > 10)
{
 ① goto Marke;
}
...
② Marke: ...
...
```



Der Programmcode wird durch Sprunganweisungen in der Regel sehr unleserlich. Verzichten Sie deshalb möglichst auf die Verwendung der `goto`-Anweisung. Setzen Sie stattdessen `switch`-Anweisungen ein oder verwenden Sie Methoden, die Sie im weiteren Verlauf des Buches kennenlernen werden. Die Verwendung von `goto` ist in der modernen Programmierung ein No-Go.

## 7.11 Codeausschnitte zu Kontrollstrukturen

| Beschreibung                        | Shortcut+  |
|-------------------------------------|-----------------------------------------------------------------------------------------------|
| Verzweigung: if...                  | if                                                                                            |
| Verzweigung: else...                | else                                                                                          |
| Verzweigung: switch                 | switch                                                                                        |
| Schleife: while ...                 | while                                                                                         |
| Schleife: do...while                | do                                                                                            |
| Schleife: for... (aufwärts zählend) | for                                                                                           |
| Schleife: for... (abwärts zählend)  | forr                                                                                          |

## 7.12 Übungen

### Übung 1: Testauswertung

Übungsdatei: --

Ergebnisdateien: *TestResultat.sln*, *TestResultat2.sln*,  
*TestResultat3.sln*

1. Schreiben Sie eine Konsolenanwendung mit dem Namen *TestResultat*, die die Bewertung eines Tests als Text ausgibt. In dem Test können maximal 10 Punkte erreicht werden. Sofern mindestens 7 Punkte erreicht wurden, gilt der Test als bestanden. Die Punktzahl soll im Programm eingelesen werden (Readline). Die Textausgaben sollen folgendermaßen lauten:  
 Mindestens 7 Punkte: *Der Test ist bestanden!*  
 anderenfalls: *Der Test ist leider nicht bestanden!*
2. Werten Sie so lange neue Punktzahlen aus, bis als Punktzahl der Wert -1 eingegeben wird, und prüfen Sie zusätzlich, ob die Punktzahl im zulässigen Bereich liegt. Andernfalls soll der Text *Ungültige Punktzahl!* ausgegeben werden.
3. Kennzeichnen Sie den Programmcode zur Auswertung der Punktzahl als Kommentar und schreiben Sie eine neue Auswertung: Verwenden Sie dabei eine Kontrollstruktur, mit der Sie folgende Auswertung vornehmen können:

10 Punkte: *Ergebnis: Sehr gut*

9 Punkte: *Ergebnis: Gut*

8 Punkte: *Ergebnis: Befriedigend*

7 Punkte: *Ergebnis: Ausreichend*

weniger als 7 Punkte: *Ergebnis: Leider nicht genügend Punkte erreicht*

## Übung 2: Passwortabfrage

**Übungsdatei:** --

**Ergebnisdatei:** Passwortabfrage.sln

1. Erstellen Sie eine Windows-Anwendung zur Passwort-Abfrage. Verwenden Sie einen Button- und zwei TextBox-Komponenten.
2. Aktivieren Sie bei der Ausgabe-Textbox über das SmartTask-Menü der Textbox die Eigenschaft Multiline.

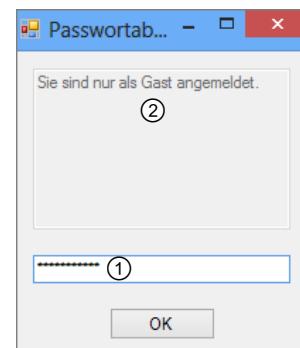
Setzen Sie die Eigenschaft Enabled der Ausgabe-Textbox auf False.

Tragen Sie bei der Eigenschaft PasswordChar der Textbox für die Eingabe ① das Zeichen 8 ein, damit die Passwörter bei der Eingabe verdeckt erscheinen.

Legen Sie zwei Passwörter (z. B. ad2013min für den Administrator und einUser für User) fest.

Nachdem der Benutzer die Schaltfläche OK betätigt hat, soll die eingegebene Zeichenkette geprüft werden: Je nachdem, welches der Passwörter der Anwender eingibt, soll in der Textbox ② folgender Text ausgegeben werden: *Sie sind als Administrator angemeldet*. Oder: *Sie haben sich als User angemeldet*. Bei jedem anderen eingegebenen Passwort soll *Sie sind nur als Guest angemeldet* angezeigt werden.

Verwenden Sie eine entsprechende Kontrollstruktur, mit der Sie noch weitere Passwörter prüfen könnten.

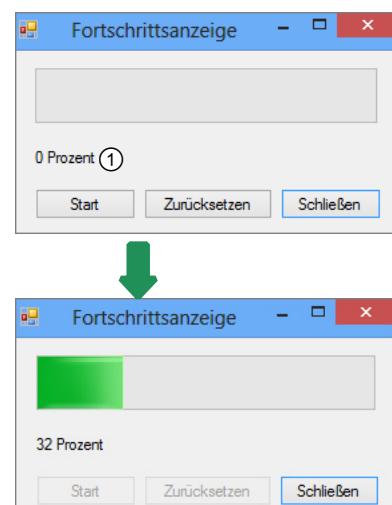


## Übung 3: Fortschrittsanzeige

**Übungsdatei:** --

**Ergebnisdatei:** Fortschrittsanzeige.sln

1. Erzeugen Sie eine Windows-Anwendung zur grafischen Anzeige eines Bearbeitungsfortschritts. Verwenden Sie eine ProgressBar-, eine Label- und drei Button-Komponenten.
2. Erstellen Sie eine Schleifenkonstruktion, um die Fortschrittsanzeige vom Minimum zum Maximum zu bewegen. Setzen Sie dazu die Eigenschaft Minimum der ProgressBar-Komponente auf den Wert 0 und die Eigenschaft Maximum auf den Wert 100. Erhöhen Sie in einer Schleife den Wert der ProgressBar-Komponente (Eigenschaft Value) bis zum Maximalwert (100).
3. Zeigen Sie den Fortschritt außerdem mithilfe der Label-Komponente an ①.
4. Um den Fortschritt mitverfolgen zu können, müssen Sie die Abarbeitung der Schleife etwas bremsen. Mithilfe der folgenden Anweisung wartet die Anwendung bei jedem Schleifendurchlauf 100 Millisekunden: `System.Threading.Thread.Sleep(100)`. Dabei wird hier ein fortgeschrittenes Thema (Multithreading) berührt, was nicht Bestandteil des Buchs ist. Nur so weit als kurze Einführung: Multithreading erlaubt die parallele Ausführung von mehreren Prozessen innerhalb eines Programms und dies ist hier notwendig. Mit der Sleep-Anweisung unterbricht man den aktuellen Thread für eine gewisse Dauer und andere Prozesse können in der Zeit aktiv werden.
5. Mit der Anweisung `Application.DoEvents()` können Sie erreichen, dass bei jedem Schleifendurchlauf geprüft wird, ob für die Anwendung vorliegende Ereignisse bearbeitet werden sollen. Dadurch wird es möglich, z. B. die Anwendung schon vor dem vollständigen Abarbeiten der Schleife durch Betätigen der Schaltfläche Schließen zu beenden.
6. Stellen Sie sicher, dass die Schaltflächen Start und Zurücksetzen nicht betätigt werden können, solange die Schleife abgearbeitet wird (Eigenschaft Enabled).

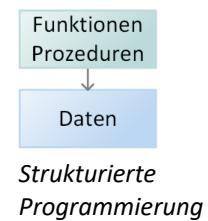


# 8 Klassen, Felder und Methoden

## 8.1 Grundlagen der objektorientierten Programmierung

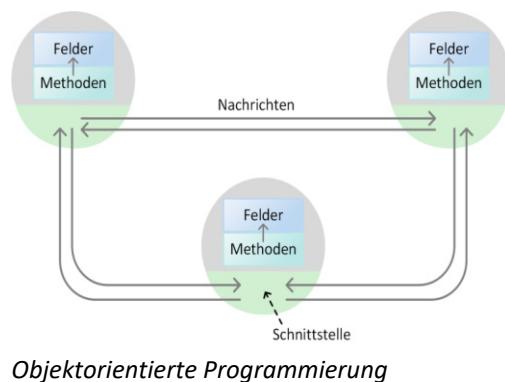
Der bis in die achtziger Jahre vorherrschende Programmierstil war die sogenannte strukturierte Programmierung. Dabei wird eine Gesamtaufgabe in einzelne Teilaufgaben zerlegt, die beispielsweise durch Funktionen und Prozeduren gelöst werden. Die Daten und die Routinen, die diese Daten bearbeiten, sind voneinander unabhängig.

Der Weiterentwicklung zur objektorientierten Programmierung lag die Idee zugrunde, einzelne Objekte mit ihrem Zustand sowie den möglichen Operationen und Eigenschaften in einer Datenstruktur zu verwalten.



*Strukturierte  
Programmierung*

- ✓ Die Objekte enthalten in ihrem Inneren sowohl **Daten** als auch **Methoden** für die Verarbeitung dieser Daten.
- ✓ Die **Methoden** beschreiben die Funktionalität der Objekte.
- ✓ Die **Daten** heißen im Sprachgebrauch von Microsoft **Felder**. Sie ähneln Variablen und speichern die spezifischen Merkmale eines Objekts. Sie unterscheiden die Objekte untereinander durch ihre unterschiedlichen Werte. In vielen anderen objektorientierten Sprachen und in den theoretischen Konzepten der Objektorientierung wird bei Feldern bzw. Daten meist von **Eigenschaften** eines Objekts gesprochen. Bei Microsoft haben Eigenschaften eine engere Bedeutung (vgl. Abschnitt 9.2).
- ✓ Bei entsprechender Programmierung besitzen Objekte einen Schutz für ihre Daten, der auch **Kapselung** genannt wird. Von **außen** kann auf die Daten dann nur mithilfe der Methoden des Objekts zugegriffen werden.
- ✓ Die verschiedenen zur Lösung einer Aufgabe erzeugten Objekte kommunizieren untereinander durch den Austausch von Nachrichten. Gegenüber der strukturierten Programmierung verbessert die objektorientierte Programmierung die Wartungsbedingungen des Programmcodes sowie die Möglichkeiten zur Wiederverwendung von Programmcode.



*Objektorientierte Programmierung*

### Vorteile der objektorientierten Programmierung

- ✓ Der Schutz von Daten verbessert sich durch Kapselung der Daten und Methoden.
- ✓ Die kompakte Struktur der Objekte erleichtert die Fehlersuche.
- ✓ Die Quelltexte werden überschaubarer.
- ✓ Änderungen können einfacher eingearbeitet werden.
- ✓ Neue Objekte bzw. Komponenten können leichter in ein Programm integriert werden.
- ✓ Das dynamische Erzeugen und Freigeben von Objekten während des Programmablaufs sorgt für einen effizienten Umgang mit Speicherplatz.

## 8.2 Klassen und Instanzen

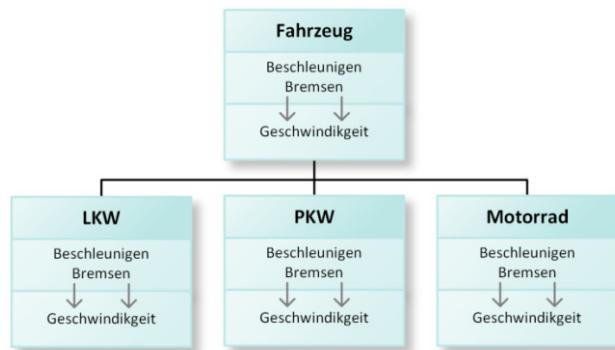
### Was ist eine Klasse, was sind Objekte?

Eine **Klasse** beschreibt als Bauplan die Gemeinsamkeiten einer Menge von **Objekten**. Eine Klasse ist somit ein Modell, auf dessen Basis Objekte erstellt werden können. Die Klasse beinhaltet die vollständige Beschreibung dieses Modells. So vereint eine Klasse alle **Felder** (auch Datenelemente genannt), die diese Klasse kennzeichnen, und alle **Methoden** zur Verwendung der Daten und zur Beschreibung der Funktionalität. Da die Felder und Methoden zu der Klasse gehören, werden sie auch als **Member** (Mitglieder) der Klasse bezeichnet.

**Objekte (Instanzen)** stellen konkrete Exemplare der Klasse dar. Auf der Basis einer Klasse können beliebig viele Objekte erzeugt (**instanziert**) werden. In den Feldern werden die objektspezifischen Werte für das jeweilige Objekt gespeichert, während die Methoden Aktionen ausführen. So können Methoden Daten für die Klasse in Empfang nehmen, Feldinhalte (Daten) der Klasse ausgeben oder die Feldinhalte der Klasse be- bzw. verarbeiten. Man sagt, dass Objekte vom **Typ** der Klasse sind.

### Beispiel

Die nebenstehende Grafik zeigt eine Klasse mit dem Namen *Fahrzeug* sowie darunter drei Instanzen dieser Klasse (beachten Sie, dass dies kein reines Klassendiagramm ist und sowohl die Klasse als auch deren Instanzen hier angezeigt werden). Die Objekte besitzen die beiden Methoden *Beschleunigen* und *Bremsen*, mit denen die Geschwindigkeit des jeweiligen Objekts verändert werden kann.



### Syntax der Klassendeklaration

- ✓ Die Klassendeklaration beginnt mit dem Schlüsselwort `class`.
- ✓ Bei Bedarf kann ein Modifizierer (modifier) vorangestellt werden. Ohne entsprechende Angabe werden Klassen in Visual C# als `internal` (intern) deklariert.
- ✓ Anschließend folgt der Name der Klasse, der sich an die Konventionen für Bezeichner halten muss und üblicherweise mit einem Großbuchstaben beginnt.
- ✓ Es folgen in geschweiften Klammern `{ } { }` die Deklarationen der Member.
- ✓ Eine Klasse kann Datenelemente (wie z. B. Konstanten und Variablen bzw. Felder) und Funktionselemente (wie z. B. Methoden, Eigenschaften, Operatoren und Konstruktoren) enthalten.
- ✓ Auch den Membern können Sie Modifizierer voranstellen (vgl. nachfolgende Übersicht). Ohne diese Angabe wird der Modifizierer `private` verwendet.
- ✓ Sie können Klassen auch ineinander verschachteln, d. h. innerhalb von Klassen weitere Klassen deklarieren. Nur dort ist es möglich, Klassen als `private` zu deklarieren.
- ✓ Die Namen der Member einer Klasse beginnen üblicherweise mit einem Großbuchstaben.
- ✓ Bei Feldern können Sie mit dem Schlüsselwort `readonly` vor dem Namen erreichen, dass das Feld vor Schreibzugriffen geschützt ist. Es kann nur bei der Deklaration oder in einem Konstruktor initialisiert werden.

```

[modifier] class Classname
{
 [modifier1] Classmember1
 [modifier2] Classmember2
 ...
}

```

```
private readonly int i;
```

## Modifizierer für die Sichtbarkeit der Member (Zugriffsmodifizierer)

| Modifizierer       | Bedeutung                                                                                                                                   |    |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------|----|
| private            | Der Zugriff auf das Element ist nur innerhalb der Klasse möglich.                                                                           |    |
| protected          | Der Zugriff ist nur innerhalb der Klasse und innerhalb aller davon abgeleiteten Klassen möglich. *)                                         | *) |
| internal           | Der Zugriff auf das Element ist innerhalb einer Assembly (Programm, Klassenbibliothek) möglich.                                             |    |
| protected internal | Der Zugriff auf das Element ist innerhalb einer Assembly (Programm, Klassenbibliothek) und innerhalb aller abgeleiteten Klassen möglich. *) | *) |
| public             | Der Zugriff auf das Element ist nicht eingeschränkt.                                                                                        |    |

\*) Anmerkung: Abgeleitete Klassen lernen Sie im Kapitel 10 kennen.

Die Modifizierer können auch auf die Klassen selbst angewendet werden. Die Beschreibung gilt dann sinngemäß. Neben Zugriffsmodifizierern gibt es weitere Modifizierer, die Sie noch im Verlauf des Buches kennenlernen werden, z. B. static, der bei der Main()-Methode verwendet wird.

## Klasseninstanzen (Objekte) erzeugen

- ✓ Um eine Instanz einer Klasse zu erzeugen, vereinbaren Sie zunächst eine Variable. Als Datentyp geben Sie den Namen der Klasse an ①.
- ✓ Die Instanz einer Klasse wird auch Objekt der Klasse genannt. Die Variable wird daher auch als Objektvariable bezeichnet
- ✓ Die Variable ist eine **Referenzvariable**. Bei der Vereinbarung der Variablen wird nur Speicher für die Referenz auf das Objekt angelegt. Das Objekt selbst existiert noch nicht. Die Variable besitzt den Wert null, d. h., sie verweist auf kein Objekt. Aus diesem Grund spricht man in dem Zusammenhang auch von **Referenztypen**.
- ✓ Mithilfe des Schlüsselwortes new wird Speicherplatz angefordert, der die erzeugte Instanz aufnimmt. Die Adresse des Speicherplatzes wird der Objektvariablen zugewiesen ②, sodass die Objektvariable auf die neu erzeugte Instanz verweist.
- ✓ Die Schritte ① und ② lassen sich auch in einer Zeile zusammenfassen ③.

```
Classname objectname; ①
objectname = new Classname(); ②
// oder die Kurzform:
Classname objectname = new Classname(); ③
```

## Auf die Member eines Objekts zugreifen

- ✓ Der Zugriff auf die Member (Methoden, Felder) erfolgt über den Namen der Instanz, einem Punkt und den Namen des Members ①. Die Bezeichner auf der linken Seite eines Punktoperators werden auch als **Qualifizierer** bezeichnet.
- ✓ Wenn Sie eine Instanz bzw. ein Objekt nicht mehr benötigen, können Sie die Objektvariable durch Zuweisung von null freigeben ②. Dies ist aber in der Regel nicht notwendig, da die Garbage Collection (automatische Speicherbereinigung) den belegten Speicherplatz auch ohne ausdrückliche Freigabe der Referenz zurückgewinnt, wenn ein Objekt nicht mehr verwendet (referenziert) wird.

```
objectname.membername = ... ①
objectname = null; ②
```

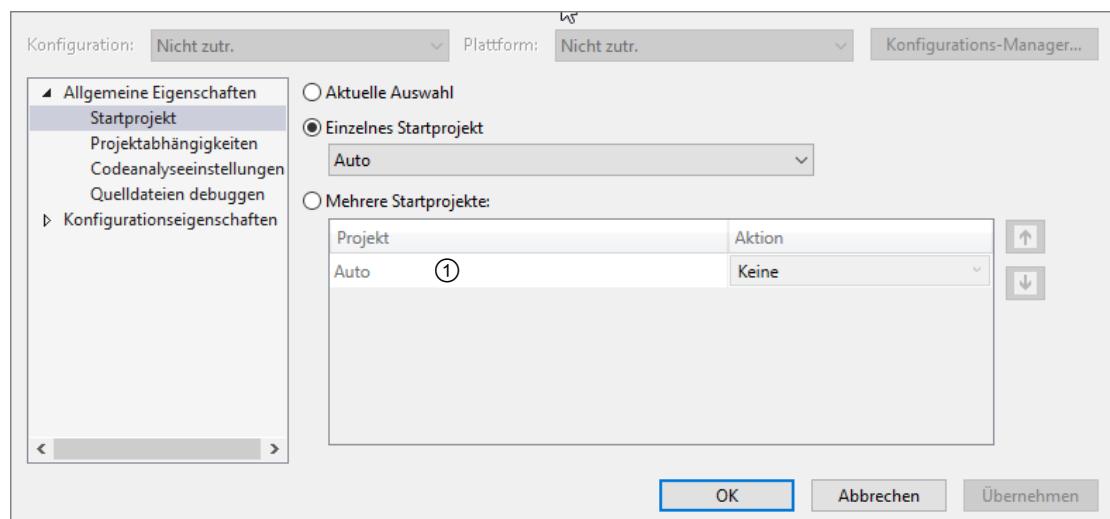
### Beispiel: Auto.sln

Im folgenden Beispiel wird neben der Klasse `Program` eine Klasse `Fahrzeug` deklariert. Sie besitzt ein Feld mit dem Namen `geschwindigkeit`, in dem ein Zahlenwert gespeichert werden kann.

Beachten Sie, dass Microsoft alle Felder per Konvention kleinschreibt und Felder explizit von Eigenschaften unterscheidet, die dann per Konvention großgeschrieben werden. Vereinfacht kann man sagen, dass Felder privat gehaltene Eigenschaften sind. Sie werden also von außerhalb einer Klasse nicht zu sehen sein. Eigenschaften hingegen sind explizit dafür gedacht, bei Bedarf nach außen sichtbar gemacht zu werden. In der Regel werden Felder und Eigenschaften in der Microsoft-Philosophie sogar identisch benannt, nur Felder eben klein- und Eigenschaften großgeschrieben. Darauf kommen wir noch zurück. In den folgenden Beispielen werden wir allerdings diese spezielle Differenzierung zwischen Feldern und Eigenschaften und die individuelle Syntax von Eigenschaften in der Microsoft-Philosophie noch nicht berücksichtigen, sondern verwenden die sonst weltweit üblichen Notationen bzw. Begrifflichkeiten, in der Felder als Eigenschaften aufgefasst werden, wenn sie von außerhalb der Klasse zugänglich sind.

Eine der beiden Klassen dieser folgenden Konsolenanwendung muss nun eine Startmethode `Main()` besitzen, damit das Programm ausführbar ist.

Im Projektmappen-Explorer stellen Sie unter den Eigenschaften des Projekts (erreichbar über das Kontextmenü oder das Menü `Projekt`) die Klasse mit der Methode `Main()` als Startobjekt ein ①, sofern es mehrere Klassen mit einer `Main()`-Methode in Ihrem Projekt gibt. Ansonsten können Sie diese Angabe frei lassen.



*Startobjekt auswählen*

```

① class Fahrzeug
{
 internal int geschwindigkeit;
}

② class Program
{
 ③ static void Main(string[] args)
 {
 ④ Fahrzeug PKW = new Fahrzeug();
 PKW.geschwindigkeit = 54;
 }
}

```

```

⑥ Fahrzeug fahrrad = new Fahrzeug();
fahrrad.geschwindigkeit = 18;
Console.WriteLine("Geschwindigkeit: {0} km/h",
 PKW.geschwindigkeit);
Console.WriteLine("Geschwindigkeit: {0} km/h",
 fahrrad.geschwindigkeit);
}
}

```

- ① Die Klasse `Fahrzeug` mit dem Feld `geschwindigkeit` wird deklariert. Dieses wird mit einem Zugriffsmodifizierer als zugänglich aus einer anderen Klasse des gleichen Assemblies markiert (`internal`).
- ② In der zweiten Klasse `Program` befindet sich die Methode `Main()` ③, die den Einstiegspunkt in das Programm repräsentiert und den Programmablauf steuert. Die Methode `Main()` muss mit dem Schlüsselwort `static` deklariert werden, da sie aufgerufen wird, ohne dass von der Klasse `Program` eine Instanz erzeugt wird.
- ④ Eine Instanz der Klasse `Fahrzeug` wird erzeugt und der lokalen Objektvariablen `PKW` zugewiesen.
- ⑤ Dem Feld `geschwindigkeit` des Objekts `PKW` wird der Wert 54 zugewiesen. Damit aus der Klasse `Program` auf das Feld der Klasse `Fahrzeug` zugegriffen werden kann, ist das Feld mit dem Modifizierer `internal` deklariert. Ein Zugriff ist so von allen Stellen innerhalb der Assembly erlaubt.
- ⑥ Eine Instanz der Klasse `Fahrzeug` wird erzeugt und einer Objektvariablen `fahrrad` zugewiesen.
- ⑦ Die aktuelle Geschwindigkeit des Objekts `PKW` und des Objekts `fahrrad` wird ausgegeben.

Beachten Sie, dass der Name der Objektvariablen `PKW` kritisch ist, denn nach Konvention schreibt man eigentlich nur Konstanten vollständig groß. Auf der anderen Seite werden aber Abkürzungen wie `PKW`, `LKW` etc. ebenfalls oft vollständig großgeschrieben. Mit der nötigen Vorsicht kann man deshalb auch solche Variablen großschreiben, wobei man in der Praxis dann besser einen Bezeichner verwendet, der solche Unklarheiten erst gar nicht aufkommen lässt.

Besteht eine Anwendung aus mehreren umfangreichen Klassen, ist es sinnvoll, für jede Klasse oder für jede Gruppe zusammengehöriger Klassen eine separate Datei zu erzeugen. Dadurch wird die Übersichtlichkeit Ihrer Programme erhöht. Zum Hinzufügen einer ausgelagerten Klasse (d. h. einer neuen Klassendatei) rufen Sie den Menüpunkt *Projekt - Klasse hinzufügen...* auf, während das Projekt im Projektmappen-Explorer ausgewählt ist. Alternativ können Sie das Kontextmenü im Projektmappen-Explorer verwenden, wenn Sie auf ein Projekt geklickt haben. Im Beispiel wurden beide Klassen in einer Datei abgelegt.



## Mit Referenzvariablen arbeiten

Bei den **Variablen primitiver Datentypen** können Sie mit einer Wertzuweisung einer Variablen den Wert einer anderen Variablen zuweisen.

```

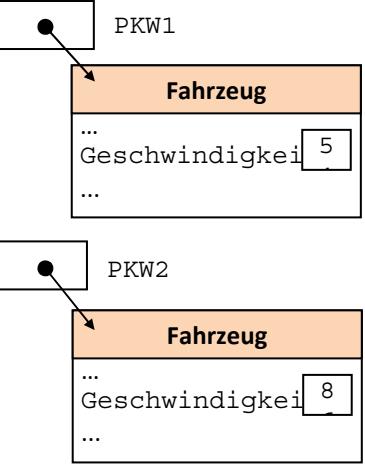
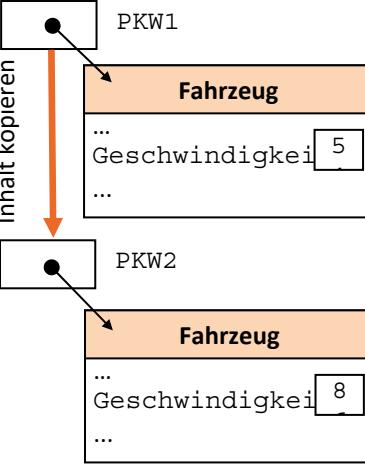
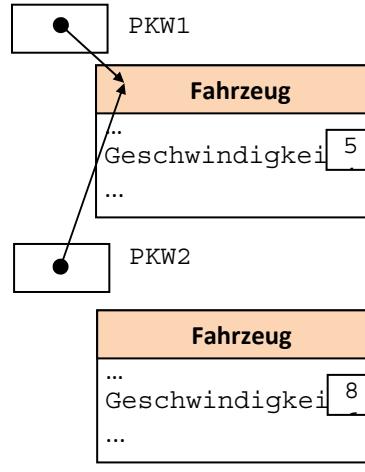
int a = 5;
int b = 6;
b = a; // b erhält den Wert 5

```

*Wertzuweisung bei Variablen primitiver Datentypen*

Bei **Objektvariablen** handelt es sich demgegenüber um sogenannte **Referenzvariablen**. In der Variablen wird eine Referenz (Verweis) auf das eigentliche Objekt gespeichert. Aber da dies nur eine Adresse und damit ein Wert ist, ist die Situation gar nicht so unterschiedlich, wie es oft dargestellt wird.

### Zuweisungen bei Referenzvariablen

| Ausgangszustand                                                                                                                               | Zuweisung                                                                         | Ergebnis                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|                                                              |  |  |
| <pre>Fahrzeug PKW1; PKW1 = new Fahrzeug(); PKW1.Geschwindigkeit = 54;  Fahrzeug PKW2; PKW2 = new Fahrzeug(); PKW2.Geschwindigkeit = 86;</pre> | <pre>PKW2 = PKW1;</pre>                                                           | Beide Referenzvariablen verweisen anschließend auf dasselbe Objekt.                 |

Entsprechend wird bei der Anwendung der Operatoren == und != auf Referenztypen geprüft, ob die Variable auf dasselbe Objekt verweist. Es wird nicht der Inhalt verglichen. Allerdings ist eine Gleichheit des Inhalts trivial, wenn es sich um das gleiche Objekt handelt. Die Umkehrung ist nur falsch oder zumindest nicht zwingend.

## 8.3 Methoden – die Funktionalität der Klassen

### Funktionalität mit Methoden beschreiben

Durch Felder legen Sie die Eigenschaften von Objekten einer Klasse fest. Neben diesen Feldern können auch Methoden zu einer Klasse gehören. Mithilfe von Methoden drücken Sie die Funktionalität der Objekte einer Klasse aus.

| Klasse   |
|----------|
| Felder   |
| Methoden |

- ✓ Methoden werden wie die Felder innerhalb einer Klasse deklariert.
- ✓ Methoden gehören immer zu einer Klasse.
- ✓ Eine Klasse kann mehrere Methoden enthalten, wobei die Methoden in beliebiger Reihenfolge deklariert werden können.
- ✓ Mit Methoden führen Sie eine Aktion aus, verändern Daten oder geben Daten aus.
- ✓ Methoden können nicht geschachtelt werden, d. h., Sie können innerhalb einer Methode keine weitere deklarieren. Andere Methodenaufrufe innerhalb einer Methode sind allerdings möglich und notwendig.

Methoden der Klasse `Fahrzeug` sind beispielsweise:

- ✓ das Ausgeben (Anzeigen) des Geschwindigkeitswertes,
- ✓ das Verändern der Geschwindigkeit.

## Mit Methoden arbeiten

- ✓ Wenn eine Methode eines Objekts in einem Programm aufgerufen wird, werden die in der Methode zusammengefassten Anweisungen abgearbeitet. Anschließend wird das Programm fortgesetzt.
- ✓ Methoden bieten auch die Möglichkeit, nach der Abarbeitung einen Wert (z. B. ein Ergebnis) zurückzugeben.
- ✓ Klassen und ihre Methoden, die einmal entwickelt und getestet wurden, können beliebig oft verwendet bzw. aufgerufen werden.

## 8.4 Einfache Methoden erstellen

### Syntax für die Beschreibung einer einfachen Methode

- ✓ Die Beschreibung einer Methode erfolgt durch den Methodenkopf ① und den Methodenrumpf ②.
- ✓ Der Methodenkopf beginnt mit dem Schlüsselwort void.

```
[modifiers] void Methodname () ①
{
 ...
 [return;]
}
```

- ✓ Nach dem Schlüsselwort void folgt der Methodename und ein Klammernpaar ① ②. Der Methodename muss sich an die Regeln für Bezeichner halten. Methoden beginnen in der Microsoft-Konvention üblicherweise mit einem Großbuchstaben. Die meisten anderen Konventionen verwenden Kleinschreibung.
- ✓ Die Anweisungen der Methode werden als Anweisungsblock in geschweifte Klammern ① ② geschrieben.
- ✓ Sie können eine Methode jederzeit mit der Anweisung return verlassen. Die Programmausführung wird mit der ersten Anweisung, die dem Methodenaufruf folgt, fortgesetzt.
- ✓ Der Methodendefinition können Sie sogenannte Modifizierer (modifiers) voranstellen, um beispielsweise spezielle Zugriffsrechte auf die Methode festzulegen.

### Syntax für die Anwendung einer Methode

- ✓ Schreiben Sie eine Anweisung, die mit dem Namen des Objektes beginnt, dessen Methode Sie aufrufen möchten.
- ✓ Anschließend folgt – abgetrennt durch einen Punkt – der Name der Methode mit einem Klammernpaar ① ② ①.
- ✓ Die Methoden, die über ein Objekt aufgerufen werden können, werden Ihnen über die IntelliSense zur Auswahl gestellt.
- ✓ Wenn Sie innerhalb einer Klasse in einer Methode eine andere Methode **derselben** Klasse aufrufen, schreiben Sie nur den Methodennamen mit dem Klammernpaar ① ② ②.

```
objectname .Methodname () ①
Methodname () ②
```

### In Methoden mit den Feldern der eigenen Klasse arbeiten

Innerhalb der Klasse können Sie direkt auf die enthaltenen Felder zugreifen. Um jedoch den Zugriff auf die **eigenen** Felder zu verdeutlichen, wird empfohlen, die Objektreferenz this zu verwenden.

### Die Objektreferenz this

Mit jedem Objekt wird vom Compiler automatisch eine Objektreferenz auf das eigene Objekt erzeugt. Diese Referenzvariable this kann in allen Methoden des Objekts eingesetzt werden. Insbesondere im Konstruktor und in allen Methoden zum Setzen von Instanz Eigenschaften ist diese Objektreferenz elementar.

### Beispiel für eine einfache Methode: Auto2.sln

In dem Beispielprogramm wird eine Methode aufgerufen, die die Geschwindigkeit des Fahrzeugs ausgibt.

```

class Fahrzeug
{
 internal int geschwindigkeit;

① internal void Anzeigen()
{
 Console.WriteLine ("Geschwindigkeit: " + this.geschwindigkeit);
}

class Program
{
 static void Main(string[] args)
 {
 Fahrzeug PKW = new Fahrzeug();
 PKW.geschwindigkeit = 54;
 Fahrzeug fahrrad = new Fahrzeug();
 fahrrad.geschwindigkeit = 18;
③ PKW.Anzeigen();
 fahrrad.Anzeigen();
 }
}

```

- ① In der Klasse Fahrzeug wird die Methode Anzeigen() deklariert.
- ② Die Methode Anzeigen() gibt die Geschwindigkeit aus.
- ③ Innerhalb von Main() wird die Methode Anzeigen() des jeweiligen Objekts aufgerufen.



### Programmcode übersichtlich anzeigen

Wenn Sie eine Methode fertiggestellt und getestet haben, können Sie sie im Programmcode-Editor in Visual Studio mithilfe des Symbols ① verkürzt anzeigen lassen (kollabiert) und so die Übersichtlichkeit im Programmcode verbessern.

```

internal int geschwindigkeit;

① 3 Verweise
internal void Anzeigen()
{
 Console.WriteLine("Geschwindigkeit: " + this.geschwindigkeit);
}

internal int geschwindigkeit;

① 3 Verweise
internal void Anzeigen()...

```

Ein erneuter Klick auf das nun als Pluszeichen dargestellte Symbol expandiert den Code wieder.

## Aufrufhierarchie anzeigen

Innerhalb von Methoden können wiederum andere Methoden aufgerufen werden. In einigen Fällen kann es hilfreich sein, diese Hierarchie der Methodenaufrufe zu kennen. Visual Studio bietet dafür eine grafische Unterstützung.

- ✓ Klicken Sie mit der rechten Maustaste auf eine Methode (eine Eigenschaft oder einen Konstruktor) und wählen Sie den Kontextmenüpunkt *Aufrufhierarchie anzeigen*. Es wird das Fenster *Aufrufhierarchie* geöffnet. Darin werden alle Aufrufe bzw. innerhalb des Members inklusive der Datei und Zeile des Aufrufs angezeigt.
- ✓ Wählen Sie in der Auswahlbox ①, ob Sie alle Aufrufe im aktuellen Dokument, Projekt oder der gesamten Projektmappe auswerten möchten.



- ✓ Der Name der Methode (es können auch mehrere Methoden darin angezeigt werden) ist die oberste Hierarchiestufe. Darunter können Sie mit *Aufrufe an „Methodename“* alle Aufrufe der Methode in anderen Methoden anzeigen. Unter *Aufrufe von „Methodename“* werden alle Methodenaufrufe dargestellt, die von der betreffenden Methode durchgeführt werden.
- ✓ Durch einen Doppelklick auf den Eintrag der rechten Seite ② wird die entsprechende Stelle im Quelltext geöffnet.

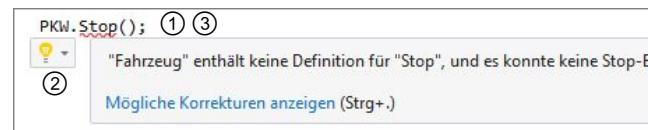
## Methoden generieren

Sie möchten in dem hier verwendeten Beispiel ein Fahrzeug anhalten und dazu eine Methode mit dem Namen `Stop` verwenden.

- Schreiben Sie eine Anweisung, mit der Sie eine bisher noch **nicht existierende** Methode anwenden ①:

z. B. `PKW.Stop();`

Wenn Sie den Mauszeiger über den Bereich bewegen, der den bisher noch nicht deklarierten Methodenaufruf enthält (mit einer roten Wellenlinie unterstrichen), erscheint ein Smarttag-Anzeiger ② mit einem Lampensymbol, die Beschreibung einer Fehler-situation, sowie mögliche Korrekturen. Die Korrekturen bieten an, dass Sie die Methode mit einem formalen Vorgabecode generieren lassen.



Alternativ sehen Sie am rechten Rand des Editors den Anzeiger mit der Lampe. Wenn Sie diesen anklicken, erhalten Sie ebenso die Möglichkeit, dass Sie die noch nicht vorhandene Methode generieren lassen.

- Sollte der Smarttag nicht zu sehen sein, zeigen Sie auf den Balken, um das Smarttag anzuzeigen, oder betätigen Sie die Tastenkombination .

- ▶ Klicken Sie auf den Befehl ③, um in der entsprechenden Klasse (hier Fahrzeug) das Grundgerüst der Methode, einen sogenannten **Methodenstub**, zu generieren ④.
- ▶ Löschen Sie die Zeile `throw new NotImplementedException();` ⑤ und geben Sie die gewünschten Anweisungen ein, z. B.: `this.Geschwindigkeit = 0;`

```

internal void Stop()
{
 throw new NotImplementedException();
}

```

*Beispiel Auto2.sln*

So bequem und einfach diese Möglichkeit zum Generieren einer nicht vorhandenen Methode ist – diese Vorgehensweise ist umstritten, weil man damit leicht der Versuchung erliegt, Mängel in der Konzeption zu kaschieren und damit schlecht durchdachten und qualitativ minderwertigen Code zu erzeugen.

## 8.5 Methoden mit Parametern erstellen

### Grundlagen zu Methoden mit Parametern

Häufig benötigen Sie innerhalb der Methode zusätzliche Informationen zur Ausführung. Sie können deshalb einer Methode Informationen übergeben, die die Methode weiterverarbeiten kann. Diese Informationen werden Parameter genannt.

Um Parameter an eine Methode übergeben zu können, stellt die Methode eine sogenannte **formale Parameterliste** zur Verfügung. In der formalen Parameterliste werden Anzahl und Art der Parameter festgelegt.

### Arten der Parameterübergabe

Eine Parameterübergabe kann auf zwei verschiedene Arten erfolgen:

- ✓ Parameterübergabe als Wert (call by value),
- ✓ Parameterübergabe als Verweis (`ref` = call by reference).

### Syntax für die Beschreibung einer Methode mit Parametern

```
[modifiers] void Methodname([ref] type1 identifier1[, ...])
{
 ...
}
```

- ✓ Innerhalb der Klammern `( )` geben Sie die Parameterliste, auch formale Parameterliste genannt, an.

Zur Beschreibung eines Parameters gehören der Datentyp (`type1`), der Bezeichner (`identifier1`) und ggf. die Parameterart (`ref`).

- ✓ Die Namen der Bezeichner halten sich an die bekannten Konventionen. Da die Parameter wie lokale Variablen nur innerhalb der Methode gültig sind, beginnen ihre Namen üblicherweise mit einem Kleinbuchstaben.
- ✓ Mehrere Parameterbeschreibungen werden jeweils durch ein Komma getrennt.
- ✓ Sie können Parameter jeden Datentyps übergeben.

## Syntax für die Anwendung einer Methode mit Parametern

- ✓ Innerhalb der Klammern des Methodenaufrufs schreiben Sie die Liste der Werte, Variablen, Konstanten oder Ausdrücke, die an die Methode übergeben werden sollen. Diese Liste wird auch als aktuelle Parameterliste bezeichnet.
- ✓ Die Parameter werden durch Kommata voneinander getrennt.
- ✓ Die aktuellen und formalen Parameter müssen im Datentyp und in der Anzahl übereinstimmen.
- ✓ Wenn Sie bei der Verwendung einer Ihrer Methoden den Namen eingegeben oder mit der IntelliSense ausgewählt haben, erscheint eine QuickInfo, die Sie über die Parameterliste informiert.

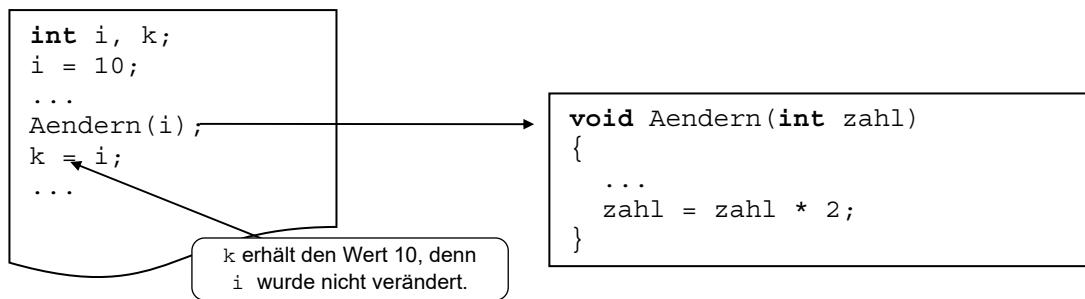
```
Methodname (expression [, . . .]) ;
```

```
PKW.Beschleunigen()
void Fahrzeug.Beschleunigen(int wert)
```

*QuickInfo bei der Anwendung einer Methode*

## Parameter als Wert übergeben

Bei der Parameterübergabe als Wert wird der Methode als Parameter eine **Kopie** des eigentlichen Wertes übergeben. Dieses Vorgehen heißt auch **call by value**. Wenn die Methode den übergebenen Wert verändert, bleibt der ursprüngliche Wert unverändert.



## Beispiel: Auto3.sln

Das Programm über gibt einen Integerwert als Werteparameter an die Methode `Beschleunigen`. Die Methode verändert diesen Wert und weist den neuen Wert dem Feld `Geschwindigkeit` zu.

```
class Fahrzeug
{
 internal int Geschwindigkeit;
 internal void Anzeigen()
 {
 Console.WriteLine("Geschwindigkeit: " + this.Geschwindigkeit);
 }
 ...
① internal void Beschleunigen(int wert)
 {
 ② wert = wert + this.Geschwindigkeit;
 }
}
```

```

③ this.geschwindigkeit = wert;
 }
}
class Program
{
 static void Main(string[] args)
 {
 Fahrzeug PKW = new Fahrzeug();
 PKW.geschwindigkeit = 54;
 PKW.Anzeigen();
④ int zahl = 17;
 Console.WriteLine("Zahl: " + zahl);
⑤ PKW.Beschleunigen(zahl);
⑥ PKW.Anzeigen();
⑦ Console.WriteLine("Zahl: " + zahl);
⑧ PKW.Beschleunigen(5);
⑨ PKW.Anzeigen();
 }
}

```

- ① Die Methode `Beschleunigen()` wird deklariert. Ihr wird beim Aufruf ein Wert vom Typ `int` als Wertekopie übergeben. Die Parametervariable `wert` ist nur innerhalb der Methode gültig.
- ② Zu der als Parameter übergebenen Zahl wird der momentane Geschwindigkeitswert addiert.
- ③ Das Ergebnis wird wieder in dem Feld `geschwindigkeit` gespeichert.
- ④ In der Startmethode `Main()` wird eine Variable mit dem Namen `zahl` deklariert, in der der Wert 17 gespeichert wird.
- ⑤ Die Methode `Beschleunigen()` wird aufgerufen. Die `int`-Variable `zahl` wird als Wert übergeben. Sie steht innerhalb der Methode unter dem Namen `wert` als Kopie der Variablen zur Verfügung. Dadurch wird die Variable `zahl` nicht verändert und behält ihren ursprünglichen Wert.
- ⑥ Der Wert des Feldes `Geschwindigkeit` des Objekts `PKW` wird ausgegeben.
- ⑦ Zur Überprüfung wird auch der Wert der Variablen `zahl` ausgegeben.
- ⑧ Hier erfolgt der Aufruf der Methode `Beschleunigen()` mit einem Literal (Zahlenwert) als Parameter.
- ⑨ Der Wert des Feldes `Geschwindigkeit` wird erneut angezeigt.

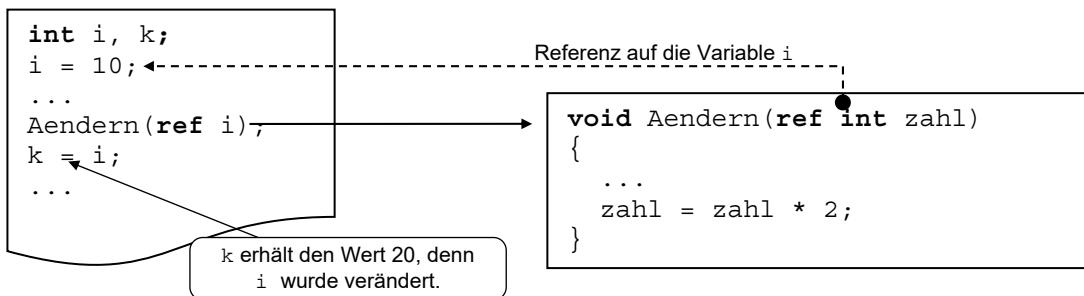
Geschwindigkeit: 54  
 Zahl: 17  
 Geschwindigkeit: 71  
 Zahl: 17  
 Geschwindigkeit: 76

*Die Ausgabe des Programms*

## Parameter als Verweis übergeben

Sie können einer Methode nicht nur eine Kopie der entsprechenden Werte übergeben, sondern auch den direkten Zugriff auf Variablen erlauben. Übergeben Sie dafür als Parameter eine Referenz (einen Verweis) auf diese Variable.

- ✓ Sie kennzeichnen einen Parameter, den Sie als Referenz übergeben möchten, in der Methodendefinition mit dem Schlüsselwort `ref`.
- ✓ Auch bei Aufruf der Methode stellen Sie vor den entsprechenden Parameter das Schlüsselwort `ref`.
- ✓ Die Methode arbeitet dann nicht mit einer Kopie, sondern mit der Variablen selbst (sozusagen einem eigenen Verweis auf die Variable) und kann diese direkt verändern. Dieses Vorgehen heißt auch **call by reference** (Aufruf durch Verweis).



### Beispiel: Auto4.sln

Der Programmcode entspricht weitgehend dem Beispiel *Auto3.sln*; jedoch erfolgt hier die Parameterübergabe als Referenz (`ref`).

- ✓ Die Parameter werden hier nicht als Wert, sondern als **Referenz** übergeben.
- ✓ Weil die Methode `Beschleunigen()` direkt auf die Variablen zugreift, wird in diesem Beispiel der ursprünglich festgelegte Wert der Variablen (17) durch die Methode `Beschleunigen()` verändert.
- ✓ Ein Literal kann nicht als Referenz übergeben werden.

```

void Beschleunigen(ref int wert)
{
 wert = wert + this.geschwindigkeit;
 this.geschwindigkeit = wert;
}

```

```

Geschwindigkeit: 54
Zahl: 17
Geschwindigkeit: 71
Zahl: 71

```

*Die Ausgabe des Programms*

### Methoden mit benannten Parametern

Die Angabe von Parametern bei einem Methodenaufruf musste sich bis zum .NET Framework 4.0 an die bei der Methodendeklaration vorgegebene Reihenfolge halten. Die mit dem .NET Framework 4.0 eingeführten **benannten Parameter** ermöglichen Methodenaufrufe, bei denen die Namen der Parameter vor die Parameterwerte geschrieben werden können. Dadurch ist ein Parameter nicht mehr durch seine Position innerhalb der Parameterliste, sondern durch seinen Namen festgelegt.

Für die bisherige Programmierung bedeutet dies allerdings keinen signifikanten Nutzen. Sie können zwar durch die Namen schneller die Funktionsweise des Parameters erkennen, allerdings kann dies auch zu einer Überfrachtung des Codes führen. Wenn Sie allerdings mittels Ihrer Anwendung die Office-Produkte mittels COM (Common Object Model) ansprechen möchten oder auch andere externe Techniken rund um .NET nutzen wollen, führt dies in einigen Fällen zu einer Vereinfachung.

- ✓ Um benannte Parameter zu nutzen, ändert sich an der Methodendeklaration selbst nichts.
- ✓ Geben Sie beim Aufruf der Methode vor jedem benannten Parameterwert den Namen des Parameters, einen Doppelpunkt und den Parameterwert an, z. B. `PersonAnlegen(name: "Meier", ort: "Leipzig")`.
- ✓ Sie können die benannten Parameter in einer beliebigen Reihenfolge verwenden. Sie müssen immer alle Parameter angeben.
- ✓ Möchten Sie zusätzlich auch unbenannte Parameter verwenden, müssen diese am Anfang stehen, da diese weiterhin durch ihre Position bestimmt sind.

### Beispiel: *ParameterBenennen.sln*

Die Anwendung zeigt die Verwendung von benannten Parametern beim Anlegen einer Person, beispielsweise in einer Personalkartei.

```
class Program
{
 static void Main(string[] args)
 {
 Personal pers = new Personal();
 pers.Anlegen(alter: 25, ort: "Leipzig", name: "Meier");
 }
}

class Personal
{
 private string name;
 private string ort;
 private int alter;

 public void Anlegen(string name, string ort, int alter)
 {
 this.name = name;
 this.ort = ort;
 this.alter = alter;
 }
}
```

- ① Beim Methodenaufruf kann durch die Verwendung von benannten Parametern eine beliebige Reihenfolge verwendet werden.
- ② Die Methodendeklaration unterscheidet sich nicht von der bisherigen Vorgehensweise.
- ③ Die übergebenen Parameter werden hier in privaten Variablen abgelegt.

### Methoden mit optionalen Parametern

Mit dem .NET Framework 4.0 wurde die Möglichkeit eingeführt, Methoden zu deklarieren, die **optionale Parameter** besitzen. Diese sind mit einem Vorgabewert versehen und müssen beim Methodenaufruf nicht zwingend angegeben werden. Obwohl auch diese Erweiterung hauptsächlich auf die Unterstützung der COM-Programmierung oder andere externe Techniken rund um .NET abzielt, kann sie für die Standardprogrammierung ebenfalls von Nutzen sein. Allerdings zahlen Sie unter Umständen einen hohen Preis, denn die Eindeutigkeit von Methoden kann verloren gehen. Der Compiler kann bei überladenen Methoden (siehe Absatz 0) mit der gleichen Anzahl und dem gleichen Typ an verpflichtenden Parametern bei fehlenden optionalen Parametern beim Aufruf die Methoden nicht unterscheiden. In Kapitel 13 verdeutlicht ein Beispiel zu den Parameter-Arrays, welche ein ähnliches Dilemma erzeugen können, diese Problematik.

- ✓ Die Verwendung optionaler Parameter ist in Methoden, Konstruktoren, Delegaten und Indexern erlaubt (vgl. später in diesem Buch).
- ✓ Möchten Sie auch normale Positionsparameter verwenden, müssen diese zu Beginn stehen. Positionsparameter dürfen nicht hinter optionalen Parametern stehen.
- ✓ Optionale Parameter benötigen einen Vorgabewert. Dieser wird ihnen mittels eines Gleichheitszeichens in der Methodendeklaration zugewiesen. Die Werte müssen Literale oder Konstanten sein.
- ✓ Beim Aufruf müssen Positionsparameter vor allen optionalen Parametern stehen, sofern sie verwendet werden.
- ✓ Am Ende können optionale Parameter weggelassen werden. Möchten Sie allerdings optionale Parameter beispielsweise an 3., 5. und 7. Stelle weglassen, müssen Sie benannte Parameter verwenden.

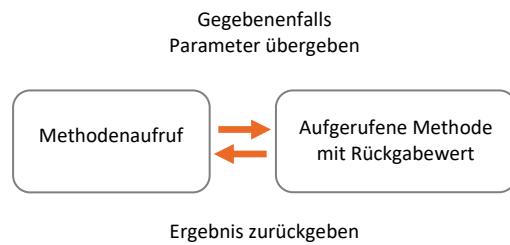
Da bei Ihnen die meisten Mitarbeiter aus dem Raum Leipzig stammen, initialisieren Sie den Ort bereits mit dieser Stadt. Eine Altersangabe ist nicht zwingend notwendig, sodass sie mit 0 vorbelegt wird.

```
public void Anlegen(string name, string ort = "Leipzig", int alter = 0)
{
 this.name = name;
 this.ort = ort;
 this.alter = alter;
}
```

## 8.6 Methoden mit Rückgabewert erstellen

**Methoden** können als Antwort einen Wert an das aufrufende Programm zurückliefern. Dadurch können Sie Methoden mit Rückgabewert auch innerhalb eines Ausdrucks verwenden und z. B. folgendermaßen einsetzen:

- ✓ auf der rechten Seite einer Wertzuweisung,
- ✓ innerhalb eines arithmetischen Ausdrucks (Berechnung),
- ✓ als Vergleichsoperand.



### Syntax für die Deklaration einer Methode mit Rückgabewert

- ✓ Die Syntax zur Deklaration einer Methode mit Rückgabewert unterscheidet sich nur geringfügig von einer Methode ohne Rückgabewert.  
Statt des Schlüsselworts `void` geben Sie den Datentyp des Rückgabewertes an.
- ✓ Mit dem Schlüsselwort `return` wird der nachfolgende Wert (`ReturnValue`) als Ergebnis der Methode zurückgegeben und die Ausführung der Methode wird beendet. Der Rückgabewert kann an einer beliebigen Stelle oder auch mehrmals (z. B. bei Verzweigungen) definiert werden.

```
[modifiers] type Methodname ([parameterlist])
{
 ...
 return ReturnValue;
```

Leider unterbindet Visual Studio in der Standardkonfiguration keinen unerreichbaren (unreachable) Code. Visual Studio meldet in der Standardkonfiguration nur eine Warnung. Damit kann man hinter der `return`-Anweisung weiteren Code in einer Methode notieren, obwohl dieser niemals erreicht wird.

### Methoden mit Rückgabewert anwenden

- ✓ Während Aufrufe von Methoden ohne Rückgabewert (`void`) **immer** eigenständige Anweisungen sind, können Aufrufe von Methoden mit Rückgabewert Ausdrücke oder Teile von Ausdrücken sein.
- ✓ Die IntelliSense und eine QuickInfo unterstützen Sie bei der Eingabe.

```
... Methodname ([parameterlist]) ...
```

**Beispiel: Auto5.sln**

In dem Beispiel wird in der Klasse Fahrzeug die Methode Tanken () hinzugefügt. Sie erhält zwei Parameter: die getankte Benzinmenge (menge) und die Anzahl der gefahrenen Kilometer (gefahrenekilometer). Die Methode soll als Ergebnis vom Typ double den durchschnittlichen Benzinverbrauch je 100 km liefern.

```

class Fahrzeug
{
 ...
① internal double Tanken(double menge, int gefahrenekilometer)
 {
 ② double literPro100km = menge * 100 / gefahrenekilometer;
 ③ return literPro100km;
 }
}

class Program
{
 static void Main(string[] args)
 {
 Fahrzeug PKW = new Fahrzeug();
 ...
 ④ double verbrauch;
 verbrauch = PKW.Tanken(43, 689);
 Console.WriteLine("Verbrauch: {0:f} Liter pro 100 km", verbrauch);
 }
}

```

- ① Der Methode Tanken () wird der Wert der getankten Menge (Liter) und die Anzahl der gefahrenen Kilometer als Parameter übergeben.
- ② Die Methode berechnet den durchschnittlichen Verbrauch je 100 km.
- ③ Der Wert der Variablen literPro100km wird als Ergebnis der Methode zurückgegeben. Die Ausführung der Methode wird beendet.
- ④ In der Methode Main () wird das Ergebnis in der Variablen verbrauch gespeichert und ausgegeben.

Verbrauch: 6,24 Liter pro 100 km

*Die Ausgabe des Programms*

**Wichtige Unterschiede zwischen Methoden mit und ohne Rückgabewert**

| Aktion                                                   | Mit | Ohne                             |
|----------------------------------------------------------|-----|----------------------------------|
| Direkte Ergebnisrückgabe                                 | Ja  | Nur über entsprechende Parameter |
| Verwendung in einer Wertzuweisung                        | Ja  | Nein                             |
| Verwendung in einem arithmetischen Ausdruck              | Ja  | Nein                             |
| Verwendung als Vergleichsoperand in Bedingungsausdrücken | Ja  | Nein                             |

## 8.7 Ausgabeparameter verwenden

Speziell für den Fall, dass eine Methode mehrere Werte zurückliefern soll, gibt es sogenannte **Ausgabeparameter**. Diese Parameter übergeben einer Methode keinen Wert, sondern teilen ihr mit, in welcher Variable sie einen bestimmten Wert speichern soll.

### Syntax für die Verwendung einer Methode mit Ausgabeparametern

```
[modifier] void|type Methodname([parameterlist,] out type identifier [, ...])
{
 identifier = ...
}

... Methodname([parameterlist,] out value [, ...])...
```

- ✓ Deklarieren Sie den Methodenkopf wie gewohnt.
- ✓ Ergänzen Sie vor dem Parameter, der als Ausgabeparameter dienen soll, das Schlüsselwort `out`.
- ✓ Ein `out`-Parameter kann an einer beliebigen Position in der Parameterliste stehen.
- ✓ Eine Methode kann mehrere `out`-Parameter besitzen.
- ✓ Innerhalb der Methode muss der mit `out` gekennzeichneten Variablen (Parameter) ein Wert zugewiesen werden.
- ✓ Beim Aufruf der Methode muss der entsprechende Parameter ebenfalls mit dem Schlüsselwort `out` gekennzeichnet werden.

Statt des Schlüsselworts `out` können Sie auch das Schlüsselwort `ref` verwenden. Allerdings müssen Sie in diesem Fall die als Referenz übergebenen Parameter vor der Übergabe initialisieren. Während sich `out`-Parameter nur zur Ausgabe von Werten **aus einer Methode** eignen, können Sie `ref`-Parameter auch für die Datenübergabe **an die Methode** verwenden. Verwenden Sie das Schlüsselwort `out` vor allem dann, wenn Sie einer Methode eine nicht initialisierte Variable übergeben wollen, die von der Methode mit einem Wert belegt werden soll.

### Beispiel

In diesem Beispiel liefert die Methode `Tanken` die Ergebnisse als Ausgabeparameter zurück.

```
...
internal void Tanken(double menge, double preisJeLiter, int gefahreneKilometer,
 out double kostenJeKm, out double verbrauchJe100Km)
{
 kostenJeKm = menge * preisJeLiter / gefahreneKilometer;
 verbrauchJe100Km = menge * 100 / gefahreneKilometer;
}
...
PKW.Tanken(43, 1.34, 689, out Kosten, out Verbrauch); // Aufruf
...
```

## 8.8 Vordefinierte Methoden nutzen

C# besitzt eine große Anzahl vordefinierter Methoden in den unterschiedlichsten Klassen. Die folgende Tabelle zeigt eine kleine Auswahl.

| Methoden der .NET-Klassenbibliothek                 | Beschreibung                                                                                                             |
|-----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>System.Windows.Forms.MessageBox.Show()</code> | Meldungsfenster anzeigen                                                                                                 |
| <code>new System.Random().Next()</code>             | Zufallszahlen erzeugen über ein Objekt vom Typ <code>System.Random</code>                                                |
| <code>System.Convert()</code>                       | Typumwandlungen                                                                                                          |
| <code>System.Object.ToString()</code>               | Ein beliebiges Objekt in eine Zeichenkette konvertieren; im Falle von Zahlen ist dies z. B. der darin gespeicherte Wert. |
| <code>System.String.ToLower()</code>                | Stringkonvertierung in Kleinbuchstaben                                                                                   |
| <code>System.String.ToUpper()</code>                | Stringkonvertierung in Großbuchstaben                                                                                    |
| <code>System.String.TrimStart()</code>              | Leerzeichen am Anfang eines Strings entfernen                                                                            |
| <code>System.String.TrimEnd()</code>                | Leerzeichen am Ende eines Strings entfernen                                                                              |
| <code>System.Array.GetLowerBound()</code>           | Array-Untergrenze ermitteln                                                                                              |
| <code>System.Array.GetUpperBound()</code>           | Array-Obergrenze ermitteln                                                                                               |

### Spezielle Methoden zur Arbeit mit Zeichenketten

C# stellt viele spezielle Methoden und Eigenschaften zur Bearbeitung von Zeichenketten bereit. Beachten Sie dabei – der Index zum Zugriff auf die Zeichen einer Zeichenkette beginnt grundsätzlich mit 0.

| Name                                   | Beschreibung                                                                                                                                                 | string s, s2;                                                                                                                 |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <code>Copy()</code>                    | Erzeugt eine Kopie s2 einer Zeichenkette s.<br><code>public static string Copy(string s)</code>                                                              | <code>s = "Test";</code><br><code>s2 = String.Copy(s);</code><br><code>s2 =&gt; "Test"</code>                                 |
| <code>Remove()</code>                  | Löscht ab der Position i in einer Zeichenkette s genau c Zeichen.<br><code>public string Remove(int i, int c)</code>                                         | <code>s = "Test";</code><br><code>s = s.Remove(1, 2);</code><br><code>s =&gt; "Tt"</code>                                     |
| <code>Insert()</code>                  | Fügt in einen String s ab der Position i den String s2 ein.<br><code>public string Insert(int i, string s2)</code>                                           | <code>s = "Tt"</code><br><code>s2 = "es"</code><br><code>s = s.Insert(1, s2)</code><br><code>s =&gt; "Test"</code>            |
| <code>IsNullOrEmptyWhiteSpace()</code> | Prüft, ob die als Parameter übergebene Zeichenkette leer ist (d. h. nur aus nicht druckbaren Zeichen bzw. Leerzeichen besteht) oder den Wert null besitzt.   | <code>s = "";</code><br><code>bool b =</code><br><code>String.IsNullOrEmptyWhiteSpace(s);</code><br><code>b =&gt; true</code> |
| <code>Trim()</code>                    | Entfernt alle Leerraumzeichen (Leerzeichen, Tabulatorzeichen, Zeilenumbrüche) am Anfang und am Ende einer Zeichenkette.<br><code>public string Trim()</code> | <code>s = " Test "</code><br><code>s = s.Trim();</code><br><code>s =&gt; "Test"</code>                                        |

| Name        | Beschreibung                                                                                                                                                                                                                | <code>string s, s2;</code>                                                              |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| IndexOf()   | Ermittelt die Position des ersten Auftretens einer Teilzeichenkette <code>part</code> in einem String <code>s</code> . Bei Misserfolg wird <code>-1</code> zurückgegeben.<br><code>public int IndexOf(string part)</code>   | <code>int i;<br/>s = "Test";<br/>s2 = "es";<br/>i = s.IndexOf(s2);<br/>i =&gt; 1</code> |
| ToUpper()   | Wandelt die Buchstaben des Strings <code>s</code> in großgeschriebene Buchstaben um.<br><code>public string ToUpper()</code>                                                                                                | <code>s = "text";<br/>s = s.ToUpper();<br/>s =&gt; "TEXT"</code>                        |
| ToLower()   | Wandelt die Buchstaben des Strings <code>s</code> in klein geschriebene Buchstaben um.<br><code>public string ToLower()</code>                                                                                              | <code>s = "TEXT";<br/>s = s.ToLower();<br/>s =&gt; "text"</code>                        |
| Replace()   | Ersetzt in einem String <code>s</code> alle Vorkommen eines Strings <code>s1</code> durch einen String <code>s2</code> .<br><code>public string Replace(string s1, string s2)</code>                                        | <code>s = "Volltext";<br/>s = s.Replace("Voll", "Kon");<br/>s =&gt; "Kontext"</code>    |
| Substring() | Gibt eine Teilzeichenfolge eines Strings <code>s</code> zurück. Die Teilzeichenfolge beginnt an der Zeichenposition <code>i</code> und hat die Länge <code>j</code> .<br><code>public string Substring(int i, int j)</code> | <code>s = "Volltext";<br/>s2 = s.Substring(4, 4);<br/>s2 =&gt; "text"</code>            |
| Concat()    | Verknüpft mehrere Strings zu einem einzigen String.<br><code>public static string Concat(string s, string s2)</code>                                                                                                        | <code>s = "ABC"; s2 = "DEF";<br/>s = String.Concat(s, s2);<br/>s =&gt; "ABCDEF"</code>  |

## 8.9 Methoden überladen

Innerhalb eines Gültigkeitsbereichs können mehrere Methoden mit demselben Namen (Bezeichner) existieren. Sie müssen jedoch in dem Fall hinsichtlich der Anzahl und/oder des Datentyps der Parameter unterscheiden.

Die Programmierung mehrerer Methoden innerhalb eines Gültigkeitsbereichs mit dem gleichen Namen und unterschiedlichen Parametern wird **Überladen (Overloading)** genannt.

### Syntax für das Überladen einer Methode

- ✓ Alle Überladungen einer Methode verwenden den gleichen Namen bzw. Bezeichner.
- ✓ Die verwendeten Parameter müssen sich in der Anzahl der Parameter oder in mindestens einem Parameter-Datentyp unterscheiden.

```
[modifier] type/void Identifier(parameterlist1)
{
 ...
}

[modifier] type/void Identifier(parameterlist2)
{
 ...
}
```

- ✓ Der Rückgabewert wird beim Überladen nicht zur Unterscheidung herangezogen. Aus diesem Grund können Sie Methoden mit Rückgabewert mit gleichnamigen Methoden ohne Rückgabewert nur dann überladen, wenn Sie sich in der Parameterliste unterscheiden.

**Beispiel: Auto6.sln**

In der Klasse Fahrzeug erhöht bzw. reduziert die Methode `Beschleunigen()` den Wert des Feldes `geschwindigkeit`. Diese Methode wird zweifach mit Parametern unterschiedlicher Datentypen überladen und aufgerufen.

```

class Fahrzeug
{
 internal int geschwindigkeit;
 ...
① internal void Beschleunigen(int wert)
 {
 this.geschwindigkeit += wert;
 }
② internal void Beschleunigen(double faktor)
 {
 double value = this.geschwindigkeit * faktor;
 this.geschwindigkeit = Convert.ToInt32(value);
 }
 ...
}
class Program
{
 static void Main(string[] args)
 {
 Fahrzeug PKW = new Fahrzeug();
 PKW.geschwindigkeit = 54;
 PKW.Anzeigen();
③ PKW.Beschleunigen(17);
 PKW.Anzeigen();
④ PKW.Beschleunigen(1.7);
 PKW.Anzeigen();
 ...
 }
}

```

- ① Die Methode `Beschleunigen()` wird mit einem Parameter vom Typ `int` deklariert. Der übergebene Wert wird zum Wert des Feldes `Geschwindigkeit` **addiert**.
- ② Die Methode `Beschleunigen()` wird hier mit einem Parameter vom Typ `double` deklariert. Der Wert im Feld `geschwindigkeit` wird mit dem übergebenen Faktor **multipliziert**.
- ③/④ Hier wird die Methode mit Parametern unterschiedlichen Datentyps aufgerufen.

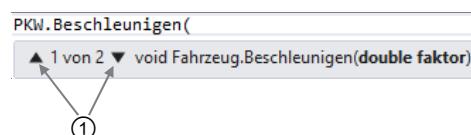
Geschwindigkeit: 54  
Geschwindigkeit: 71  
Geschwindigkeit: 121

*Die Ausgabe des Programms*

Die Überladung spielt in modernen Programmiersprachen eine große Rolle. Die Konsolenausgabe `WriteLine()` des .NET Frameworks hat beispielsweise 19 Überladungen für die unterschiedlichsten Parameterzahlen und -typen. Daher können Sie mithilfe der Methode `WriteLine()` die verschiedensten Datentypen ausgeben, ohne sie konvertieren zu müssen.



Wenn Sie eine überladene Methode anwenden möchten, können Sie in der Parameterinfo die Parameterliste der gewünschten Überladung anzeigen lassen. Wählen Sie die entsprechende Überladung mithilfe der Pfeilschaltflächen ① oder der Pfeiltasten **↑** bzw. **↓**.



## 8.10 Erweiterungsmethoden

Während der Programmierung an einem Projekt kann die Anforderung bestehen, eine bestimmte Klasse um eine oder mehrere Methoden zu erweitern. Dazu muss allerdings der Quellcode der Klasse vorliegen und die Klasse muss erneut übersetzt werden.

Mit Erweiterungsmethoden können Klassen erweitert werden,

- ✓ wenn nicht deren Quellcode vorliegt,
- ✓ wenn sie sich in einer anderen Assembly befinden und auch
- ✓ wenn die Klasse als `sealed` (vgl. Kapitel 11) gekennzeichnet ist.

Der Aufruf der Methoden erfolgt später genauso, wie bei den direkt in der Klasse implementierten Methoden. Sie können jetzt Erweiterungen über neue Assemblies bereitstellen, die Sie beispielsweise später durch die Verwendung einer anderen Assembly austauschen können. Um eine Erweiterungsmethode zu erstellen, gehen Sie folgendermaßen vor:

- ▶ Erstellen Sie eine statische Klasse (vgl. Kapitel 10). Diese kann sich auch in einem anderen Namespace als die zu erweiternde Klasse befinden.
- ▶ Erstellen Sie eine statische Methode (vgl. Kapitel 10) und übergeben Sie ihr als ersten Parameter ein Objekt vom Typ der zu erweiternden Klasse. Dem Parameter wird das Schlüsselwort `this` vorangestellt.

### Beispiel: *Erweiterungsmethoden.sln*

Die nicht ableitbare Klasse `String` der .NET Framework-Klassenbibliothek wird hier durch eine Methode `Reverse()` erweitert, welche die Reihenfolge der Buchstaben umkehrt. Diese Methode wird in einer separaten Klasse `StringExtension` verpackt und in der Methode `Main()` zum Test einmal aufgerufen.

```
class Program
{
 static void Main(string[] args)
 {
 ① Console.WriteLine("Test".Reverse());
 }
}

② public static class StringExtension
{
 ③ public static string Reverse(this string original)
 {
 ④ string reverseString = null;
 ⑤ for(int i = original.Length - 1; i >= 0; i--)
 reverseString = reverseString + original[i];
 ⑥ return reverseString;
 }
}
```

- ① Die neue Methode der Klasse `String` wird aufgerufen.
- ② Eine Erweiterungsmethode muss in einer statischen Klasse verpackt werden.
- ③ Die Erweiterungsmethode `Reverse()` liefert als Ergebnis eine Zeichenkette und übernimmt als Parameter den umzukehrenden String.
- ④ In der Variable `reverseString` wird später die umgekehrte Zeichenkette gespeichert. Da es sich um eine lokale Variable handelt, wird sie hier außerdem noch mit `null` initialisiert.

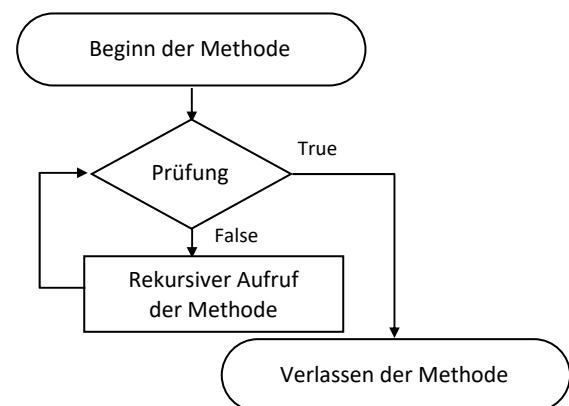
- ⑤ Über eine Schleife, die den originalen String rückwärts durchläuft, wird jeder einzelne Buchstabe an die Variable `reverseString` angefügt.
- ⑥ Das Ergebnis der Methode, die umgekehrte Zeichenkette, wird zurückgegeben.



Erweiterungsmethoden stellen ein mächtiges Instrument dar, um Klassen fehlende Funktionalitäten nachträglich hinzuzufügen. Allerdings kann dies in umfangreichen Klassenbibliotheken auch zu Unübersichtlichkeit und Fehlverhalten führen. Dies kann beispielsweise der Fall sein, wenn die Klasse bereits über eine Methode mit der gleichen Signatur wie eine Erweiterungsmethode verfügt (allerdings ohne den Zusatz `static` und den ersten Parameter). In diesem Fall wird kein Fehler produziert, sondern es wird der Aufruf der in der Klasse bereits vorhandenen Methode durchgeführt.

## 8.11 Rekursion

Methoden können sich selbst mit jeweils anderen Parametern aufrufen. Dies wird als direkte Rekursion bezeichnet. Die Rekursion stellt in einigen Fällen eine Alternative zur Verwendung von Schleifen (Iteration) dar. Analog zu den Schleifen darf der rekursive Aufruf nur dann durchgeführt werden, wenn eine bestimmte Bedingung erfüllt bzw. ein Abbruchkriterium nicht erfüllt ist.



### Beispiel: `Teiler.sln`

Die Aufgabe des folgenden Programms besteht darin, den größten gemeinsamen Teiler zweier natürlicher Zahlen über den Algorithmus von Euklid zu ermitteln.

- ✓ Der Methode `GGTeiler()` werden zwei Zahlen übergeben.
- ✓ Die Methode teilt die erste Zahl ① durch die zweite Zahl ②.
- ✓ Bleibt bei der Division ein Rest ③, dann wird die Methode erneut aufgerufen und dabei der bisherige Divisor als neuer Dividend (1. Zahl ④) und der Rest als neuer Divisor (2. Zahl ⑤) verwendet.
- ✓ Die Methode `GGTeiler()` ruft sich immer wieder selbst auf, bis bei der Division kein Rest mehr entsteht (spätestens bei der Division durch 1).
- ✓ Bleibt kein Divisionsrest, dann ist der Quotient dieser Division der gesuchte größte gemeinsame Teiler der ursprünglichen Zahlen ① und ②.
- ✓ Bei jedem Aufruf wird als erste Zahl die zweite Zahl des letzten Aufrufs und als zweite Zahl der Rest der Division übergeben.

| Dividend                            | Divisor | Quotient | Rest     |
|-------------------------------------|---------|----------|----------|
| 75 ①                                | 54 ②    | 1        | 21 ③ > 0 |
| 54 ④                                | 21 ⑤    | 2        | 12 > 0   |
| 21                                  | 12      | 1        | 9 > 0    |
| 12                                  | 9       | 1        | 3 > 0    |
| 9                                   | 3       | 3        | 0 = 0    |
| Der größte gemeinsame Teiler ist 3. |         |          |          |

```

class Zahlen
{
 public int GGTeiler(int z1, int z2)
 {
 Console.WriteLine("GGTeiler({0}, {1})", z1, z2);
 if (z2 == 0)
 return z1;
 else
 }
}

```

```

⑤ return GGTeiler(z2, z1 % z2);
}
}
class Program
{
 static void Main(string[] args)
 {
 int zahl1, zahl2;
 Zahlen zahlenObjekt = new Zahlen();
 Console.WriteLine("Bitte eine ganze Zahl eingeben: ");
 zahl1 = Convert.ToInt32(Console.ReadLine());
 Console.WriteLine("Bitte noch eine ganze Zahl eingeben: ");
 zahl2 = Convert.ToInt32(System.Console.ReadLine());
 if ((zahl1 != 0) && (zahl2 != 0))
 Console.WriteLine(zahlenObjekt.GGTeiler(zahl1, zahl2));
 Console.WriteLine(" ist der grösste gemeinsame Teiler.");
 }
}

```

- ① In einer Klasse mit dem Namen `Zahlen` wird die Methode `GGTeiler()` mit zwei Werteparametern vom Typ `int` deklariert.
- ② Zur Kontrolle wird jeder Methodenaufruf mit einer Ausgabeanweisung dokumentiert.
- ③ Wenn bei der letzten Division kein Rest mehr entstanden ist, dann wurde der größte gemeinsame Teiler gefunden.
- ④ Der größte gemeinsame Teiler wird zurückgegeben. Mit dieser Anweisung bricht die Rekursion ab. Die Methode ruft sich nicht mehr erneut auf.
- ⑤ Wenn bei der letzten Division ein Rest entstanden ist, ruft sich die Methode selbst auf (rekursiv). Dabei wird als erster Parameter die bisherige zweite Zahl verwendet. Als zweiter Parameter wird der Ausdruck der Division und somit als Ergebnis dieses Ausdrucks der Divisionsrest übergeben.
- ⑥ Im Hauptprogramm werden zunächst die beiden Zahlen eingegeben und eingelesen.
- ⑦ Wenn beide Zahlen ungleich 0 sind, erfolgt der erste Methodenaufruf ⑧.

```

Bitte eine ganze Zahl eingeben: 75
Bitte noch eine ganze Zahl eingeben: 54
GGTeiler (75, 54)
GGTeiler (54, 21)
GGTeiler (21, 12)
GGTeiler (12, 9)
GGTeiler (9, 3)
3 ist der grösste gemeinsame Teiler.

```

*Die Ausgabe des Programms*

### Hinweise zum Einsatz rekursiver Methoden

Eine Rekursion kann einen hohen Speicherbedarf verursachen, der auch zum Programmabsturz führen kann. Die Rücksprungadressen und lokalen Variablen aller durchlaufenen Methodenaufrufe werden im sogenannten Stapspeicher (Stack) des Computers abgelegt, um die Methoden nach dem Beenden der Rekursion in umgekehrter Reihenfolge wieder durchlaufen zu können. Es ist sinnvoll, sich den Einsatz einer Rekursion genau zu überlegen und sie zu testen sowie die Aufruftiefe definiert zu begrenzen. In vielen Fällen lassen sich die Aufgaben auch mit iterativen Strukturen (Schleifen) lösen, die nicht so viel Speicher benötigen.

## 8.12 Übungen

### Übung 1: Parameterübergabe an eine Methode

**Übungsdatei:** --

**Ergebnisdatei:** SwapText.sln

1. Erstellen Sie eine Konsolenanwendung, die eine Klasse `Txt` enthält. Schreiben Sie in dieser Klasse eine Methode, die zwei Variableninhalte vom Typ `string` vertauscht. Verwenden Sie die Parameterübergabe per Referenz.
2. Erzeugen Sie in der Methode `Main()` ein Objekt der Klasse `Txt`, um die Methode zu testen.

### Übung 2: Methode für Primzahltest

**Übungsdatei:** --

**Ergebnisdatei:** Primzahlen.sln

1. Schreiben Sie eine Klasse `Zahl`, die nur die eine Methode `IsPrime()` enthalten soll. Diese Methode prüft einen übergebenen `int`-Wert daraufhin, ob es sich um eine Primzahl handelt. Primzahlen sind nur durch sich selbst und durch 1 teilbar. Die Methode soll den Rückgabetyp `bool` besitzen. Sie gibt `true` zurück, wenn es sich um eine Primzahl handelt, ansonsten `false`.
2. Testen Sie die Methode in einer Konsolenanwendung. Nach der Eingabe eines Zahlenbereichs sollen alle Zahlen dieses Bereichs in einer Schleife durchlaufen werden. Darin werden sie mithilfe der Primzahl-Methode getestet. Im Falle des Rückgabewertes `true` wird die Primzahl auf dem Bildschirm ausgegeben.

**Lösungshinweis:** Für einen simplen Primzahltest reicht es aus, die betreffende Zahl durch alle Zahlen von 2 bis Zahl geteilt durch 2 zu teilen und zu prüfen ob der Rest 0 ist. In diesem Fall würde es sich nicht um eine Primzahl handeln.

```
for (int i = 2; i <=(zahl / 2) + 1; i++)
 if (zahl % i ==0)
 ...

```

*Beispiel für einen Primzahltest*

**Hinweis:** Beachten Sie bei der Eingabe des Zahlenbereichs, dass sehr große Zahlen möglicherweise nicht dem Wertebereich der verwendeten Variablen entsprechen und dadurch zu einem Fehler führen. Testen Sie die Anwendung beispielsweise mit dem Zahlenbereich 10...500.

### Übung 3: Überladen einer Methode

**Übungsdatei:** --

**Ergebnisdatei:** Ueberladen.sln

1. Erstellen Sie in einem Konsolenprogramm eine Klasse `Wert`. Deklarieren Sie in dieser Klasse überladene Methoden, die einen `int`- bzw. einen `double`-Wert oder einen `string` als Parameter entgegennehmen und eine entsprechende Ausgabe durchführen:
  - ✓ Die Ganzzahl lautet: *übergebene Zahl*
  - ✓ Die Dezimalzahl lautet: *übergebener Wert*
  - ✓ Der Text lautet: *übergebener Text*
2. Testen Sie die Methoden, indem Sie sie je einmal mit einem Wert der drei verschiedenen Datentypen aufrufen.

## Übung 4: Ganzzahl-Taschenrechner objektorientiert

**Übungsdatei:** Kap07\IntegerCalc\IntegerCalc.sln

**Ergebnisdatei:** IntegerCalcOOP.sln

1. Verändern Sie den Taschenrechner aus der Übung des Kapitels 7.
2. Erstellen Sie in einer separaten Datei eine Klasse Rechner. Die Klasse erhält zwei als public deklarierte Felder zur Speicherung der beiden Operanden (int).
3. Ergänzen Sie in der Klasse Rechner fünf Methoden für die verschiedenen mathematischen Operationen, die jeweils das Ergebnis zurückgeben.
4. Erzeugen Sie in der Formular-Klasse eine Instanz rechenoperationen der Klasse Rechner. Schreiben Sie eine Initialisierungsmethode init (), die die Textfelder ausliest, in Zahlenwerte konvertiert und in den Feldern des Objekts rechenoperationen speichert. Erstellen Sie außerdem eine Methode anzeigen (), die ein als Parameter übergebene Ergebnis (int) in dem Textfeld Ergebnis anzeigt.
5. Schreiben Sie die Ereignisbehandlungsmethoden für das Click-Ereignis der Schaltflächen, in denen Sie die Initialisierung durchführen, die entsprechende Methode zur Berechnung aufrufen und das Ergebnis anzeigen lassen.



**Hinweis:** Achten Sie bei der Eingabe von Werten in die Textfelder *Zahl 1* und *Zahl 2* darauf, dass der Wertebereich eingehalten wird. Sehr große Zahlen sowie Texteingabe in die Felder führen zu einem Programmabbruch.

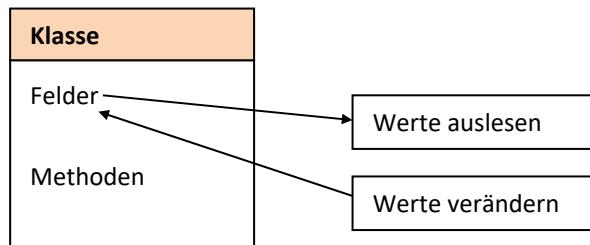
# 9 Kapselung, Konstruktoren und Namensräume

## 9.1 Kapselung

### Zugriff auf die Felder kontrollieren

Eine Klasse vereinigt die Daten (**Felder** bzw. Attribute oder Eigenschaften) und die Funktionalität (**Methoden**) in einer einzigen Struktur.

In den vorherigen Beispielen konnten Methoden anderer Klassen (beispielsweise die Methode `Main()`) auf die Felder direkt zugreifen und die Werte auslesen und verändern.

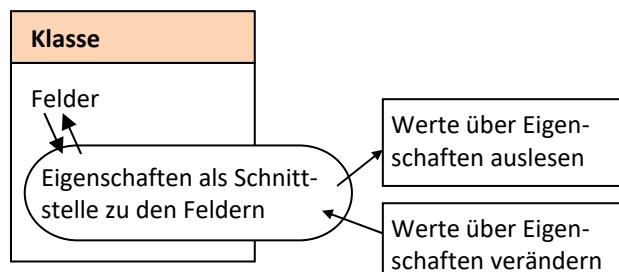


*Felder im direkten Zugriff von „außen“*

### Was ist Kapselung?

Bei der **Kapselung** werden die Felder/Attribute wie in einer Kapsel in der Klasse eingeschlossen und „nach außen abgeschirmt“. Nur sogenannte **Eigenschaften** (wie sie im Microsoft-Kontext gesehen werden) und die Methoden der Klasse selbst haben Zugriff auf die Felder. Auf diese Weise lässt sich der Zugriff auf die Felder kontrollieren.

- ✓ Werte können geprüft und verändert werden, bevor sie in den Feldern gespeichert werden.
- ✓ Die Werte der Felder lassen sich aufbereiten und verändern, bevor sie weitergegeben werden.



*Kontrollierter Zugriff auf gekapselte Felder*

## 9.2 Eigenschaften

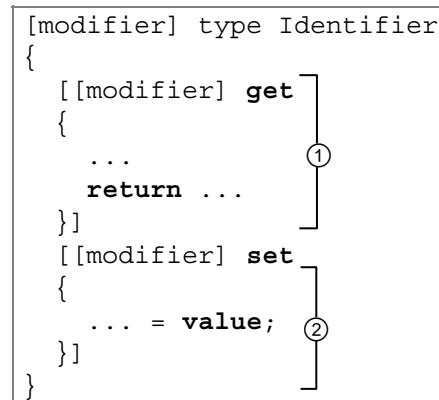
In einigen Programmierkonzepten differenziert man zwischen Feldern/Attributen und Eigenschaften. So auch im Umfeld von .NET, wie schon an diverseren Stellen erwähnt wurde. Felder und Attribute sind in diesen Konzepten von außen nicht zugänglich, sondern nur indirekt. Um die dafür notwendige Kapselung der Felder zu erreichen, werden die Felder bei der Klassendeklaration mit dem Zugriffsmodifizierer `private` vereinbart. Der exklusive Zugriff auf ein Feld wird über eine sogenannte **Eigenschaft** gesteuert, die normalerweise als `public` deklariert wird.



Beachten Sie, dass in den meisten objektorientierten Sprachen bzw. theoretischen Konzepten der OOP bereits von außerhalb der Klasse zugängliche Felder bzw. Attribute als Eigenschaften bezeichnet werden. Die offizielle Definition der OOP setzt also die Begriffe **Feld**, **Attribut**, **Objektvariable**, **Instanzvariable**, **Datenelement** und **Eigenschaft** meist gleich. Die eingeschränkte Verwendung eines Feldes und das spezielle Verständnis von Eigenschaften als indirekten Zugriffsweg auf Felder in der Microsoft-Notation ist vermutlich durch die starke Verbindung von Visual Studio als IDE, weiterer Technologien des .NET-Frameworks und der eigentlichen Programmiersprachen entstanden. Aber auch in einigen weiteren Programmiersprachen gibt es so eine Differenzierung.

## Syntax für die Deklaration von Eigenschaften

- ✓ Bei Bedarf kann ein Zugriffsmodifizierer vorangestellt werden. Ohne Angabe werden Eigenschaften als `private` vereinbart und ein Zugriff über eine Methode außerhalb der Klasse ist nicht möglich.
- ✓ Danach folgen der Typ der Eigenschaft und der Eigenschaftsname (`Identifier`), der sich an die Konventionen für Bezeichner halten muss.
- ✓ Es folgen in geschweiften Klammern die Blöcke `get` ① und `set` ②, die den eigentlichen Zugriff auf das entsprechende Feld regeln.
- ✓ Eigenschaften, die nur einen Lesezugriff ermöglichen sollen, erhalten nur den `get`-Block.
- ✓ Eigenschaften, die nur einen Schreibzugriff zulassen sollen, erhalten nur den `set`-Block.
- ✓ Für den `set`- und den `get`-Block lassen sich mithilfe der Modifizierer die Zugriffsrechte für den jeweiligen Block einschränken. So können Sie z. B. bestimmen, dass die Werte zwar von allen Stellen gelesen (`public`), aber nur aus der eigenen Klasse und allen erbenden Klassen geschrieben werden können (`protected`).
- ✓ Wenn Sie für den `set`- bzw. den `get`-Block keinen speziellen Modifizierer angeben, gelten die Zugriffsrechte, die Sie für die Eigenschaft festgelegt haben.
- ✓ Im `set`- und `get`-Block werden in geschweiften Klammern die Anweisungen integriert, mit denen beispielsweise die ein- und auszugebenden Daten kontrolliert und behandelt werden. Dazu können auch weitere Methoden aufgerufen werden.
- ✓ Im `set`-Block steht automatisch eine kontextabhängige lokale Variable `value` zur Verfügung, die den an die Eigenschaft zugewiesenen Wert enthält.



## Bereits deklarierte Felder kapseln

Haben Sie bereits ein Feld beispielsweise als `public` deklariert ①, können Sie dieses Feld nachträglich kapseln und eine Eigenschaft für den kontrollierten Zugriff bereitstellen. Den Befehl finden Sie im Menü *Bearbeiten* unter *Umgestalten* oder einem Smarttag, der nur auftaucht, wenn Sie in der Deklaration eines Feldes stehen und Visual Studio eine Kapselung als sinnvoll erachtet. In dem Fall gehen Sie wie folgt vor:

- Klicken Sie am linken Rand des Editors auf das Smarttag-Symbol mit der Lampe.
- Wählen Sie im Dropdown-Menü den Befehl zum Kapseln des Feldes.

Wenn der Feldname mit einem Kleinbuchstaben beginnt, schlägt Visual Studio automatisch den gleichen Namen beginnend mit einem Großbuchstaben vor. Beginnt er mit einem Großbuchstaben, hängt Visual Studio an den Namen eine durchlaufende Nummer an.



```
static readonly String b = "Die Antwort ist";
public static string B
{
 get
 {
 return b;
 }
}
```

*Feld kapseln und Eigenschaft erzeugen*

Visual Studio ändert in der Regel die Zugriffsrechte des Feldes und generiert automatisch die Programmzeilen zur Deklaration der Eigenschaft.

**Beispiel: Fahrzeuge.sln**

Im folgenden Programm wird nicht mehr direkt auf ein Feld der Klasse zugegriffen, sondern über eine Eigenschaft. Beim Schreibzugriff wird der zu speichernde Geschwindigkeitswert auf den Wert 200 begrenzt.

```

class Fahrzeug
{
 ① private int aktuelleGeschwindigkeit;

 ② public int Geschwindigkeit
 {

 ③ get
 {
 ④ return this.aktuelleGeschwindigkeit;
 }

 ⑤ set
 {
 ⑥ if (value > 200)
 value = 200;
 ⑦ this.aktuelleGeschwindigkeit = value;
 }
 }

 class Program
 {
 static void Main(string[] args)
 {
 ⑧ Fahrzeug PKW = new Fahrzeug();
 ⑨ PKW.Geschwindigkeit = 54;
 ⑩ Console.WriteLine("Geschwindigkeit: {0} km/h", PKW.Geschwindigkeit);
 }
 }
}

```

- ① Das Feld `aktuelleGeschwindigkeit` wird als `private` deklariert und ist somit in der Klasse `Fahrzeug` gekapselt.
- ② Über die Eigenschaft `Geschwindigkeit` werden sowohl der Lese- als auch der Schreibzugriff auf das Feld `aktuelleGeschwindigkeit` gesteuert.
- ③ Mithilfe des `get`-Blocks wird der Wert des Feldes `aktuelleGeschwindigkeit` gelesen und mit `return` an die aufrufende Anweisung zurückgegeben.
- ④/⑦ Zur besseren Übersicht wird empfohlen, innerhalb der Klasse auf die **eigenen** Felder mit `this.` zuzugreifen.
- ⑤ Im `set`-Block wird ein zugewiesener Wert in einer automatisch erzeugten lokalen Variablen mit dem Namen `value` bereitgestellt.
- ⑥ Der übergebene Wert `value` wird geprüft, ggf. auf den Wert 200 begrenzt und dem Feld `aktuelleGeschwindigkeit` zugewiesen.
- ⑧ Eine Instanz (ein Objekt) der Klasse `Fahrzeug` wird erzeugt und der Objektvariablen `PKW` zugewiesen.
- ⑨ Aus der Klasse `Program` erfolgt der Schreibzugriff auf das Feld `aktuelleGeschwindigkeit` über die Eigenschaft `Geschwindigkeit`. Auf das Feld `aktuelleGeschwindigkeit` kann aus der Methode `Main()` nicht zugegriffen werden, da das Feld in der Klasse `Program` mit `private` gekapselt ist.
- ⑩ An dieser Stelle wird über die Eigenschaft `Geschwindigkeit` **lesend** auf das Feld `aktuelleGeschwindigkeit` zugegriffen und der Wert ausgegeben.

Beachten Sie, dass der Name des Feldes und der Eigenschaft sich in der Praxis meist nur dadurch unterscheiden, dass das Feld kleingeschrieben und die zugehörige Eigenschaft großgeschrieben wird. So wird es auch durch Visual Studio automatisch gemacht, wenn Sie dessen Features zum Generieren von Codestrukturen nutzen. Aber das ist nicht zwingend, wie Sie anhand des Beispiels sehen. Man kann durchaus Felder und Eigenschaften nicht nur durch Klein- und Großschreibung unterscheiden. Dennoch – in der Praxis sollte man dies nicht tun. Diese Empfehlung sollte schon deshalb überzeugend sein, weil die reine Unterscheidung von Feldern und den zugeordneten Eigenschaften durch Groß- und Kleinschreibung in Visual Studio als Automatismus Einzug gehalten hat.

## Automatisch implementierte Eigenschaften

Häufig besitzen Eigenschaften eine sehr einfache Implementierung. Im `get`-Block wird lediglich ein Variablenwert gelesen, der im `set`-Block direkt geschrieben wird.

```
// Eine Eigenschaft, welche einen Namen verwaltet
private string name;
public string Name
{
 get { return name; }
 set { name = value; }
}
```

Diese einfache Programmlogik kann seit dem .NET Framework 3.5 auch der Compiler über eine vereinfachte Schreibweise erzeugen. Er erzeugt dazu eine private, anonyme Variable zur Aufnahme des Eigenschaftswertes und die beiden `get`- und `set`-Zugriffe, wie sie im Code-Abschnitt oben zu sehen sind. Auf die bereitgestellte anonyme Variable kann von Ihnen nicht direkt zugegriffen werden. Die `get`- und `set`-Blöcke sind automatisch `public`. Es müssen immer beide Blöcke (`get` und `set`) angegeben werden. Allerdings können Sie vor dem `set`-Block mittels der Angabe von `private` eine nur lesbare Eigenschaft erzeugen.

```
public string Name { get; set; }
```

Eine solche Syntax hat Visual Studio in früheren Versionen automatisch generiert, wenn man über den Umgestalten-Befehl ein Feld gekapselt hat.

In neuen Versionen von C# kann man bei der Formulierung von Eigenschaften auch sogenannte **Lambda-Ausdrücke** verwenden, die neue Varianten von Visual Studio auch standardmäßig generieren, wenn man aus einem Feld durch die IDE eine Eigenschaft generieren lässt. Diese Varianten einer Eigenschaft sehen dann so aus:

```
public string Name { get => name; set => name = value; }
```

Lambda-Ausdrücke werden in dem Buch aber nur angedeutet, denn sie werden hauptsächlich in der fortgeschrittenen Programmierung (etwa der Ereignisbehandlung mit Delegates, generischen Ausdrücken, Umgang mit LINQ, asynchronen Prozessen etc), die jenseits des Scopes von dem Buch liegen, verwendet. Ein Lambda-Ausdruck erlaubt in gewissen Situationen eine verkürzte Schreibweise und verwendet den Lambda-deklarationsoperator `=>`, um eine Parameterliste des Lambdas von dessen Implementierung zu trennen. Diese Implementierung ist ein üblicher C#-Ausdruck oder -Anweisungsblock. Ein Lambdaausdruck kann auch einen Rückgabewert liefern, wie Sie an dem `get`-Block der Eigenschaft erkennen können.

C# bzw. .NET allgemein entwickelt sich permanent weiter und mittlerweile kann man Eigenschaften sogar ganz ohne zugrundeliegende Felder verwenden. Das wird von Visual Studio auch bei der automatischen Codegenerierung angeboten. Das bedeutet, dass es zu einer Eigenschaft wie `Vorname` nicht zwingend ein Feld `vorname` (oder ein anders benanntes Feld) geben muss, sondern die Eigenschaft „alleine“ existiert.

## 9.3 Konstruktoren und Destruktoren

### Was sind Konstruktoren und Destruktoren?

- ✓ Konstruktoren und Destruktoren sind spezielle Methoden einer Klasse.
- ✓ Ein **Konstruktor** wird automatisch beim **Erzeugen** einer Instanz aufgerufen. Wichtig – der Konstruktor erzeugt nicht selbst das Objekt. Die Betonung liegt auf **Aufgerufen**.
- ✓ Ein **Destruktor** wird automatisch beim **Zerstören** einer Instanz aufgerufen. Wie beim Konstruktor gilt auch bei dem Destruktor – die Betonung liegt auf **Aufgerufen**.
- ✓ Sie können Konstruktoren und Destruktoren nicht selbst aufrufen.

### Standardkonstruktor

C# hält standardmäßig für jede (normale<sup>1</sup>) Klasse, die Sie definieren, einen parameterlosen Standardkonstruktor bereit. Dieser Konstruktor wird aufgerufen, wenn Sie mit `new` ein neues Objekt vom Typ dieser Klasse erzeugen.

Bei der Erzeugung eines Objekts mit `new` wird der entsprechende Speicherplatz reserviert. Dabei werden die Felder des Objektes je nach Typ mit `0`, `null` oder `false` initialisiert.

### Einen individuellen Konstruktor erstellen

In vielen Fällen ist es notwendig, dass beispielsweise spezielle Initialisierungen durchgeführt werden, wenn ein Objekt erzeugt wird. Dazu programmieren Sie einen individuellen Konstruktor, der als Anweisungen die gewünschten Initialisierungen enthält.

- ✓ Der Konstruktor wird wie eine Methode ohne Rückgabewert und ohne die Angabe des Schlüsselwortes `void` deklariert.
  - ✓ Statt des Methodennamens schreiben Sie den Klassennamen.
  - ✓ Dem Konstruktor können Werte übergeben werden (`parameterlist`), mit denen sich beispielsweise die Felder initialisieren lassen.
  - ✓ Sobald Sie einen Konstruktor erstellen, ist der parameterlose Standardkonstruktor nicht mehr verfügbar.
  - ✓ Sie können den Konstruktor mehrfach überladen, um bei der Erzeugung Parameter unterschiedlicher Anzahl oder unterschiedlichen Typs übergeben zu können. So können Sie beispielsweise auch wieder einen parameterlosen Konstruktor deklarieren.
- ```
[modifier] Classname (parameterlist)
{
    ...
}
```



Sie sollten möglichst auch in den Konstruktoren die Initialisierung der Felder über Eigenschaften vornehmen. Dadurch können Sie die in den `set`-Blöcken der Eigenschaften bereits programmierten Überprüfungen und Modifikationen der zu speichernden Werte nutzen. Es kann allerdings Ausnahmefälle geben, bei denen die in den Eigenschaften gesetzten Schritte beim Erzeugen explizit **nicht** gewünscht oder sinnvoll sind. Dann können Sie auch direkt die Felder initialisieren. Aber im Regelfall ist die Verwendung von Eigenschaften zu empfehlen.

¹ Zu Details kommen wir an anderer Stelle.

Beispiel: Fahrzeuge1.sln

Das folgende Programm enthält das Beispiel *Fahrzeuge.sln* in abgewandelter Form. Bereits bei der Erzeugung eines Objekts kann das Feld `aktuelleGeschwindigkeit` mit einem bestimmten Wert initialisiert werden. Es ist sinnvoll, auch im Konstruktor soweit möglich vorhandene Eigenschaften zu nutzen. Da in den Eigenschaften gewünschte Prüfungen und Modifikationen durchgeführt werden, müssen diese so nicht erneut programmiert werden.

```
class Fahrzeug
{
    private int geschwindigkeit;

    public int Geschwindigkeit
    {
        get ...
        set ...
    }

①   public Fahrzeug(int wert)
    {
        this.Geschwindigkeit = wert;
    }

②   public Fahrzeug()
    {
        this.Geschwindigkeit = 50;
    }
}

class Program
{
    static void Main(string[] args)
    {

③       Fahrzeug PKW1 = new Fahrzeug();
④       Fahrzeug PKW2 = new Fahrzeug(38);
        Console.WriteLine("Geschwindigkeit 1: {0} km/h",
PKW1.Geschwindigkeit);
        Console.WriteLine("Geschwindigkeit 2: {0} km/h",
PKW2.Geschwindigkeit);
    }
}
```

- ① Innerhalb der Klassendeklaration wird ein Konstruktor deklariert. Er weist den Wert des übernommenen Parameters `wert` der Eigenschaft `Geschwindigkeit` zu.
- ② Ein zweiter parameterloser Konstruktor gibt für die Eigenschaft einen Standardwert vor (hier 50).
- ③ Eine Instanz `PKW1` wird ohne Parameter erzeugt. Somit wird der parameterlose Konstruktor aufgerufen und die Eigenschaft `Geschwindigkeit` erhält den Wert 50.
- ④ Die Instanz `PKW2` wird mit einem Parameter erzeugt. Mit dem übergebenen Wert wird die Geschwindigkeit initialisiert.

Geschwindigkeit 1: 50 km/h
Geschwindigkeit 2: 38 km/h

Die Ausgabe des Programms

Überladenen Konstruktor derselben Klasse nutzen

Über die Referenzvariable `this` kann in einem Konstruktor ein anderer überladener Konstruktor derselben Klasse aufgerufen werden. Dieser Aufruf wird vor allen anderen Anweisungen des Konstruktors ausgeführt. Auf diese Weise können Sie in der Regel den Initialisierungscode in einem einzigen Konstruktor unterbringen. Dieser wird über die anderen Konstruktoren mit den entsprechenden Parametern aufgerufen.

- ✓ Der Aufruf des Konstruktors ① wird mit einem Doppelpunkt von der Deklaration des Konstruktors ② abgetrennt und üblicherweise eingerückt in eine neue Zeile geschrieben.
- ✓ Der Aufruf erfolgt über die Referenzvariable `this` und die entsprechende Parameterliste, die in Klammern geschrieben wird.
- ✓ Die runden Klammern müssen auch gesetzt werden, wenn keine Parameter übergeben werden.

```
...
public Fahrzeug(int wert)
{
    this.Geschwindigkeit = wert;
}

public Fahrzeug()      ②
    : this(50)          ①
{
}
...
```

Objektinitialisierer

Viele Klassen besitzen zahlreiche Eigenschaften und Felder, die an geeigneter Stelle initialisiert werden müssen. Außerdem kann zusätzlich die Anforderung bestehen, unterschiedliche Initialisierungen dieser Eigenschaften und Felder zu ermöglichen. Bisher mussten Sie mehrere Konstruktoren bereitstellen, die diese Member auf unterschiedlichste Art initialisieren. Um dies zu vermeiden, wurden im .NET Framework 3.5 Objektinitialisierer eingeführt.

Syntax für die Deklaration von Objektinitialisierern

- ✓ Einem Konstruktor werden zusätzliche Parameter in geschweiften Klammern angefügt.
- ✓ Die Angabe erfolgt in einer Name=Wert-Schreibweise, wobei mehrere Paare durch Komma getrennt werden.
- ✓ Der Name entspricht dabei dem Namen einer öffentlichen Variablen oder Eigenschaft, der Wert wird diesem Feld bzw. der Eigenschaft zugewiesen.
- ✓ Wenn Sie später bei der Objekterstellung den parameterlosen Konstruktor nutzen wollen, können Sie auf die Angabe des runden Klammernpaars verzichten. Der Konstruktor selbst muss allerdings existieren.

```
Type var = new Type(Parameter) {Name2=Wert2, Name2=Wert2, ...}
// oder
Type var = new Type{Name2=Wert2, Name2=Wert2, ...}
```

Falls Sie dieselbe Eigenschaft oder Variable über den Konstruktor **und** den Objektinitialisierer initialisieren, wird zuerst der Konstruktor verwendet und dann der Objektinitialisierer.

Beispiel: `AutoInit.sln`

Die Klasse `Auto` stellt zwei Möglichkeiten zur Verfügung, die Eigenschaft `Marke` zu initialisieren: über einen Konstruktor und über die Objektinitialisierung. Zur Verwendung der Objektinitialisierung ohne Angabe der runden Klammern muss der parameterlose Konstruktor bereitgestellt werden.

```

using System;
namespace AutoInit
{
    class Program
    {
        static void Main(string[] args)
        {
            ①     Auto bmw = new Auto { Marke = "BMW" };
            ②     Auto audi = new Auto("Audi");

            Console.WriteLine(bmw.Marke);
            Console.WriteLine(audi.Marke);
        }
    }

    public class Auto
    {
        ③     private string marke;
        ④     public string Marke
        {
            get { return marke; }
            set { marke = value; }
        }
        ⑤     public Auto()
        {
        }
        ⑥     public Auto(string marke)
        {
            this.marke = marke;
        }
    }
}

```

- ① Die erste Instanz von `Auto` wird mittels der Objektinitialisierung erzeugt. Der öffentlichen Eigenschaft `Marke` wird dadurch der Wert `BMW` zugewiesen. Der Vorteil an dieser Stelle ist nun, dass Sie beliebig viele Initialisierungen vornehmen können. Dabei spielt es weder eine Rolle, welche Felder und Eigenschaften Sie initialisieren, noch, in welcher Reihenfolge.
- ② Die zweite Initialisierung erfolgt über den Konstruktor.
- ③ Die private Variable `marke` speichert intern den Wert der Eigenschaft `Marke`.
- ④ Die öffentliche Eigenschaft `Marke` stellt den kontrollierten Zugriff auf die Variable `marke` bereit.
- ⑤ Da es einen parametrisierten Konstruktor ⑥ gibt, muss auch der parameterlose Konstruktor bereitgestellt werden, um die Objektinitialisierung, wie in ① verwendet, durchzuführen.

Destruktoren

Wenn ein Objekt nicht mehr benötigt wird, beispielsweise weil keine Objektvariable mehr auf diese Instanz verweist, wird das Objekt automatisch aus dem Speicher entfernt. Diese Aufgabe übernimmt die sogenannte Garbage Collection.

Manchmal sollen bestimmte Arbeiten erledigt werden, bevor ein Objekt zerstört wird. Dazu schreiben Sie einen sogenannten Destruktor. Ein Destruktor ist das Gegenstück zum Konstruktor.

Sie möchten beispielsweise die geöffneten Fenster einer Anwendung zählen. Im Konstruktor der Fensterklasse wird der Zähler jeweils beim Öffnen eines Fensters erhöht. Im Destruktor wird vor dem Zerstören des Objekts (Fensters) der Zähler wieder um 1 verringert.

- ✓ Einen Destruktor deklarieren Sie wie einen Konstruktor mit dem Namen der Klasse.
 - ✓ Zur Kennzeichnung als Destruktor schreiben Sie vor den Namen eine Tilde .
 - ✓ Ein Destruktor besitzt keine Parameter.
 - ✓ Ein Destruktor kann nicht direkt aufgerufen werden und erhält keine Zugriffsmodifizierer.
- ```
...
~Fahrzeug()
{
 Console.WriteLine("Das Objekt wird zerstört");
}
...
```

*Destruktor der Klasse Fahrzeug*

### Die Methode `Dispose()` aufrufen

Die Aufräumarbeiten werden von der Garbage Collection möglicherweise zeitverzögert ausgeführt, da sie dann durchgeführt werden, wenn das Programm möglichst wenig beschäftigt ist, z. B. wenn es auf Benutzereingaben wartet.

In manchen Fällen kann es erforderlich sein, dass Bereinigungsarbeiten für ein Objekt – z. B. eine geöffnete Datei schließen oder eine Datenbankverbindung beenden – zu einem ganz bestimmten Zeitpunkt erfolgen sollen, den Sie selbst bestimmen möchten.

C# ermöglicht jederzeit den Aufruf einer Methode namens `Dispose()`, mit deren Hilfe Ressourcen ohne Verzögerung automatisiert freigegeben werden können (Sie können natürlich jederzeit eine solche mit einem anderen Namen selbst bereitstellen und aufrufen). Für die Verwendung von `Dispose()` muss die Schnittstelle `IDisposable` in die Klasse implementiert werden. Mit Schnittstellen befasst sich ein späteres Kapitel dieses Buchs.

```
class Classname : System.IDisposable
{
 void Dispose()
 { ... }
}

Classname objectname = new Classname();
... // Code für die Verwendung
 // des Objekts Objectname
objectname.Dispose();
```

### Die `using`-Anweisung einsetzen

Die `using`-Anweisung enthält einen Anweisungsblock, in dem Sie ein Objekt `objectname` verwenden ①, das im Kopf der `using`-Anweisung erzeugt wird. Sobald mit der schließenden geschweiften Klammer ② das Ende der `using`-Anweisung erreicht wird ②, wird automatisch die Methode `Dispose()` ausgeführt und das Objekt somit freigegeben. Die Klasse, die als Typ verwendet wird, muss die Schnittstelle `IDisposable` implementieren.

```
using (Classname objectname = new Classname())
{
 ... // Code für die Verwendung
 // des Objekts objectname ①
} ②
```

## 9.4 Statische Member und statische Klassen

### Was sind statische Member?

Statische Member sind Eigenschaften, Felder oder Methoden, die von allen Instanzen einer Klasse gemeinsam verwendet werden. Sie sind mit der Klasse selbst verknüpft und nicht mit einem speziellen Objekt verbunden. Man nennt sie deshalb auch **Klassenelemente**. Für Member, die Funktionalitäten oder Informationen bereitstellen, die unabhängig von einem speziellen Objekt sind, ersparen Sie sich dadurch die Objekterzeugung, und die Ausführungsgeschwindigkeit des Programms wird erhöht. Dies ist insbesondere bei sehr häufig eingesetzten Methoden beachtenswert.

- ✓ Statische **Felder** und **Eigenschaften** werden für Informationen verwendet, die **Teil der Klasse** und somit für alle Instanzen dieser Klasse gleich sind.
- ✓ Statische **Methoden** sind Klassenmethoden, die mit der Klasse selbst und nicht mit einer Instanz einer Klasse verbunden sind. Sie können solche Methoden verwenden, ohne eine Instanz der entsprechenden Klasse erzeugen zu müssen. Viele Methoden innerhalb der Klassenbibliothek des .NET Frameworks sind Klassenmethoden, z. B. die mathematischen Funktionen der Klasse `System.Math`. Sie können diese Methoden verwenden, ohne eine Instanz der Klasse `Math` bilden zu müssen. Auch die Methode `Main()` ist als statische Klasse deklariert. Beim Programmstart kann die Methode `Main()` ausgeführt werden, ohne dass eine Instanz der umschließenden Klasse (in den Beispielen meist die Klasse `Program`) erzeugt wird.

### Syntax für statische Member

```
[public|private...] static type identifier1; // Feld
[public|private...] static type identifier2; // Eigenschaft
{
 ...
}

[public|private...] static void|type name([parameterlist]) // Methode
{
 ...
}

Classname.membername // Zugriff über den Namen der Klasse
```

- ✓ Die statischen Eigenschaften des Elements werden durch das Schlüsselwort `static` gekennzeichnet.
- ✓ Bei Bedarf kann ein Zugriffsmodifizierer vorangestellt werden.
- ✓ Auf statische Member können Sie direkt über den Klassennamen zugreifen.

Sie können auch über die Objektvariable einer zuvor erstellten Instanz auf einen freigegebenen Member zugreifen. Es wird jedoch empfohlen, statische (`static`) Member immer über den Klassennamen und nicht über Instanzvariablen anzusprechen. Der Zugriff über den Klassennamen verdeutlicht, dass es sich um ein statisches Element handelt, und erspart Ihnen die Objekterzeugung.



**Konstanten** (`const`) sind immer statisch, werden aber nicht explizit mit dem Schlüsselwort `static` deklariert. Es ist im Gegenteil sogar so, dass ein vorangestelltes `static` **verboten** ist.

**Beispiel: AnzahlAutos.sln**

Im folgenden Programm werden drei Instanzen einer Klasse erzeugt. Jedes Mal, wenn eine Instanz erzeugt wurde, wird die in einem statischen Feld gespeicherte aktuelle Anzahl der Instanzen mithilfe einer statischen Eigenschaft auf dem Bildschirm ausgegeben.

```
public class Fahrzeug
{
 ① private static int anzahl = 0;
 private int geschwindigkeit;

 ② public static int Anzahl
 {
 ③ get { return Fahrzeug.anzahl; }
 }

 public int Geschwindigkeit
 {
 get { return this.geschwindigkeit; }
 set
 {
 if (value > 200)
 value = 200;
 geschwindigkeit = value;
 }
 }
 ④ public Fahrzeug(int value)
 {
 Fahrzeug.anzahl++;
 this.Geschwindigkeit = value;
 }

 public Fahrzeug() : this(30) {}
}

class Program
{
 static void Main(string[] args)
 {
 ⑤ Console.WriteLine("Anzahl der Fahrzeuge: " + Fahrzeug.Anzahl);
 Fahrzeug PKW1 = new Fahrzeug(45);
 Console.WriteLine("Anzahl der Fahrzeuge: " + Fahrzeug.Anzahl);
 Fahrzeug PKW2 = new Fahrzeug(38);
 Console.WriteLine("Anzahl der Fahrzeuge: " + Fahrzeug.Anzahl);
 Fahrzeug PKW3 = new Fahrzeug(78);
 Console.WriteLine("Anzahl der Fahrzeuge: " + Fahrzeug.Anzahl);
 Console.WriteLine("Fahrzeug 1: {0} km/h", PKW1.Geschwindigkeit);
 Console.WriteLine("Fahrzeug 2: {0} km/h", PKW2.Geschwindigkeit);
 Console.WriteLine("Fahrzeug 3: {0} km/h", PKW3.Geschwindigkeit);
 }
}
```

- ①/② Die Klasse `Fahrzeug` des vorherigen Beispiels wurde erweitert. Sie erhält zusätzlich das Feld `anzahl` zum Speichern der gebildeten Instanzen und die Eigenschaft `Anzahl`. Die Klasse wird hier als `public` deklariert, um sie überall (z. B. in später folgenden Projekten) nutzen zu können.
- ② Die Eigenschaft `Anzahl` enthält nur einen `get`-Block und darf daher nur gelesen werden. Sie gibt den Wert des Feldes `anzahl` zurück.
- ③ Die Referenzvariable `this` existiert nur für jedes erzeugte Objekt. Da statische Methoden nicht an Objekte gebunden sind, kann in statischen Methoden die Referenzvariable `this` nicht verwendet werden. Zur besseren Übersicht sollten Sie als Referenz den Klassennamen (hier: `Fahrzeuge`) schreiben.
- ④ Der Konstruktor speichert nicht nur den Geschwindigkeitswert, sondern erhöht zudem bei der Erzeugung einer Instanz den Wert des statischen Feldes `anzahl` um 1.
- ⑤ Noch bevor eine Instanz erzeugt wurde, kann an dieser Stelle schon die statische Eigenschaft verwendet werden. Sie zeigt an, dass noch keine Instanz der Klasse `Fahrzeug` existiert.
- ⑥ Nacheinander werden drei Instanzen der Klasse `Fahrzeug` mit unterschiedlichen Geschwindigkeitswerten erzeugt. Jede neue Instanz wird über den Konstruktor gezählt und die Anzahl wird ausgegeben.

```
Anzahl der Fahrzeuge: 0
Anzahl der Fahrzeuge: 1
Anzahl der Fahrzeuge: 2
Anzahl der Fahrzeuge: 3
Fahrzeug 1: 45 km/h
Fahrzeug 2: 38 km/h
Fahrzeug 3: 78 km/h
```

*Die Ausgabe des Programms*

Auch einen statischen Konstruktor können Sie mithilfe des Schlüsselworts `static` deklarieren. Dieser Konstruktor wird vor dem ersten Aufruf einer statischen Methode oder vor dem Erstellen der ersten Instanz ausgeführt. Sie können beispielsweise eine Meldung ausgeben lassen, sobald das erste Objekt von der Klasse erzeugt wird, denn genau dann wird der Konstruktor aufgerufen.

## Statische Klassen

Wenn Sie eine Klasse mit ausschließlich statischen Membern ausstatten, können Sie auch diese Klasse als statisch deklarieren.

Eine statische Klasse hat folgende Eigenschaften:

- ✓ Statische Klassen dürfen lediglich statische Member enthalten. Beachten Sie, dass Konstanten (`const`) ebenfalls statisch sind und somit Bestandteil einer statischen Klasse sein können.
- ✓ Von statischen Klassen dürfen Sie keine Instanzen bilden. Bei der Verwendung von `new` mit einer statischen Klasse erhalten Sie eine Fehlermeldung des Compilers.

## 9.5 Namensräume

Module und Klassen können in sogenannten Namensräumen (engl. Namespaces) hierarchisch organisiert werden.

- ✓ Der Code wird übersichtlicher, weil Namensräume die erstellten Typen nach Themen oder Aufgaben ordnen.
- ✓ Namen müssen innerhalb eines Namensraums eindeutig sein. Der gleiche Bezeichner kann somit in verschiedenen Namensräumen vorkommen.
- ✓ Namensräume können eingebunden werden. Der Zugriff auf die Klassen- oder Modulelemente wird dadurch einfacher und kürzer, weil auf die Qualifizierung der Elemente verzichtet werden kann.

### Einen eigenen Namensraum (Namespace) einrichten

Erzeugen Sie Ihre eigene Namespace-Struktur. Ordnen Sie Ihre Projekte darin beispielsweise nach Themen.

- ✓ Es ist üblich, die Namespace-Struktur anhand der Projektstruktur absteigend zu gliedern, z. B. `Firma.PunktName.UnterPunkt` usw.
- ✓ Innerhalb dieser Hierarchie können Sie weitere Namespaces einrichten, mit denen Sie die Klassen beispielsweise thematisch ordnen können.

## Syntax zur Erstellung eines Namensraums

- ✓ Die Deklaration eines Namensraumes beginnt mit dem Schlüsselwort `namespace`, gefolgt vom Bezeichner des Namensraums. Die Inhalte des Namensraums bilden einen Block, der in geschweifte Klammern `{ }`  eingeschlossen wird.

```
namespace NameOfNamespace
{
 class Classname // z. B. Definition
 {
 ...
 }
 ...
}
```

// Ende des Namensraums

- ✓ Innerhalb von Namensräumen können Sie Klassen, Schnittstellen, Delegaten, Enumerationen, Strukturen und weitere Namensräume deklarieren. Eigenschaften, Variablen, Methoden und Ereignisse können dagegen nicht direkt auf der Namensraumebene, sondern nur innerhalb von Klassen oder Strukturen deklariert werden.

Jede in Visual C# erstellte ausführbare Datei erhält automatisch einen Namensraum, der den gleichen Bezeichner hat wie das Projekt, zu der die ausführbare Datei gehört. Der Bezeichner lässt sich unter den Projekteigenschaften einsehen und verändern, z. B. wenn Sie die Namensräume anders ordnen wollen. Der dort angegebene Namensraum wird dem Namen, den Sie über das Schlüsselwort `namespace` angeben, vorangestellt. Entfernen Sie ggf. den in den Projekteigenschaften angegebenen Namen.

Wenn Sie in einem Projekt Unterordner erstellen, wird bei den darin erstellten Dateien der Name des Unterordners an den Namespace-Namen angefügt.

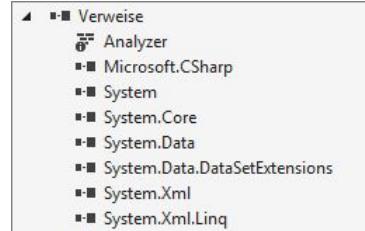
## Auf Namensräume zugreifen

Damit auf Klassen eines Namensraums zugegriffen werden kann, muss ein Verweis auf den Namensraum bestehen. Standardmäßig besitzen neue Projekte bereits mehrere Verweise auf Namensräume.

### Bestehende Verweise auf Namensräume anzeigen

Im Projektmappen-Explorer werden Ihnen die bestehenden Verweise auf Namensräume eingeblendet.

Auf welche Namensräume Verweise bestehen, hängt vom Projekttyp ab. Die Abbildung zeigt die standardmäßig enthaltenen Verweise einer Windows-Anwendung.



Verweise auf Namensräume  
(Windows-Anwendung)

### Verweise auf Namensräume hinzufügen

- ▶ Rufen Sie den Menüpunkt *PROJEKT - Verweis hinzufügen...* auf.
- oder Öffnen Sie im Projektmappen-Explorer das Kontextmenü des Eintrags *Verweise* und wählen Sie den Befehl *Verweis hinzufügen....*.
- ▶ Wechseln Sie im geöffneten Dialogfenster in das Register *Durchsuchen* und suchen Sie nach der gewünschten Datei, die die gewünschten Klassen in dem entsprechenden Namensraum beinhaltet.



Die ausführbaren Dateien eines anderen Projekts (z. B. *AnzahlAutos*) finden Sie im Projektordner des Projekts im Unterordner *bin\Release* bzw. *bin\Debug*. Welcher dieser beiden Ordner die aktuelle Version der ausführbaren Datei enthält, hängt davon ab, mit welcher Einstellung im Listenfeld *Projektmappenkonfigurationen* das Projekt zuletzt kompliiert wurde. Standardmäßig ist bei neuen Projekten die Einstellung *Debug* gewählt und die ausführbare Datei wird im Ordner *Debug* erstellt. Bei der Einstellung *Release* wird sie im Ordner *Release* erstellt. Die Kompilierung erfolgt immer, wenn Sie das Programm in Visual Studio ausführen oder einen der Menüpunkte aus dem Menü *Erstellen* verwenden und keine aktuelle Version vorliegt.

Wenn Sie ein Projekt, wie beispielsweise *AnzahlAutos*, geöffnet und fertig getestet haben, sollten Sie mit der Einstellung *Release* die ausführbare Datei *AnzahlAutos.exe* erzeugen. Diese wird im Unterordner *Release* abgelegt und sollte auch für den Import des Namensraums verwendet werden. Bei der Einstellung *Release* wird das Projekt ohne Debug-Informationen und mit eingeschalteter Optimierung für den Programmcode kompiliert. Es entsteht die sogenannte „Freigabeversion“ Ihres Programms.



### Verweis entfernen

Im Projektmappen-Explorer können Sie über das Kontextmenü des Verweises den Verweis mit *Entfernen* löschen.

### Syntax für die Verwendung von Klassen eines Namensraums

- ✓ Schreiben Sie entsprechend der Namensraumhierarchie, abgetrennt durch Punkte, die Namen der Namensräume.
- ✓ Fügen Sie anschließend, ebenfalls durch einen Punkt getrennt, den Namen der Klasse an.

`NameOfNamespace [ .NameOfSubNamespace ... ] .Classname`

### Beispiel

Da auf den Namensraum `System` standardmäßig ein Verweis besteht, können Sie beispielsweise die Klassen `Console` und `Math` sofort verwenden.

```
System.Console.WriteLine("Test");
int number = System.Math.Max(3, 5);
```

### Namensräume einbinden

Mithilfe des Schlüsselworts `using` am Anfang einer Programmcode datei können Sie einen Namensraum importieren. Auf diese Weise vereinfachen Sie den Zugriff auf die enthaltenen Elemente, beispielsweise auf enthaltene Klassen.

```
using System;
...
int number = Math.Max(3, 5);
// statt System.Math.Max(3, 5);
```

### Standardmäßig eingebundene Namensräume entfernen

Wenn Sie in Visual Studio ein neues Projekt erstellen, werden standardmäßig verschiedene Namespaces eingebunden, im Falle einer Konsolenanwendung z. B.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Es wird dabei davon ausgegangen, dass Sie diese Namespaces häufig benötigen. Allerdings können Sie diese Einbindungen entfernen, wenn Sie sie nicht benötigen, um überflüssige Anweisungen zu vermeiden. In den mitgelieferten Beispielen wurde dies größtenteils getan.



### Beispiel: *Namensraeume.sln* und *AnzahlAutos.sln*

In einem neuen Projekt *Namensraeume.sln* soll die Klasse Fahrzeug des Projekts *AnzahlAutos.sln* verwendet werden. Erzeugen Sie dazu im Projekt *Namensraeume* einen Verweis auf den Namensraum des Projekts *AnzahlAutos*, indem Sie beispielsweise über den Menüpunkt *PROJEKT - Verweis hinzufügen...* einen Verweis auf die Datei *AnzahlAutos.exe* erzeugen.

```
using System;
...
① using AnzahlAutos;
...
static void Main(string[] args)
{
 Fahrzeug LKW = new Fahrzeug(80); // statt AnzahlAutos.Fahrzeug...
 Fahrzeug PKW = new Fahrzeug(130);
 Console.WriteLine("Anzahl der Fahrzeuge: " + Fahrzeug.Anzahl);
 Console.WriteLine("Fahrzeug 1 {0} km/h", LKW.Geschwindigkeit);
 Console.WriteLine("Fahrzeug 2 {0} km/h", PKW.Geschwindigkeit);
}
...
...
```

- ① Der Namensraum `AnzahlAutos` wird importiert.
- ② Durch das Importieren können Sie auf die Klasse `Fahrzeug` ohne die Angabe des Namensraums zugreifen.
- ③ Auch die freigegebenen Member, z. B. `Anzahl`, können Sie ohne Angabe des Namensraums nutzen.

### Aliasnamen verwenden

Um ein Element anzusprechen, dessen Name in **mehreren** eingebundenen Namensräumen existiert, müssen Sie die vollständige Qualifizierung (`NameOfNameSpace.Classname`) angeben, auch wenn Sie die entsprechenden Namensräume importiert haben.

```
using Aliasname1 = System.Windows.Forms.Button;
using Aliasname2 = MyFirstProjekt.Form1.Button;
...
Aliasname1 btn1 = new Aliasname1();
Aliasname2 btn2 = new Aliasname2();
...
```

In solchen Fällen ist es aber möglich, Aliasnamen für die meist sehr langen Qualifizierer einzurichten.

### Namensräume mit gleichlautenden Bezeichnern

Falls Sie Klassen mit gleichlautenden Bezeichnern in einem anderen Namensraum ansprechen möchten, müssen Sie den vollständigen qualifizierten Namen einschließlich des Namensraums angeben.

In dem nebenstehenden Beispiel führt aber auch die Angabe des Namensraums zu einem Fehler:

```
System.Console.Write("Hallo!");
```

In der hier erstellten Klasse `System` ① existiert kein Bezeichner `Console`.

```
① class System
{
 static void Main(string[] args)
 {
 ② global::System.Console.Write("Hallo!");
 }
}
```

In diesem Fall können Sie mit angeben, dass der Compiler im globalen Namensraum des .NET Frameworks nach dem gewünschten Bezeichner suchen soll ②.

- ✓ Geben Sie das Schlüsselwort `global` ein.
- ✓ Anschließend folgt, abgetrennt mit zwei Doppelpunkten `::`, der qualifizierte Name des Bezeichners.

## 9.6 Partielle Klassen erstellen

### Programmcode einer Klasse auf mehrere Dateien aufteilen

Seit dem .NET Framework 2.0 haben Sie die Möglichkeit, die Beschreibung einer Klasse auf mehrere Dateien aufzuteilen. So können Sie beispielsweise in einer Datei alle grundlegenden Beschreibungen wie Felder, Eigenschaften und Konstruktoren speichern und in einer zweiten Datei die Funktionalität, also alle weiteren Methoden, deklarieren. Üblicherweise wird diese Möglichkeit von Entwicklerteams genutzt, in denen mehrere Programmierer gemeinsam an einer Klasse arbeiten, bzw. von Tools, die mehrere Dateien (z. B. eine für die Benutzeroberfläche und eine für die Funktionalität) generieren. Aber auch bei grafischen Oberflächen findet man oft eine solche Aufteilung auf mehrere zusammengehörende Dateien.

#### Syntax für partielle Klassen

- ✓ Ergänzen Sie in der Deklaration für die Klasse in allen Dateien das Schlüsselwort `partial` vor dem Schlüsselwort `class`.
- ✓ Deklarieren Sie die Klasse ansonsten wie gewohnt, sofern erforderlich, mit den gewünschten Modifizierern.
- ✓ Das Schlüsselwort `partial` muss **direkt vor** dem Schlüsselwort `class` notiert werden.
- ✓ Alle Klassen-Deklarationen müssen die gleichen Zugriffsberechtigungen besitzen.

```
[modifier] partial class Classname
{
 ...
}
```

*Klassendeklaration in den einzelnen Dateien*

In Windows Forms wird die Beschreibung eines Formulars beispielsweise automatisch auf zwei Dateien aufgeteilt. Eine Datei enthält den vom Designer erstellten Programmcode zur Gestaltung des Formulars. In der anderen Datei programmieren Sie beispielsweise Ihre Ereignismethoden. Dennoch werden beide Bestandteile als eine Klasse (mit zwei partiellen Klassendeklarationen) deklariert.

## 9.7 Partielle Methoden

Neben partiellen Klassen, die mit dem .NET Framework 2.0 eingeführt wurden, existieren seit dem .NET Framework 3.5 auch partielle Methoden. Das Einsatzgebiet partieller Methoden ist durch ihre Funktionsweise begründet. Codegeneratoren oder Programmierer können partielle Methoden deklarieren, die sie **irgendwann** implementieren. Die Methoden werden vollständig vom Compiler entfernt (Deklaration und alle Aufrufe), wenn die Implementierung fehlt. Sie erhalten bei fehlender Implementierung demnach immer noch eine ausführbare Anwendung.

#### Syntax partieller Methoden

- ✓ Eine partielle Methode wird in einer partiellen Klasse wie bei einem Interface deklariert und in einer weiteren partiellen Klasse implementiert.
- ✓ Der Rückgabewert einer partiellen Methode ist immer `void`.
- ✓ Partielle Methoden sind implizit `private`. Die Angabe eines Modifizierers wie `public` ist verboten. Einzig die Angabe von `static` ist erlaubt, da partielle Methoden auch statisch sein dürfen.
- ✓ Als Parameter sind keine `out`-Parameter erlaubt.
- ✓ Wenn Sie eine partielle Methode deklarieren, aber nicht implementieren, werden die Deklaration und alle Verwendungen entfernt.

### Beispiel: TaschenRechner.sln

Eine Anwendung soll als Taschenrechner dienen. Einige Methoden wurden aber noch nicht vollständig implementiert. Dennoch sollten über die Methodendeklarationen schon einmal die möglichen Operationen festgelegt werden, die der Rechner bereitstellen soll.

Weiterhin zeigt das Beispiel eine Möglichkeit, wie mithilfe von Referenzparametern (wenn auch ohne die Berücksichtigung aller Sonderfälle) über partielle Methoden dennoch Rückgabewerte verarbeitet werden können.

```
using System;
namespace TaschenRechner
{
 class Program
 {
 static void Main(string[] args)
 {
 Rechner rech = new Rechner();
 Console.WriteLine(rech.Add(10, 11));
 }
 }

 ① public partial class Rechner
 {
 ② public int Add(int zahl1, int zahl2)
 {
 int summe = 0;
 InternalAdd(zahl1, zahl2, ref summe);
 return summe;
 }

 ③ partial void InternalAdd(int zahl1, int zahl2, ref int summe);
 }

 ④ public partial class Rechner
 {
 ⑤ partial void InternalAdd(int zahl1, int zahl2, ref int summe)
 {
 summe = zahl1 + zahl2;
 }
 }
}
```

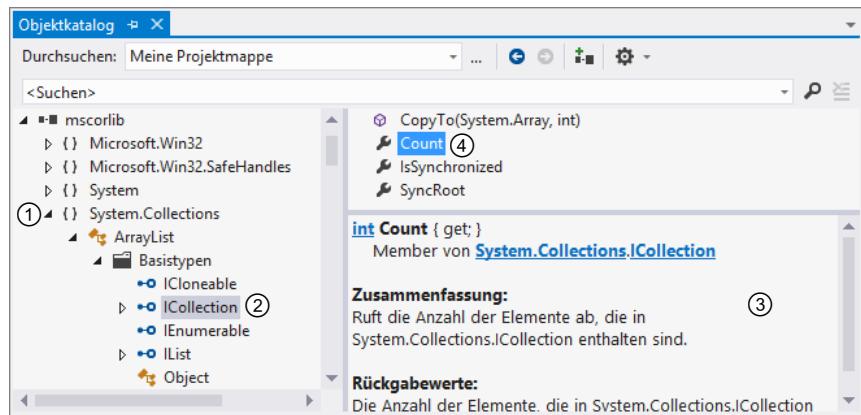
- ① Die partielle Klasse Rechner wird deklariert.
- ② Da partielle Methoden private sind, muss eine Rahmenmethode zum Aufruf bereitgestellt werden.
- ③ Hier erfolgt die Deklaration der partiellen Methode InternalAdd(). Diese ist implizit private und kann nur innerhalb der Klassendeklaration aufgerufen werden.
- ④ Ein weiterer Teil der Klasse Rechner wird deklariert. In der Praxis befinden sich die verschiedenen Teile partieller Klassen in unterschiedlichen Dateien. Dies wurde zur Vereinfachung für das Beispiel nicht berücksichtigt.
- ⑤ Hier erfolgt nun die Implementierung der Methode InternalAdd(). Wird diese Implementierung auskommentiert, erfolgt in der Methode Add() kein Aufruf von InternalAdd() und der Wert 0, der der Variablen summe bei der Deklaration zugewiesen wurde, wird ausgegeben.

## 9.8 Informationen zu Klassen erhalten

### Informationen im Objektkatalog erhalten

- Rufen Sie den Menüpunkt **Ansicht - Objektkatalog** auf, um den Objektkatalog einzublenden.  
Alternative: **Strg W** und dann **J**

Der Objektkatalog ermöglicht das Anzeigen von Namensräumen (Namespaces) ① und den darin enthaltenen Typen und ihren Elementen ②.



Im Bereich ③ erhalten Sie Informationen zu dem zuletzt ausgewählten Element, hier zu der im Listenfeld markierten Methode ④.

Sie können auf diese Weise die Klassenbibliothek des .NET Frameworks (z. B. *mscorlib*) betrachten oder durchsuchen. Die Klassenstruktur eigener (z. B. *Namensräume*) oder fremder .NET-Projekte (z. B. *AnzahlAutos*) lässt sich ebenfalls darstellen.

- Mit dem Symbol fügen Sie dem geöffneten Projekt den markierten Namensraum als Verweis hinzu.
- Über das Symbol wählen Sie, welche Informationen im Objektbrowser angezeigt werden sollen.



### Übersicht wichtiger Symbole

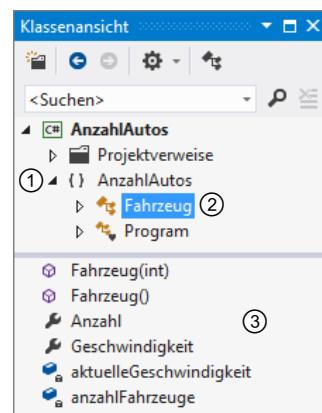
| Symbol | Beschreibung       | Symbol | Beschreibung  | Symbol | Beschreibung |
|--------|--------------------|--------|---------------|--------|--------------|
|        | Namespace          |        | Eigenschaft   |        | Operator     |
|        | Klasse             |        | Schnittstelle |        | Enumeration  |
|        | Methode            |        | Ereignis      |        | Struktur     |
|        | Variable oder Feld |        | Konstante     |        | Delegat      |
|        | Exception          |        |               |        |              |

### Klassenstruktur des geöffneten Projekts anzeigen

Mithilfe der Klassenansicht erhalten Sie einen Überblick über die Klassen des geöffneten Projekts.

- Rufen Sie den Menüpunkt **Ansicht - Klassenansicht** auf, um die Klassenansicht zu öffnen.  
Alternative: **Strg W** und dann **C**

Die Ansicht zeigt innerhalb des Projekts den Namespace ①, die enthaltenen Klassen ② und die darin integrierten Member ③.



## 9.9 Übungen

### Übung 1: Klassen und Klasseninstanzen

**Übungsdatei:** --

**Ergebnisdatei:** SomeCars.sln

1. Erstellen Sie eine Konsolenanwendung.
2. Deklarieren Sie eine Klasse `Auto` und in dieser Klasse je ein Feld für das Baujahr und die Farbe sowie zwei Eigenschaften für den Schreib-/Lesezugriff auf die Felder. Deklarieren Sie außerdem einen Konstruktor, mit dem Sie die beiden Felder initialisieren können.
3. Erzeugen Sie in der Hauptmethode der Anwendung zwei Instanzen der Klasse `Auto` und zeigen Sie die Werte der Felder nach der Initialisierung mithilfe der Eigenschaften auf dem Bildschirm an.
4. Ändern Sie dann die Farbwerte der beiden Objekte. Geben Sie zur Kontrolle der Änderung die Werte der Felder noch einmal aus.

### Übung 2: Kapselung – Zugriff auf Felder über Eigenschaften

**Übungsdatei:** --

**Ergebnisdatei:** Kontofuehrung.sln

1. Erstellen Sie eine Konsolenanwendung, um ein Programm für eine einfache Kontoführung zu schreiben.
2. Entwickeln Sie eine in einer separaten Datei gespeicherte Klasse `Konto`. Als einziges Feld soll diese Klasse eine als `private` deklarierte Variable für den Kontostand besitzen.
3. Der Zugriff auf den Kontostand erfolgt ausschließlich über eine als `public` deklarierte Eigenschaft für den Schreib-/Lesezugriff.
4. Implementieren Sie einen Konstruktor, über den Sie den Kontostand mit einem bestimmten Wert (hier 5000) initialisieren können.
5. Die Benutzerschnittstelle sollte so aufgebaut sein, dass sowohl zu Beginn des Programms als auch nach jeder Ein- bzw. Auszahlung der aktuelle Kontostand angezeigt wird. Mit einer `switch`-Anweisung ermöglichen Sie dem Benutzer, Ein- bzw. Auszahlungen vorzunehmen sowie das Programm zu beenden.

```
Kontostand: 5000
Einzahlen(1), Auszahlen(2), Beenden(0):
1
Betrag: 900
Kontostand: 5900
Einzahlen(1), Auszahlen(2), Beenden(0):
2
Betrag: 750
Kontostand: 5150
Einzahlen(1), Auszahlen(2), Beenden(0):
```

*Die Ausgabe des Programms*

### Übung 3: Statische Member

**Übungsdatei:** --

**Ergebnisdatei:** Steuern.sln

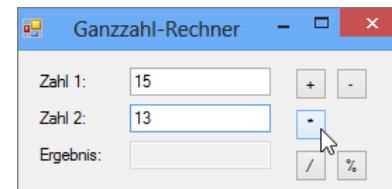
1. Erstellen Sie eine neue Konsolenanwendung.
2. Schreiben Sie in der Datei `Program.cs` eine Klasse `Steuern`, die ein statisches Feld `Mehrwertsteuer` vom Typ `double` und eine statische Eigenschaft `MWST` für den Schreib-/Lesezugriff auf diese Variable enthält.
3. Implementieren Sie zusätzlich in der Klasse eine öffentliche Methode `InclMWST`, die zu einem übergebenen Betrag vom Typ `double` die Mehrwertsteuer hinzurechnet und das Gesamtergebnis zurückgibt.
4. Legen Sie in der Klasse `Program` mit der Methode `Main()` die Mehrwertsteuer auf 19 % (=0,19) fest. Speichern Sie in einer Variablen einen beliebigen Betrag (`netto`) und berechnen Sie anschließend den Gesamtbetrag inklusive Mehrwertsteuer (`brutto`). Geben Sie den Mehrwertsteuersatz sowie den Netto- und den Bruttobetrag aus.

## Übung 4: Ganzzahl-Taschenrechner objektorientiert

**Übungsdatei:** Kap09\IntegerCalc\IntegerCalcOOP.sln

**Ergebnisdatei:** IntegerCalcProp.sln

1. Verändern Sie den Taschenrechner aus der Übung des Kapitels 9. Kapseln Sie die beiden Felder für die Speicherung der eingegebenen Zahlen und implementieren Sie zwei Schreib-/ Leseeigenschaften, um auf die beiden Felder zuzugreifen.



# 10 Vererbung

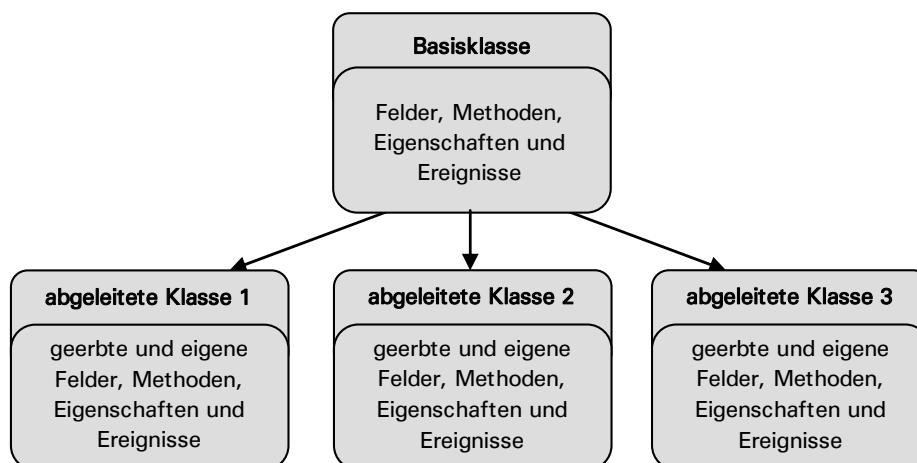
## 10.1 Grundlagen zur Vererbung

### Was ist Vererbung?

Die Vererbung ist ein wichtiges Merkmal objektorientierter Programmiersprachen.

- ✓ Über Vererbung erzeugen Sie eine neue Klasse auf der Basis einer bereits vorhandenen Klasse, der sogenannten Basisklasse. Die neue Klasse wird somit von der Basisklasse abgeleitet. Die Basisklasse wird auch als Superklasse bezeichnet.
- ✓ Die neue **abgeleitete Klasse** (oft als Subklasse bezeichnet) übernimmt dabei automatisch die nicht privaten Felder, Eigenschaften, Methoden und Ereignisse der **übergeordneten Basisklasse**.
- ✓ Sie können zu den geerbten Feldern, Eigenschaften, Methoden und Ereignissen weitere hinzufügen oder geerbte Methoden ändern. Die abgeleitete Klasse ist dann gegenüber der Basisklasse erweitert.
- ✓ Eine Klasse kann in C# nur von **einer Klasse** abgeleitet werden (**Einfachvererbung**).

Die Gemeinsamkeiten mehrerer potenzieller Klassen werden in einer Basisklasse zusammengefasst (Generalisierung). Die Nachfahren erhalten dann nur noch Methoden, Felder, Eigenschaften und Ereignisse, in denen sie sich unterscheiden.



Beachten Sie, dass das Diagramm die Vererbung anzeigen soll und kein Klassendiagramm darstellt. Dort wird eine andere Symbolik verwendet (insbesondere in Hinsicht auf die Pfeile) und die Sichtweise ist anders.

## 10.2 Klassen ableiten und erweitern

### Syntax für das Ableiten und Erweitern von Klassen

- ✓ Beginnen Sie die Deklaration der abgeleiteten Klasse mit dem Schlüsselwort `class` und dem Klassennamen.
- ✓ Fügen Sie – abgetrennt mit einem Doppelpunkt `:` – den Namen der Basisklasse an ①.

```
[modifier] class Classname : Baseclassname
{
 ...
}
```

- ✓ Die neue Klasse erbt alle nicht privaten Felder, Eigenschaften, Methoden und Ereignisse der Basisklasse (`public`, `protected`, `internal`, `protected internal`), solange sich Basisklasse und abgeleitete Klasse bei einschränkenden Modifizierern im gleichen Gültigkeitsbereich befinden.
- ✓ Sie können die Basisklasse jetzt erweitern, indem Sie innerhalb der Klassenbeschreibung weitere Felder, Eigenschaften, Methoden und Ereignisse hinzufügen, die für die **abgeleitete** Klasse spezifisch sind.

## Klassen versiegeln

- ✓ Wenn Sie verhindern wollen, dass von einer Klasse weitere Klassen abgeleitet werden können, fügen Sie bei der Klassendeklaration das Schlüsselwort `sealed` hinzu. Die Klasse kann dann nicht als Basisklasse verwendet werden.
- ✓ Auf private Member der Basisklasse kann in abgeleiteten Klassen nicht zugegriffen werden. Wenn Sie auf diese Member zugreifen wollen, müssen Sie sie in der Basisklasse mindestens als `protected` deklarieren.

```
sealed class Classname
{
 ...
}
```

## Beispiel: *Zahlen.sln*

Im folgenden Programm werden die Basisklasse `Zahl1` und die davon abgeleitete Klasse `Zahl2` deklariert. `Zahl2` erbt nicht nur das Feld und die Methode der übergeordneten Klasse, sondern wird zusätzlich um die Methode `Eingabe()` erweitert.

- ① Die Basisklasse `Zahl1` wird deklariert. Sie besitzt das Feld `wert` und die Methode `Ausgabe()`. Das Feld `wert` wird als `protected` deklariert, damit in abgeleiteten Klassen darauf zugegriffen werden kann. Ein Zugriff auf das Feld aus der Klasse `Program` heraus ist jedoch nicht möglich, da diese Klasse nicht von der Klasse `Zahl` abgeleitet wird.
- ② Die Klasse `Zahl2` wird von der Klasse `Zahl1` abgeleitet. Die Klasse wird erweitert, indem sie zusätzlich die Methode `Eingabe()` erhält. Im Gegensatz zur Basisklasse kann der Wert des Feldes `wert` nicht nur ausgegeben, sondern auch verändert werden.
- ③ In der Methode `Main()` des Programms wird mit `new` eine Instanz der abgeleiteten Klasse `Zahl2` erzeugt und der Objektvariablen (`eineZahl`) vom Typ dieser Klasse zugewiesen.
- ④ Die Methode `Eingabe()` der abgeleiteten Klasse nimmt einen Wert entgegen und speichert ihn in dem Feld `wert`.
- ⑤ Die von der Basisklasse geerbte Methode `Ausgabe()` gibt den Wert des Feldes `wert` auf dem Bildschirm aus.

```
① class Zahl1
{
 protected int wert;
 public void Ausgabe()
 {
 Console.WriteLine(this.wert);
 }
}
② class Zahl2 : Zahl1
{
 public void Eingabe(int i)
 {
 this.wert = i;
 }
}
class Program
{
 ③ static void Main(string[] args)
 {
 Zahl2 eineZahl = new Zahl2();
 ④ eineZahl.Eingabe(10);
 eineZahl.Ausgabe();
 }
}
```

## Konstruktoren aufrufen

Bei der Vererbung werden keine Konstruktoren vererbt. Stattdessen hat jede Klasse ihren eigenen Konstruktor.

- ✓ Für die abgeleitete Klasse steht zunächst nur der parameterlose Standardkonstruktor zur Verfügung.
- ✓ Besitzt die **Basisklasse** einen **parameterlosen Konstruktor**, wird dieser automatisch vom Konstruktor der abgeleiteten Klasse aufgerufen. Sie können dies überprüfen, indem Sie im vorherigen Beispiel *Zahlen.sln* einen parameterlosen Konstruktor ① einfügen, der lediglich eine Ausgabe erzeugt.
- ✓ Besitzt die **Basisklasse** einen **Konstruktor mit Parametern** ② und keinen parameterlosen Konstruktor, existiert für diese Klasse der Standardkonstruktor nicht mehr. In diesem Fall müssen Sie in der abgeleiteten Klasse einen Konstruktor schreiben ③, der den speziellen Konstruktor der Basisklasse mit geeigneten Argumenten aufruft ④.
- ✓ Schreiben Sie dazu direkt hinter die Deklaration des Konstruktors einen Doppelpunkt [:]. Anschließend folgt das Schlüsselwort base mit einem Klammernpaar, das ggf. die erforderlichen Argumente enthält.

```
class Zahl1
{
 protected int wert;
 public Zahl1()
 {
 Console.WriteLine(...);
 }
 public void Ausgabe()
 {...}
```

Parameterloser Konstruktor mit Ausgabeanweisung

```
class Zahl1
{
 ...
② public Zahl1(int wert)
{
 ...
}
}

class Zahl2 : Zahl1
{
 public Zahl2()
③ : base(5)
④ {
 Console.WriteLine(...);
 }
 ...
```

Aufruf des Konstruktors der Basisklasse (*Zahlen2.sln*)

Mit dem Schlüsselwort base stellen Sie eine Referenz auf die **Basisklasse** her. Auf diese Weise können Sie mit base auch auf die Member einer Klasse zugreifen oder den Konstruktor der Basisklasse aufrufen.

## 10.3 Vererbungsketten

### Eine Klassenhierarchie bilden

Von einer Klasse können Klassen abgeleitet werden, von denen wiederum Klassen abgeleitet werden. So entstehen Vererbungsketten, die eine **Klassenhierarchie** bilden.

### Die Basisklasse `Object`

Jede Klasse, die Sie verwenden oder neu erzeugen, ist direkt oder indirekt, d. h. über weitere Vorfahren, von der Klasse `System.Object` abgeleitet. Dies müssen Sie jedoch bei der Klassendefinition nicht angeben.

```
class Zahl1 [: Object]
```

Die in der Klasse `Object` definierten Methoden sind aufgrund der Vererbung in allen Klassen verfügbar. So erben Klassen beispielsweise

- ✓ die Methode `GetType()`, die den genauen Typ eines Objekts zurückgibt

*und*

- ✓ die Methode `ToString()`, die eine String-Repräsentation des Objekts liefert.

```
Object
+ ~Object()
+ Equals(object, object)
+ Equals(object)
+ GetHashCode()
+ GetType()
+ MemberwiseClone()
+ Object()
+ ReferenceEquals(object, object)
+ ToString()
```

```
public class Object
 Member von System
```

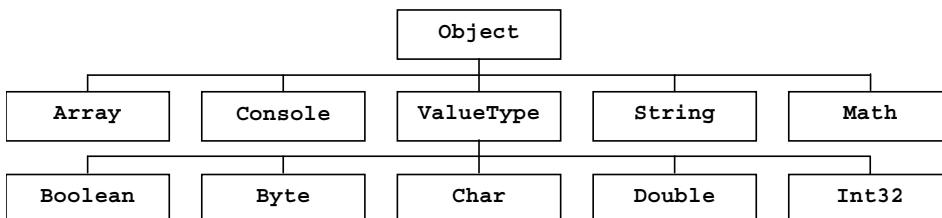
#### Zusammenfassung:

Unterstützt sämtliche Klassen in der Hierarchie von .NET Framework-Klassen und stellt abgeleiteten Klassen Low-Level-Dienste zur Verfügung. Dies ist die allen Klassen von .NET Framework übergeordnete Basisklasse und stellt den Stamm der Typ hierarchie dar.

*Die Klasse Object im Objektkatalog*

### Klassenhierarchie im .NET Framework

Im .NET Framework sind alle Klassen thematisch in verschiedenen Namensräumen geordnet. Nachfolgend wird ein kleiner Teil der Klassenhierarchie grafisch dargestellt und beschrieben. Einige dieser Klassen werden Sie in den folgenden Kapiteln kennenlernen.



| Klasse                        | Beschreibung                                                                                                                                                                                                                                                                            |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>System.Object</code>    | Die Klasse <code>Object</code> beinhaltet allgemeine Methoden, die alle Objekte gemeinsam haben.                                                                                                                                                                                        |
| <code>System.Array</code>     | Die Klasse <code>Array</code> stellt Methoden zum Erstellen, Bearbeiten, Durchsuchen und Sortieren von Arrays bereit. Sie ist die Basisklasse für alle Arrays in den .NET-Programmiersprachen.                                                                                          |
| <code>System.Console</code>   | Die Klasse <code>Console</code> stellt grundlegende Funktionen für Anwendungen bereit, die Zeichen von der Konsole lesen und auf die Konsole schreiben. Sie kann nicht weitervererbt werden.                                                                                            |
| <code>System.ValueType</code> | Die Klasse <code>ValueType</code> ist die Basisklasse für alle Werttypen.                                                                                                                                                                                                               |
| <code>System.String</code>    | Die Klasse <code>String</code> enthält Methoden zur Bearbeitung von Zeichenfolgen. Ein Objekt der Klasse ist eine unveränderliche Folge von Zeichen. Die Methoden der Klasse geben nach der Bearbeitung immer die entsprechend veränderte Kopie der ursprünglichen Zeichenkette zurück. |
| <code>System.Math</code>      | Die Klasse <code>Math</code> stellt Konstanten und Methoden für viele mathematische Funktionen bereit.                                                                                                                                                                                  |

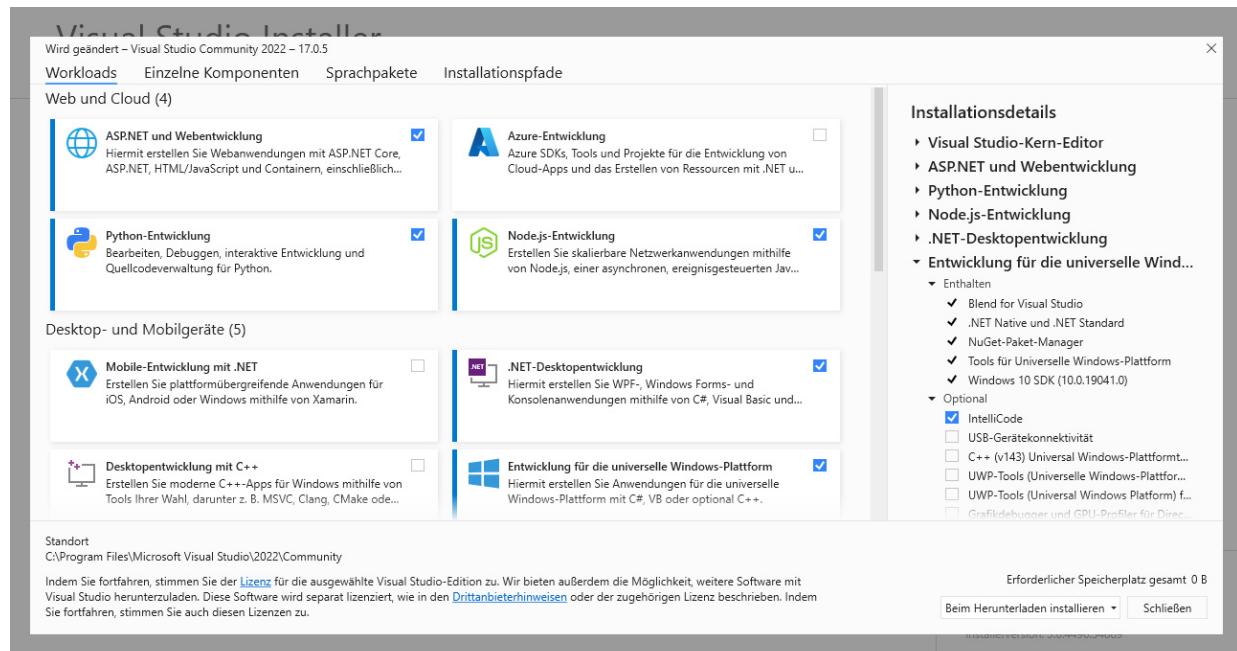
Von der Klasse `ValueType` sind u. a. folgende Klassen abgeleitet:

| Klasse                      | Beschreibung                                                                                                      |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>System.Boolean</code> | Diese Klasse stellt einen Werttyp für boolesche Werte zur Verfügung.                                              |
| <code>System.Byte</code>    | Diese Klasse stellt einen Werttyp für 8-Bit-Ganzzahlen ohne Vorzeichen zur Verfügung.                             |
| <code>System.Char</code>    | Die Klasse <code>Char</code> stellt einen Werttyp für Unicode-Zeichen zur Verfügung.                              |
| <code>System.Double</code>  | Die Klasse <code>Double</code> stellt einen Werttyp für Gleitkommazahlen mit doppelter Genauigkeit zur Verfügung. |
| <code>System.Int32</code>   | Diese Klasse stellt einen Werttyp für 32-Bit-Ganzzahlen mit Vorzeichen zur Verfügung.                             |

## 10.4 Mit Klassendiagrammen arbeiten

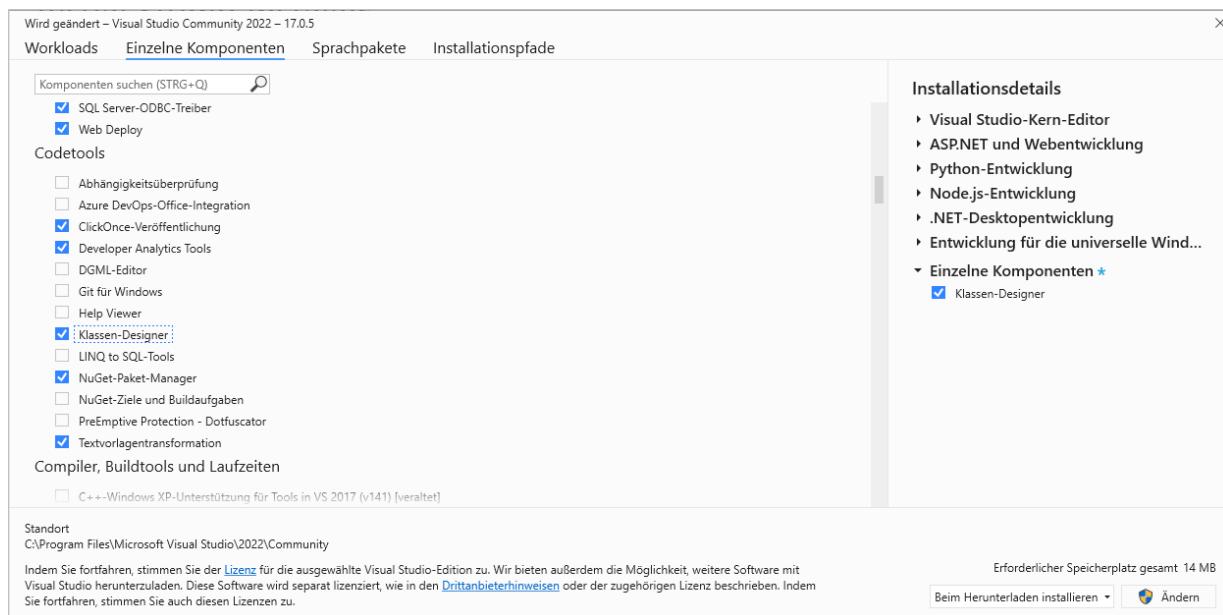
### Klassenhierarchien mit dem Klassen-Designer darstellen

In Visual Studio haben Sie die Möglichkeit, bei einer .NET-Framework-Applikation – derzeit aber (noch) **nicht** bei einer Core-Applikation – die Klassenhierarchie Ihrer Projekte grafisch darzustellen. Allerdings wird die dazu notwendige Komponente **Klassen-Designer** nicht immer in Visual Studio automatisch mitinstalliert. Sofern diese Komponente nicht vorhanden ist, können Sie wie folgt vorgehen, um sie nachträglich zu installieren:



### Der Visual Studio-Installer

- ▶ Öffnen Sie den **Visual Studio-Installer** über die Auswahl von *Extras - Get Tools and Features (Tools und Features abrufen)* über die Menüleiste in Visual Studio. Mit dem Visual Studio-Installer können Sie Visual Studio um zahlreiche Tools und Features erweitern.
- ▶ Wählen Sie die Registerkarte *Einzelne Komponenten* aus und scrollen Sie nach unten zur Kategorie *Codetools*.
- ▶ Wählen Sie den Klassen-Designer aus, und klicken Sie anschließend auf *Ändern*. Damit wird der Klassen-Designer installiert. Dabei wird Visual Studio geschlossen und neu gestartet. Deshalb sollten Sie alle Projekte vorher speichern.



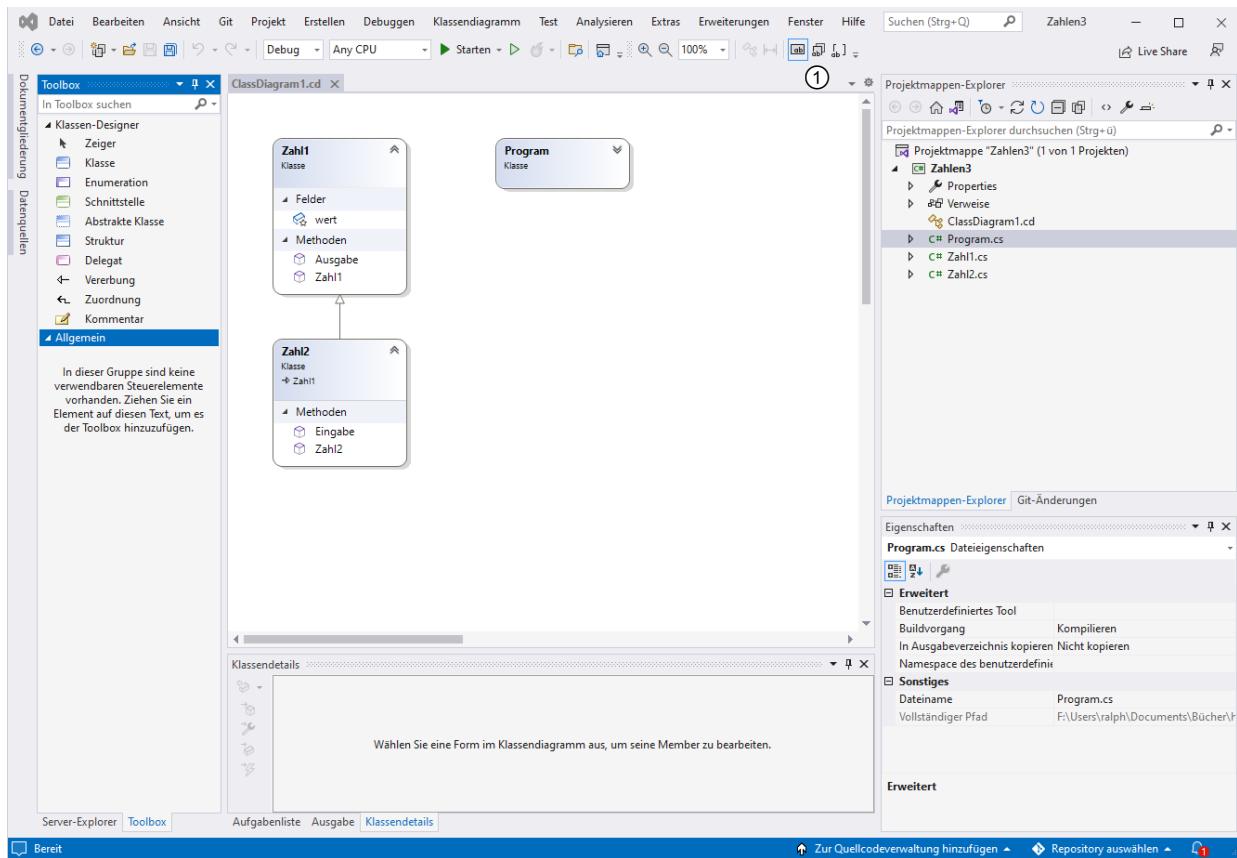
### *Der Klassen-Designer wird über den Visual Studio-Installer verfügbar gemacht*

Ist der Klassen-Designer vorhanden und Sie arbeiten mit einem .NET-Framework-Projekt, können Sie wie folgt vorgehen:

- Öffnen Sie das gewünschte Projekt.
- Markieren Sie im Projektmappen-Explorer die C#-Dateien (\*.cs), deren Klassen Sie im Diagramm darstellen möchten.
- oder    Markieren Sie im Projektmappen-Explorer den Projektnamen, um alle Klassen des Projekts im Diagramm darzustellen.
- Wählen Sie den Befehl *Klassendiagramm anzeigen* im Kontextmenü der gewünschten .cs-Datei bzw. im Kontextmenü des Projekts im Projektmappen-Explorer.

Das Klassendiagramm wird erstellt und angezeigt. Außerdem wird dem Projekt eine Diagrammdatei mit der Erweiterung .cd hinzugefügt.

Beachten Sie, dass die Klassendiagramme in dem Klassen-Designer **nicht** dem UML-Standard entsprechen – auch wenn sie sehr ähnlich sind. Der entscheidende Grund ist, dass UML-Klassendiagramme den allgemeinen Definitionen der objektorientierten Programmierung entsprechen, von denen Microsoft bei .NET an mehreren Stellen abweicht (etwa das Verständnis von Feldern, Attributen und Eigenschaften oder die Sichtbarkeitsmodifizierer). UML (Unified Modeling Language) ist eine Standardsprache zur Spezifikation, Visualisierung, Konstruktion und Dokumentation der Bestandteile von Softwaresystemen. UML wurde von der Object Management Group (OMG) entwickelt und der Entwurf der Spezifikation UML 1.0 wurde der OMG im Januar 1997 vorgelegt. Ursprünglich wurde die UML entwickelt, um das Verhalten von komplexen Software- und Nicht-Software-Systemen zu erfassen, und ist inzwischen zu einem OMG-Standard geworden, über den diverse Vorgänge in der objekt-orientierten Analyse, dem objektorientierten Design und der objektorientierten Programmierung einheitlich abgebildet werden können – wenn sie mit den UML-Vorgaben in Einklang zu bringen sind.



Klassendiagramm des Projekts „Zahlen3.sln“



- ✓ Mit den Symbolen der Symbolleiste *Klassen-Designer* ① können Sie das Diagramm bearbeiten.
- ✓ Falls Sie die Symbolleiste versehentlich ausgeblendet haben, können Sie sie mit dem Menüpunkt *Ansicht - Symbolleisten - Klassen-Designer* wieder einblenden.
- ✓ Über den Menüpunkt *Datei - Drucken* können Sie das Diagramm ausdrucken.

### Klassen aus dem Diagramm entfernen

- Klicken Sie im Diagramm auf eine Klasse, um sie zu markieren.
- Entfernen Sie die Darstellung der Klasse im Diagramm mit der [Entf]-Taste.  
Der Code der Klasse ist weiterhin im Projekt vorhanden.

Um auch den Code der Klasse zu entfernen, betätigen Sie die Tastenkombination [Strg] [Entf]. Die C#-Datei ist im Projekt zwar weiterhin vorhanden; der Code der Klasse wird aber ohne weitere Nachfrage entfernt.

### Weitere Klassen des Projekts im Diagramm darstellen

Wenn Sie eine Klasse aus dem Diagramm entfernt haben, können Sie diese erneut darstellen. Außerdem können Sie weitere Klassen des Projekts jederzeit in das Diagramm aufnehmen.

- Markieren Sie im Projektmappen-Explorer die C#-Dateien, deren enthaltene Klasse Sie in das Diagramm einfügen möchten.
- Ziehen Sie die markierten Dateien an die gewünschte Position im Diagramm.

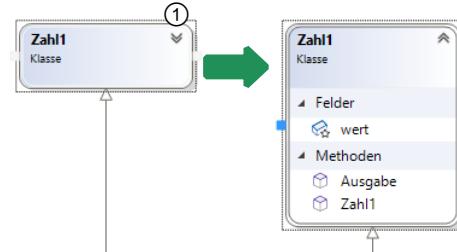
## Klassen positionieren

- ▶ Markieren Sie die Klasse und verschieben Sie sie an die gewünschte Position.
  - oder Rufen Sie den Menüpunkt *Klassendiagramm - Layout für Diagramm* auf, um die Symbole automatisch anordnen zu lassen.
- Alternative: Symbol  in der Symbolleiste *Klassen-Designer*

## Member einer Klasse anzeigen

- ▶ Klicken Sie in der Klassendarstellung der entsprechenden Klasse auf das Symbol  ①, um die Member der Klasse anzeigen zu lassen.

Die Member werden aufgelistet und dabei nach der Memberart, z. B. nach Feldern, Methoden und Konstruktoren, gruppiert.



Mit Symbolen oder entsprechenden Menüpunkten können Sie die Darstellung beeinflussen:

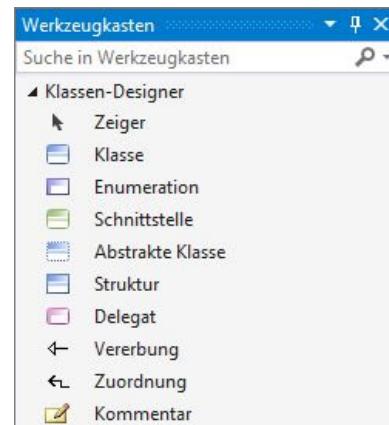
- ✓ Über die Symbole ,  und  bzw. die Befehle im Menü *Klassendiagramm - Memberformat ändern* lassen Sie entweder nur den Namen, den Namen und den Datentyp oder die vollständige Signatur der Member anzeigen.
- ✓ Über die Befehle im Menü *Klassendiagramm - Member gruppieren* können Sie festlegen, ob die Member nach der Art oder den Zugriffsrechten gruppiert oder alphabetisch sortiert aufgelistet werden.

Mit dem Symbol  können Sie die Member wieder ausblenden.

## Kommentarelemente hinzufügen

Mit sogenannten Kommentarelementen haben Sie die Möglichkeit, ein Klassendiagramm zu beschreiben und zu dokumentieren.

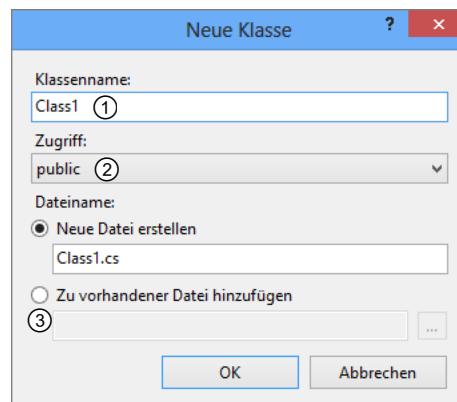
- ▶ Öffnen Sie den Werkzeugkasten.  
Er zeigt statt der Windows Forms-Elemente jetzt Elemente des Klassendiagramms.
- ▶ Ziehen Sie das Element *Kommentar* aus dem Werkzeugkasten an die gewünschte Position im Diagramm.  
Alternative: *Klassendiagramm - Hinzufügen - Kommentar*  
Ein Kommentarfeld wird eingefügt.
- ▶ Klicken Sie in das markierte Kommentarfeld ① und geben Sie den gewünschten Text ein.



## Neue Klassen erzeugen

- ▶ Öffnen Sie den Werkzeugkasten und ziehen Sie das Element *Klasse* an die gewünschte Position im Diagramm.  
Alternative: *Klassendiagramm - Hinzufügen - Klasse*
- ▶ Geben Sie im geöffneten Dialogfenster den Namen der Klasse ein ①.
- ▶ Legen Sie die Zugriffsrechte fest ②.
- ▶ Bestimmen Sie, ob die Klasse in einer neuen oder einer bestehenden C#-Datei erstellt werden soll ③.

Die Klasse wird im Diagramm angezeigt und der Programmcode für die Klasse wird generiert.



## Vererbungsbeziehung einrichten

Auch eine Vererbungsbeziehung können Sie im Klassendiagramm festlegen.

- ▶ Klicken Sie im Werkzeugkasten auf das Element *Vererbung*.
- ▶ Klicken Sie auf die Klasse, die als abgeleitete Klasse von einer anderen Klasse erben soll, und halten Sie die Maustaste gedrückt.
- ▶ Ziehen Sie die Maus auf die Klasse, die als Basisklasse dienen soll, und lassen Sie die Maustaste wieder los.



Im Programmcode wird die Vererbung automatisch eingetragen.

Der Vererbungspfeil zeigt immer von der abgeleiteten Klasse zur Basisklasse.

## Klassen bearbeiten

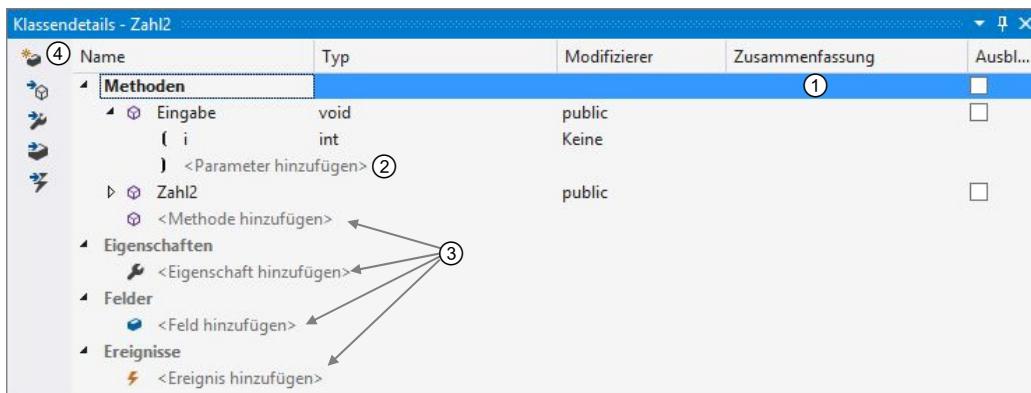
|                                                |                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Klasse bzw. Member umbenennen</b>           | <ul style="list-style-type: none"> <li>▶ Klicken Sie auf die Klasse bzw. den Member, um diese(n) zu markieren.</li> <li>▶ Klicken Sie auf den entsprechenden Bezeichner, um in den Bearbeitungsmodus zu gelangen, und geben Sie den neuen Namen ein.<br/>Alle Vorkommen des Namens werden im Programmcode angepasst.</li> </ul> |
| <b>Neue Member erstellen</b>                   | <ul style="list-style-type: none"> <li>▶ Markieren Sie die Klasse im Klassendiagramm und wählen Sie im Menü <i>Klassendiagramm - Hinzufügen</i> den gewünschten Befehl.</li> </ul>                                                                                                                                              |
| <b>Zum Programmcode einer Klasse wechseln</b>  | <ul style="list-style-type: none"> <li>▶ Klicken Sie im oberen Bereich der Klassendarstellung doppelt in einen freien Bereich der Klasse.</li> </ul>                                                                                                                                                                            |
| <b>Zum Programmcode eines Members wechseln</b> | <ul style="list-style-type: none"> <li>▶ Klicken Sie im Klassendiagramm doppelt auf den Membernamen.</li> </ul>                                                                                                                                                                                                                 |

Die Änderungen werden nicht nur ins Diagramm, sondern automatisch auch in den Programmcode übernommen.

## Mit dem Fenster *Klassendetails* arbeiten

Unterhalb des Diagramms wird standardmäßig das Fenster *Klassendetails* eingeblendet.

- ✓ Falls ein anderes Fenster angezeigt wird, wechseln Sie über das entsprechende Register in das Fenster *Klassendetails*.
- ✓ Falls das Fenster geschlossen wurde, können Sie es einblenden, sofern das Klassendiagramm angezeigt wird.
- ✓ Zum Einblenden des Fensters wählen Sie im Menü *Ansicht - Weitere Fenster - Klassendetails*.



|                                                                    |                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Details zu einer Klasse anzeigen</b>                            | ► Markieren Sie die Klasse im Klassendiagramm.                                                                                                                                                                                                                                                                                             |
| <b>Methoden (oder Konstruktoren) im Klassendiagramm ausblenden</b> | ► Aktivieren Sie das entsprechende Kontrollfeld ①.                                                                                                                                                                                                                                                                                         |
| <b>Detailinformationen ein- bzw. ausblenden</b>                    | ✓ Verwenden Sie die Symbole  und , um die Member eines Typs ein- bzw. auszublenden.<br>✓ Verwenden Sie die Symbole  und , um Detailinformationen zu einem Member, z. B. zu einer Methode, zu erhalten bzw. auszublenden.                                                                                                                   |
| <b>Parameter zu einer Methode hinzufügen</b>                       | ► Klicken Sie auf den Eintrag <Parameter hinzufügen> ② und geben Sie in den entsprechenden Spalten den Namen, den Typ und den Modifizierer ein.                                                                                                                                                                                            |
| <b>Weitere Member hinzufügen</b>                                   | ► Klicken Sie auf den gewünschten Eintrag, z. B. auf <Methode hinzufügen> oder <Eigenschaft hinzufügen> ③.<br>► Definieren Sie den Member mit einem Namen und geben Sie ggf. den Datentyp und den Modifizierer ein.<br>Über die Pfeilschaltfläche des Symbols  ④ und den entsprechenden Eintrag können Sie ebenfalls neue Member erzeugen. |

Auch die Änderungen im Fenster *Klassendetails* werden in den Programmcode übernommen.

## 10.5 Übungen

### Übung 1: Vererbung

**Übungsdatei:** --

**Ergebnisdatei:** *Datum.sln*

1. Erstellen Sie eine neue Konsolenanwendung.
2. Entwickeln Sie analog zum Beispiel *Zahlen.sln* aus diesem Kapitel eine Klasse, die ein Datum als Date speichern und mit einer Methode auf dem Bildschirm ausgeben kann. Erzeugen Sie eine davon abgeleitete Klasse, die eine zusätzliche Methode zum Speichern eines Datumswertes besitzt. Testen Sie die Methoden mithilfe einer Instanz der abgeleiteten Klasse.

## Übung 2: Vererbung

**Übungsdatei:** Kap10\SomeCars\SomeCars.sln

**Ergebnisdatei:** SomeCars2.sln

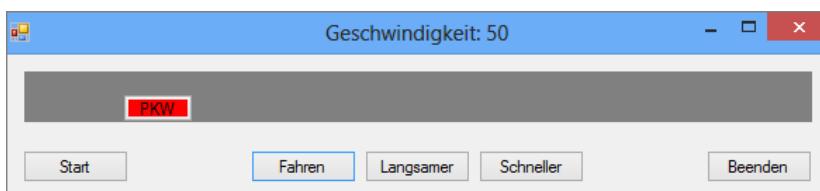
1. Erstellen Sie eine neue Konsolenanwendung.
2. Erweitern Sie die erste Übung des Kapitels 9 um eine abgeleitete Klasse PKW. Implementieren Sie darin ein zusätzliches Feld zum Speichern der Anzahl der Sitzplätze und eine Eigenschaft zum Schreib-/Lesezugriff auf dieses Feld.
3. Legen Sie einen Konstruktor an, der den Konstruktor der Basisklasse zur Initialisierung der Felder für Farbe und Baujahr aufruft und selbst nur das zusätzliche Feld (Sitze) initialisiert.
4. Passen Sie das Programm so an, dass es auch mit Instanzen der abgeleiteten Klasse arbeitet und die Werte aller drei Felder (Farbe, Baujahr, Sitzplätze) auf dem Bildschirm ausgibt. Demonstrieren Sie die Funktionalität mit je einer Instanz der Basisklasse und der abgeleiteten Klasse.
5. Verwenden Sie für die drei Klassen des Programms drei separate Dateien.
6. Erzeugen Sie ein Klassendiagramm und blenden Sie darin alle Member und jeweils auch die Datentypen ein.

## Übung 3: Fahrzeug bewegen

**Übungsdatei:** --

**Ergebnisdatei:** Autofahren.sln

1. Erstellen Sie eine neue Windows-Anwendung, deren Ziel es ist, die Bewegung eines Fahrzeugs auf einer Straße zu simulieren.
2. Als Straße verwenden Sie eine Komponente Panel. Der "PKW" soll dynamisch erzeugt werden. Legen Sie dazu in einer separaten Datei eine Klasse Auto an, die von der Klasse System.Windows.Forms.Button abgeleitet ist.
3. In der Klasse Auto implementieren Sie ein geeignetes Feld sowie eine Schreib-/Lese-Eigenschaft für die Geschwindigkeit.



4. Fügen Sie der Klasse eine Methode für das Fahren des Fahrzeugs hinzu. Die Bewegung realisieren Sie durch das Erhöhen des Wertes für die Eigenschaft Left des PKW-Objekts. Mithilfe der Anweisung Application.DoEvents() erreichen Sie, dass die Anwendung während der Fahrzeuggbewegung getätigte Benutzereingaben verarbeitet. Über die folgende Anweisung lässt sich der Ablauf des Programms in Abhängigkeit von dem Geschwindigkeitswert verzögern und so eine einfache Steuerung der Fahrzeuggbewegung vornehmen:

```
System.Threading.Thread.Sleep(100 - Geschwindigkeit);
```

5. Über die Schaltflächen Langsamer und Schneller soll der Geschwindigkeitswert des PKW in 10er-Schritten verändert werden.
6. Am Ende der Straße (am rechten Rand der Panel-Komponente) soll das Fahrzeug automatisch stoppen.

# 11 Polymorphismus

## 11.1 Polymorphie in der Vererbung

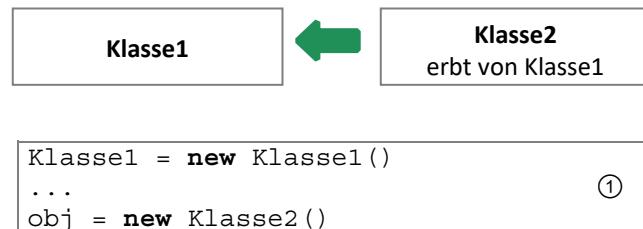
Bei der Vererbung werden Felder, Eigenschaften und Methoden der Basisklasse von den abgeleiteten Klassen übernommen, soweit sie von der abgeleiteten Klasse zugänglich sind. Die abgeleitete Klasse ist daher zunächst nahezu identisch mit der Basisklasse. In den meisten Fällen benötigt der Programmierer an den speziellen Zweck angepasste Klassen mit gemeinsamen und spezifischen Eigenschaften. Es gibt daher folgende Möglichkeiten, abgeleitete Klassen an die gewünschten Aufgaben anzupassen, wobei sich Änderungen in den abgeleiteten Klassen nicht auf die Vorfahren auswirken.

- ✓ Die abgeleiteten Klassen können um zusätzliche Member erweitert werden.
- ✓ Die geerbten Member können verborgen, überschrieben oder überladen werden.

Daraus ergibt sich, dass die gleichnamigen Member der veränderten abgeleiteten Klassen sehr vielseitig bzw. vielgestaltig (griechisch: polymorph) auftreten können.

### Kompatibilität von Objekten einer Klasse und Objekten der Basisklasse

- ✓ Objekte einer Klasse besitzen immer mindestens die nicht privaten Methoden und Felder der Basisklasse und sind somit vollständig als solche Objekte funktionstüchtig.
- ✓ Innerhalb der Klassenhierarchie sind Objekte abgeleiteter Klassen daher immer zu Objektvariablen der Basisklassen zuweisungskompatibel.
- ✓ Die Zuweisungskompatibilität erlaubt es daher, einer Objektvariablen vom Typ der Basisklasse eine Instanz einer abgeleiteten Klasse zuzuweisen.
- ✓ Eine Objektvariable kann somit auf Instanzen abgeleiteter Klassentypen verweisen ①. Wird z. B. eine Methode des zugewiesenen Objekts aufgerufen, wird automatisch die zu diesem Klassentyp gehörende Methode verwendet.



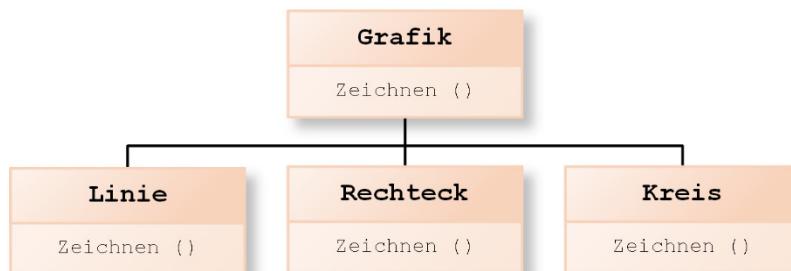
```

Klasse1 = new Klasse1()
...
obj = new Klasse2() ①

```

### Beispiel

Eine Klasse **Grafik** besitzt die Methode **Zeichnen()** und dient als Basisklasse für die drei abgeleiteten Klassen **Linie**, **Rechteck** und **Kreis**. Die Methoden werden in den abgeleiteten Klassen so angepasst, dass sie bei Aufruf die entsprechende geometrische Figur auf dem Bildschirm zeichnen. Von den abgeleiteten Klassen wird je eine Instanz erzeugt. Diese Instanzen werden nacheinander einer Objektvariablen vom Typ der Basisklasse **Grafik** zugewiesen. Über die Objektvariable wird jeweils die Methode **Zeichnen()** aufgerufen. Trotz des identischen Aufrufs **Objektvariable.Zeichnen()** wird immer die zur zugewiesenen Instanz gehörende Methode ausgeführt, die dann eine Linie, ein Rechteck oder einen Kreis zeichnet.



**Beispiel: SomeObjects.sln**

Da die Klasse `Object` die Basisklasse aller Klassen ist, sind alle Klassen zur Klasse `Object` zuweisungskompatibel. Bei der Verwendung des Datentyps `object` (Schlüsselwort) nutzen Sie die Klasse `Object` der .NET Frameworks.

```

① object obj;
② obj = "eine Zeichenkette";
③ Console.WriteLine(obj.GetType().Name); // => String
④ obj = 5;
⑤ Console.WriteLine(obj.GetType().Name); // => Int32
Console.WriteLine(typeof(object)); // => System.Object
⑥ if (obj is int)
 Console.WriteLine("Integer"); // => Integer

```

- ① Eine Variable `obj` vom Typ `object` wird deklariert.
- ②/④ Der Variablen `obj` wird ein String und in Zeile ④ eine Zahl zugewiesen.
- ③/⑤ Mithilfe der Methode `GetType()` kann der jeweils aktuelle Datentyp der Variablen `obj` ermittelt werden.
- ⑥ Über den Operator `is` lässt sich der Typ einer Variablen überprüfen. In diesem Beispiel wird geprüft, ob `obj` vom Typ `int` ist.

## 11.2 Member verbergen

Das Prinzip beim Verbergen eines Members besteht darin, in einer abgeleiteten Klasse den Bezeichner eines geerbten Members neu zu vergeben. Dadurch wird für die Instanz der abgeleiteten Klasse das neu deklarierte Element aufgerufen. Es ist aber weiterhin möglich, innerhalb der abgeleiteten Klasse mittels des Schlüsselworts `base` das verdeckte Element des Vorfahren aufzurufen.

**Syntax**

- ✓ Die (erneute) Deklaration eines Members erfolgt, wie an den entsprechenden Stellen bereits beschrieben wurde. Der zusätzliche Modifizierer `new` (nicht zu verwechseln mit dem Operator `new`) zeigt an, dass der Member neu deklariert und in einer Basisklasse verborgen wird.
- ✓ Der Datentyp spielt beim Verbergen von Membern keine Rolle. Beispielsweise kann eine `int`-Eigenschaft eine Methode verbergen oder umgekehrt. Wenn Sie eine Methode durch eine andere Methode verbergen, können Sie eine andere Parameterliste und/oder einen anderen Rückgabetyp verwenden.
- ✓ Verborgene Member der Basisklasse stehen bei Bedarf innerhalb der abgeleiteten Klasse über das vorangestellte Schlüsselwort `base` zur Verfügung ①.

```

new memberdeclaration
...
base.Name... ①

```

**Beispiel: Zahlen4.sln**

Das Beispiel zeigt, wie sowohl auf ein neu deklariertes als auch auf ein dadurch verborgenes Element zugegriffen wird. Die Basisklasse besitzt das Feld `wert` vom Typ `int`. Die abgeleitete Klasse deklariert dieses Element neu als `string`, um den Wert des Feldes auch als Zeichenkette ausgeben zu können. Dadurch wird das geerbte Feld verborgen bzw. überschattet. Dass es trotzdem noch vorhanden ist, wird durch die Zuweisung eines Wertes und die anschließende Ausgabe dieses Wertes demonstriert.

```

class Zahl1
{
 protected int wert;
}

class Zahl2 : Zahl1
{
 new protected string wert;
 public Zahl2(int i, string s)
 {
 base.wert = i;
 wert = s;
 }
 public void Ausgabe()
 {
 Console.WriteLine(wert);
 Console.WriteLine(base.wert);
 }
}

class Program
{
 static void Main(string[] args)
 {
 5 Zahl2 eineZahl = new Zahl2(10, "Zehn");
 6 eineZahl.Ausgabe();
 }
}

```

- ① Die Basisklasse erhält bei der Deklaration das Feld `wert` vom Datentyp `int`.
- ② Bei der Vereinbarung der abgeleiteten Klasse wird das Feld `wert` neu als Zeichenkette deklariert.
- ③ An dieser Stelle wird der Konstruktor vereinbart. Er initialisiert sowohl das neu deklarierte als auch das verborgene Feld.
- ④ Die Methode `Ausgabe()` zeigt den Wert des von der Basisklasse geerbten Feldes und des verborgenen Feldes der abgeleiteten Klasse auf dem Bildschirm an.
- ⑤ Eine Instanz der abgeleiteten Klasse `Zahl2` wird erzeugt, initialisiert und der Objektvariablen `eineZahl` zugewiesen.
- ⑥ Die Felder werden mithilfe der Methode `Ausgabe()` ausgegeben.

### 11.3 Member überschreiben

Methoden, Eigenschaften und auch Indexer, die Sie in Kapitel 13 kennenlernen, können in abgeleiteten Klassen überschrieben werden, sofern diese Möglichkeit bei der Deklaration der Basisklasse freigegeben wurde. Im Gegensatz zum Verbergen müssen dabei Typen bzw. Rückgabetypen und eventuell vorhandene Parameter übereinstimmen.

### Beispiel: *Print.sln*

Im folgenden Beispiel wird eine Basisklasse `Klasse1` mit der überschreibbaren Methode `Print()` deklariert. In der abgeleiteten Klasse `Klasse2` erfolgt das Überschreiben der Methode `Print()`. Abschließend wird die überschriebene Methode der abgeleiteten Klasse aufgerufen.

```
...
class Klasse1
{
 public virtual void Print() // Die Methode wird als überschreibbar deklariert.
 {
 ...
 }
}

class Klasse2 : Klasse1
{
 public override void Print() // Die Methode Print() wird überschrieben.
 {
 ...
 }
}
...
Klasse2 obj = new Klasse2(); // Hier wird eine Instanz der Klasse2 erzeugt.
obj.Print(); // Die überschriebene Methode wird ausgeführt.
...
```

### Syntax für das Überschreiben von Membern

- ✓ Ereignisse und Felder können nicht überschrieben werden, sondern nur Eigenschaften, Methoden und Indexer.
  - ✓ Fügen Sie in der Basisklasse vor der Eigenschafts-, Methoden- oder Indexer-Deklaration das Schlüsselwort `virtual` hinzu.  
Dadurch wird das Überschreiben des Klassen-elements (Members) in einer abgeleiteten Klasse gestattet.
  - ✓ In abgeleiteten Klassen verwenden Sie das Schlüsselwort `override` (überschreiben), um ein Klassen-element der Basisklasse zu überschreiben.
  - ✓ Die Parameterliste (die Reihenfolge und der Typ der Parameter) und der Typ des Rückgabewertes (falls vorhanden) müssen mit der Deklaration in der Basisklasse übereinstimmen.
  - ✓ Die Methode der Basisklasse existiert dann in der abgeleiteten Klasse nicht mehr. Sie kann jedoch bei Bedarf über das Schlüsselwort `base` eingebunden werden.
  - ✓ Im Unterschied zum Verbergen von Membern müssen die Zugriffsebenen (`public`, `internal` usw.) des überschreibenden und des ursprünglichen, überschriebenen Members übereinstimmen.
- ```
// in der Basisklasse
[modifier] virtual void Identifier(parameterlist)
{
    ...
}

// in den abgeleiteten Klassen
[modifier] override void Identifier(parameterlist)
{
    ...
}
```

Als `virtual` deklarierte Methoden bieten den Vorteil, dass Sie über eine Objektvariable der Basisklasse auf die mit `override` überschriebenen Methoden der abgeleiteten Klasse zugreifen können, wenn der Objektvariablen eine Instanz der abgeleiteten Klasse zugewiesen wurde. Ohne die Vereinbarung mit `virtual` kann nur auf die von der Basisklasse geerbten Methoden zugegriffen werden.

Beispiel: Zahlen5.sln

Das Beispiel lehnt sich weitgehend an das letzte an (*Zahlen4.sln*). Die Methode `Ausgabe()` der Basisklasse `Zahl1` wird im folgenden Beispiel jedoch als überschreibbar deklariert und in der abgeleiteten Klasse `Zahl2` überschrieben. In Abhängigkeit vom Objekt, welches sich hinter der Objektvariablen `eineZahl` verbirgt, wird die Methode der Basisklasse oder die Methode der abgeleiteten Klasse ausgeführt.

```
① class Zahl1
{
    protected int wert;

②     internal virtual void Ausgabe()
    {
        Console.WriteLine(wert);
    }
}

class Zahl2 : Zahl1
{
    new protected string wert;

    public Zahl2(int i, string s)
    {
        base.wert = i;
        wert = s;
    }

③     internal override void Ausgabe()
    {
        base.Ausgabe();
        Console.WriteLine(wert);
    }
}

class Program
{
    static void Main(string[] args)
    {

④         Zahl1 eineZahl;
        eineZahl = new Zahl1();
⑤         eineZahl.Ausgabe();
⑥         eineZahl = new Zahl2(10, "Zehn");
⑦         eineZahl.Ausgabe();
    }
}
```

- ① Die Basisklasse `Zahl1` wird deklariert.
- ② Die Methode `Ausgabe()` wird als überschreibbar gekennzeichnet. Die Methode `Ausgabe()` gibt den Wert des Feldes `wert` auf dem Bildschirm aus.
- ③ Bei der Deklaration der abgeleiteten Klasse `Zahl2` wird die Methode `Ausgabe()` der Basisklasse überschrieben. Sie ist in der Lage, durch zusätzlichen Aufruf der überschriebenen Methode der Basisklasse sowohl den Wert des verborgenen als auch den Wert des neu deklarierten Feldes `wert` anzuzeigen.
- ④ Eine Objektvariable `eineZahl` vom Typ der Basisklasse wird vereinbart. Ihr wird anschließend eine Instanz der Basisklasse zugewiesen.
- ⑤ Die Methode `Ausgabe()` der Basisklasse wird aufgerufen.

- ⑥ Der Objektvariablen wird nun eine Instanz der abgeleiteten Klasse zugewiesen.
- ⑦ Anschließend wird die Methode `Ausgabe()` der abgeleiteten Klasse aufgerufen und ausgeführt. An dieser Stelle wird deutlich, was Polymorphismus bedeutet: Ein Aufruf eines gleichnamigen Members führt zu unterschiedlichen Ergebnissen, je nachdem, auf welches Objekt von der Objektvariablen verwiesen wird.

11.4 Member überladen

Sie können Methoden im selben Gültigkeitsbereich mit identischen Namen deklarieren. Dabei ist es unerheblich, ob die Methoden innerhalb einer Klasse deklariert werden oder sich in verschiedenen Klassen befinden, die durch Vererbung miteinander verbunden sind. Dieses Verfahren wird auch **Überladen** genannt. Überladene Methoden müssen sich in Parameterzahl und/oder -typ unterscheiden.

Die Unterscheidung nur durch den Rückgabetyp ist nicht möglich, da Methoden auch ohne Auswertung des Rückgabewertes ausgerufen werden können. Überladungen sind z. B. dann sinnvoll, wenn gleichartige Aktionen mit verschiedenen Datentypen oder mit unterschiedlich vielen Parametern ausgeführt werden sollen. Der Compiler entscheidet dann zur Laufzeit, welche der Überladungen für die aktuellen Parameter genutzt wird.

Syntax für das Überladen einer Methode

```
class Classname1
{
    [modifier] type|void Identifier(parameterlist1)
    {...}
}
class Classname2 : Classname1
{
    [modifier] type|void Identifier(parameterlist2)
    {...}
}
```

Unterschiedliche Parameterlisten

- ✓ Verwenden Sie bei der Deklaration den gleichen Bezeichner für alle Überladungen einer Methode oder einer Eigenschaft.
- ✓ Die verwendeten Parameter müssen sich in der Anzahl und/oder in mindestens einem Typ unterscheiden.
- ✓ Der Rückgabewert wird beim Überladen nicht zur Unterscheidung herangezogen. Aus diesem Grund können Sie gleichnamige Methoden mit und ohne Rückgabewert überladen.

Beispiel: *OverloadMember.sln*

Im folgenden Programm wird die Methode `Plus()` der Basisklasse `Klasse1` in der abgeleiteten Klasse `Klasse2` so überladen, dass sie, statt zwei `int`-Zahlen zu addieren, die ASCII-Code-Werte zweier Zeichen addiert. In Abhängigkeit von den übergebenen Parametern wird automatisch die dazu passende Methode ausgeführt.

```
① class Klasse1
{
    internal int Plus(int i, int j)
    {
        ②     return i + j;
    }
}
```

```

③ class Klasse2 : Klasse1
{
    internal int Plus(char s, char t)
    {
        return Convert.ToByte(s) + Convert.ToByte(t);
    }
}

class Program
{
    static void Main(string[] args)
    {
        ⑤ Klasse2 einObjekt = new Klasse2();
        Console.WriteLine(einObjekt.Plus(40, 60));
        Console.WriteLine(einObjekt.Plus('a', 'c'));
    }
}

```

- ① Die Basisklasse `Klasse1` mit der Methode `Plus()` wird deklariert.
- ② Die Methode `Plus()` gibt die Summe der beiden übergebenen Parameter vom Typ `int` zurück.
- ③ Hier erfolgt die Vereinbarung der abgeleiteten Klasse `Klasse2` mit ihrer überladenen Methode `Plus()`.
- ④ Diese Überladung der Methode `Plus()` konvertiert die Zeichen in den Typ `byte` und gibt die Summe der beiden Werte zurück.
- ⑤ Eine Objektvariable vom Typ `Klasse2` wird vereinbart. Ihr wird eine Instanz dieser Klasse zugewiesen.
- ⑥ Die Methode `Plus()` wird zweimal mit unterschiedlichen Parametertypen aufgerufen.

11.5 Abstrakte Klassen und Methoden

Abstrakte Klassen implementieren ihre Methoden und Eigenschaften nur teilweise oder auch gar nicht. Die Implementierung der abstrakten Methoden muss in den abgeleiteten Klassen erfolgen. Von abstrakten Klassen können keine Instanzen gebildet werden. Mithilfe abstrakter Klassen können Sie eine gewisse Grundfunktionalität für alle abgeleiteten Klassen zur Verfügung stellen. Außerdem können Sie erzwingen, dass bestimmte Elemente in den abgeleiteten Klassen implementiert und angepasst werden.

Syntax für abstrakte Klassen und abstrakte Methoden bzw. Eigenschaften

- ✓ Um eine Klasse als **abstrakte Basis-Klasse** zu kennzeichnen, fügen Sie das Schlüsselwort `abstract` der Klassendeklaration hinzu ①.
- ✓ Methoden oder Eigenschaften, die in allen abgeleiteten Klassen speziell implementiert werden müssen, werden als **abstrakte Methoden** bzw. **abstrakte Eigenschaften** implementiert und werden mit dem Schlüsselwort `abstract` gekennzeichnet ②.

```

abstract class Klassenname ①
{
    modifier abstract type|void Methodname(); ②
    [virtual] type|void Methodname2();③
    {
    ...
    }
}

```

- ✓ Sie können auch in abstrakten Klassen einige oder alle Methoden oder Eigenschaften vollständig implementieren ③. Diese Implementierung wird dann von allen abgeleiteten Klassen geerbt. Durch die Angabe von `abstract` wird allerdings sichergestellt, dass die Klasse in keinem Fall direkt zur Objekterzeugung verwendet werden kann. Mit der Angabe des Schlüsselworts `virtual` können Sie das Überschreiben der Implementierung in den abgeleiteten Klassen gestatten.

Beispiel: *Datumsformate.sln*

Eine abstrakte Klasse `Datum` dient als Basisklasse für zwei Klassen `DatumsForm1` und `DatumsForm2`. Die Objekte dieser abgeleiteten Klassen sollen das aktuelle Datum (`DateTime.Now`) speichern und in unterschiedlichen Formaten ausgeben können. Die Implementierung der Methoden erfolgt ausschließlich in den abgeleiteten Klassen. Von diesen werden anschließend Instanzen erzeugt. Durch das Aufrufen der Methoden wird deutlich, dass je nach Typ des Objekts die dazugehörige Methode ausgeführt wird.

```

① abstract class Datum
{
    protected DateTime datumsWert;
    abstract public void Ausgabe();
}

② class DatumsForm1 : Datum
{
    public override void Ausgabe()
    {
        datumsWert = DateTime.Now;
        Console.WriteLine(datumsWert.ToString("d"));
    }
}

③ class DatumsForm2 : Datum
{
    public override void Ausgabe()
    {
        datumsWert = DateTime.Now;
        Console.WriteLine(datumsWert.ToString("F"));
    }
}

class Program
{
    static void Main(string[] args)
    {
        ④ Datum einDatum;
        einDatum = new DatumsForm1();
        ⑤ einDatum.Ausgabe();
        einDatum = new DatumsForm2();
        ⑥ einDatum.Ausgabe();
    }
}

```

- ① Die abstrakte Basisklasse `Datum` wird mit ihrer abstrakten Methode `Ausgabe()` deklariert.
- ② Die abgeleitete Klasse `DatumsForm1` wird deklariert. Die abstrakte Methode der Basisklasse muss in dieser abgeleiteten Klasse implementiert werden. Sie speichert das aktuelle Datum (`DateTime.Now`) im Feld `datumsWert` und gibt es im langen Datumsformat auf dem Bildschirm aus.

- ③ An dieser Stelle erfolgt die Implementierung der abgeleiteten Klasse DatumsForm2. Die Methode Ausgabe () dieser Klasse speichert ebenfalls das aktuelle Datum und gibt es im kurzen Datumsformat aus.
- ④ Eine Objektvariable vom Typ der Basisklasse wird vereinbart. Ihr wird anschließend eine Instanz der abgeleiteten Klasse DatumsForm1 zugewiesen.
- ⑤ Die zur Klasse DatumsForm1 gehörende Methode Ausgabe () wird ausgeführt.
- ⑥ Hier wird eine Instanz der abgeleiteten Klasse DatumsForm2 erzeugt und der Objektvariablen zugewiesen.
- ⑦ An dieser Stelle wird die Methode der Klasse DatumsForm2 ausgeführt.

Freitag, 22. Mai 2020
22.05.2020

Die Ausgabe des Programms

Die zu implementierenden Methoden schnell einfügen

Wenn Sie von einer abstrakten Klasse erben, können Sie schnell das Grundgerüst für alle zu implementierenden Methoden und Eigenschaften erzeugen.

- ▶ Schreiben Sie die Deklaration der Klasse einschließlich der Vererbung.
- ▶ Klicken Sie mit der rechten Maustaste auf den Namen der Basisklasse ① und wählen Sie im Kontextmenü den Befehl *Abstrakte Klasse implementieren*.

①
class Classname : BaseClass

Es wird allerdings nur ein Methodenstub ohne sinnvolle Implementierung generiert, der nur rein formal den notwendigen Anforderungen entspricht.

Übersicht

		Verbergen	Überschreiben	Überladen	Abstraktion
Member		alle	Eigenschaften, Methoden, Indexer	Eigenschaften, Methoden, Indexer	Eigenschaften, Methoden, Indexer
Schlüsselwörter	in Basisklasse		virtual		abstract
	in abgeleiteter Klasse	new	override		override

11.6 Typprüfung und -konvertierung

Die Operatoren `is` und `as` erleichtern den Umgang mit dem polymorphen Erscheinungsbild von Klasseninstanzen.

- ✓ Mit dem Operator `is` können Sie zur Laufzeit eine Typprüfung von Klassentypen vornehmen.
- ✓ Den Operator `as` können Sie für Typkonvertierungen (Typumwandlungen) verwenden.

Syntax für die Verwendung von `is` und `as`

- ✓ Ein mit dem Operator `is` versehener Ausdruck ergibt `true`, wenn die Instanz vom Typ der nach `is` angegebenen Klasse oder einer davon abgeleiteten Klasse ist. Sonst wird `false` zurückgegeben. Sie können diesen Ausdruck z. B. in einer `if`-Anweisung verwenden.
- ✓ Der Operator `as` wandelt eine Objektvariable in den angegebenen Typ (`ClassType`) um. Diese Typumwandlung ist nur mit Instanzen einer Klasse und ihrer abgeleiteten Klassen (Nachkommen) möglich. Ob eine Instanz für ein solches Typecasting geeignet ist, lässt sich vorher mit dem Operator `is` prüfen. Ist die Umwandlung nicht möglich, liefert der Operator `as` Ergebnis `null`.
- ✓ Sie können einen Ausdruck mit dem Operator `as` auch als Qualifizierer einsetzen, um auf die Methoden verwandter Typen zuzugreifen.

```
if (objectName is ClassType)
{
    ...
}
```

```
objectName1 = objectName as ClassType;
```

```
(objectName as ClassType).Method();
...
```

Bei Bedarf ist es möglich, einer Objektvariablen gleich bei der Deklaration den Typ einer abgeleiteten Klasse zuzuweisen.

```
baseClass ObjektName = new descendantClass();
```

Beispiel: *Konvert.sln*

Im folgenden Programm werden drei Instanzen von verschiedenen Klassen erzeugt, wobei zwei Klassen miteinander verwandt sind. Mithilfe des Operators `is` werden drei Instanzen dieser Klassen daraufhin geprüft, ob sie Instanzen der Klasse `Klasse1` (bzw. von deren Nachkommen) sind. Abschließend wird eine Methode mithilfe der Typumwandlung `as` aufgerufen.

```
① class Klasse1
{
    public void Test()
    {
        Console.WriteLine("Die Methode \"Test\" von Klasse1 wurde ausgeführt.");
    }
}

class Klasse2 : Klasse1
{
    new public void Test()
    {
        Console.WriteLine("Die Methode \"Test\" von Klasse2 wurde ausgeführt.");
    }
}

class Klasse3
{
    public void Test()
    {
        Console.WriteLine("Die Methode \"Test\" von Klasse3 wurde ausgeführt.");
    }
}

class Program
{
    ② static void pruefen(object sender)
    {
        if (sender is Klasse1)
        {
            Console.WriteLine("Das ist eine Instanz der Klasse1 \"Klasse1\" +
                "oder einer davon abgeleiteten Klasse");
        }
        else
        {
            Console.WriteLine("Das ist keine Instanz der Klasse1 \"Klasse1\" +
                "oder einer davon abgeleiteten Klasse");
        }
    }

    static void Main(string[] args)
    {
        ③ Klasse1 instanz1 = new Klasse1();
        Klasse2 instanz2 = new Klasse2();
        Klasse3 instanz3 = new Klasse3();

        ④ instanz1.Test();
        pruefen(instanz1);
        instanz2.Test();
        pruefen(instanz2);
        instanz3.Test();
        pruefen(instanz3);
        (instanz2 as Klasse1).Test();
    }
}
```

- ① Die Klassen Klasse1, Klasse2 und Klasse3 werden deklariert. Klasse2 wird von Klasse1 abgeleitet. Alle Klassen erhalten je eine Methode Test(), die ihren Aufruf auf dem Bildschirm anzeigt.

- ② Eine Methode `pruefen()` wird deklariert, die mithilfe des Operators `is` feststellt, ob eine übergebene Objektvariable auf eine Instanz vom Typ `Klasse1` (oder eines davon abgeleiteten Typs) verweist.
- ③ Von jedem der drei Klassentypen wird je eine Objektvariable vereinbart, der gleichzeitig eine Instanz der entsprechenden Klasse zugewiesen wird.
- ④ Die Methode `Test()` aller drei Instanzen wird aufgerufen. Außerdem werden die drei Instanzen mithilfe der Methode `pruefen()` getestet.
- ⑤ An dieser Stelle erfolgt eine Typumwandlung mithilfe des Operators `as`. Dabei wird die Instanz `instanz2` vom Typ `Klasse2` in eine Instanz vom Typ `Klasse1` konvertiert. Anschließend erfolgt der Aufruf der Methode `Test()` der Klasse `Klasse1`.

Die Methode "Test" von Klasse1 wurde ausgeführt.
 Das ist eine Instanz der Klasse "Klasse1" oder einer davon abgeleiteten Klasse.
 Die Methode "Test" von Klasse2 wurde ausgeführt.
 Das ist eine Instanz der Klasse "Klasse1" oder einer davon abgeleiteten Klasse.
 Die Methode "Test" von Klasse3 wurde ausgeführt.
 Das ist keine Instanz der Klasse "Klasse1" oder einer davon abgeleiteten Klasse.
 Die Methode "Test" von Klasse1 wurde ausgeführt.

Die Ausgabe des Programms

Explizite und implizite Typumwandlungen

Wenn selbst definierte Typen in andere umgewandelt werden sollen, kann es notwendig sein, die Konvertierung ausdrücklich vorzugeben, um Datenverluste zu verhindern. Diesem Zweck dienen die Schlüsselwörter `explicit`, `implicit` und `operator`, mit deren Hilfe ein Konvertierungsoperator definiert werden kann.

Syntax

```
public static implicit|explicit operator TargetType (SourceType);
{
    ... // der Code für die spezielle Typumwandlung
    return instanceOfTargetType;
}
...
objectOfTargetType = (TargetType)objectOfSourceType;      // bei explicit
objectOfTargetType = objectOfSourceType;                  // bei implicit
```

- ✓ Ein Konvertierungsoperator muss als `static` und `public` vereinbart werden.
- ✓ Die implizite Konvertierung geschieht bei einer entsprechenden Zuweisung automatisch, während bei der expliziten Konvertierung der Ziieldatentyp in Klammern anzugeben ist.
- ✓ Die explizite Konvertierung ist vorzuziehen, da sie die vorgenommene Konvertierung im Quelltext deutlicher anzeigt.

Beispiel: `ExplicitConvert.sln`

Im folgenden Programm wird ein Konvertierungsoperator definiert, der einen Wert vom Typ `double` in eine Instanz vom Typ `Klasse1` konvertiert. Dazu erzeugt der Operator eine neue Instanz der Klasse `Klasse1` und weist den Feldern `wert` und `text` die `double`-Zahl einmal als Zahl und einmal als Zeichenkette zu. Anschließend gibt der Operator die Instanz zurück. Innerhalb der Methode `Main()` wird die Konvertierung getestet.

```

class Klasse1
{
    internal double wert;
    internal string text;
    public static explicit operator Klasse1(double d) // Konvertierungsoperator
    {
        Klasse1 c = new Klasse1(); // Instanz c wird erzeugt
        c.wert = d; // Zuweisung als Zahl
        c.text = d.ToString(); // Zuweisung als String
        return c; // Instanz c wird zurückgegeben
    }
}

class Program
{
    static void Main(string[] args)
    {
        double zahl = 34.2342;
        Klasse1 instanz = (Klasse1)zahl; // Aufruf des Konvertierungoperators
        Console.WriteLine("Wert: " + instanz.wert + "\nText: " + instanz.text);
    }
}

```

11.7 Operatoren überladen

Sie können die in Visual C# vorhandenen Operatoren durch Überladen für die Verwendung mit bestehenden oder selbst erstellten Klassen anpassen. Solche benutzerdefinierten Operatordefinitionen haben Vorrang vor den standardmäßig für alle Operatoren vorhandenen Definitionen. Allerdings können nicht alle Operatoren überladen werden. Die folgende Tabelle zeigt eine Auswahl an überladbaren Operatoren.

Unäre Operatoren (besitzen einen Operanden)	+ - ! ~ ++ -- true und false	*
Binäre Operatoren (besitzen zwei Operanden)	+ - * / % & ^ << >>	
Vergleichsoperatoren	== und != < und > <= und >=	*

Die mit *) gekennzeichneten Operatoren müssen immer **paarweise** überladen werden. Wenn Sie beispielsweise für eine Klasse den Operator == definieren, ist auch die Überladung des Operators != erforderlich.

Nachfolgend sind Anwendungsbeispiele aufgeführt, bei denen es sinnvoll sein könnte, Operatoren zu überladen, um bestimmte Operationen zu vereinfachen:

Zusammenfügen von Zeichenketten	In vielen Programmiersprachen (auch in Visual C#) ist der Operator + bereits standardmäßig überladen, um mit seiner Hilfe zwei Zeichenketten zu verknüpfen.
Mathematische Operationen mit komplexen Zahlen	Komplexe Zahlen besitzen einen Realteil und einen Imaginärteil. Diese Zahlen können als Klasse abgebildet werden. Bei mathematischen Operationen mit komplexen Zahlen müssen jeweils beide Anteile getrennt berechnet werden. Dazu werden die mathematischen Operatoren überladen.
Listen zusammenfügen	Zwei Listen aus mehreren Elementen mit einem neuen Operator + zu einer einzigen Liste zusammenfassen

Syntax für die Definition von Operatoren

```
public static ResultType operator OperatorSymbol(Type1 operand1, [Type2
operand2])
{
    ... // der Code für die spezielle Operation
    return result;
}
```

- ✓ Ein benutzerdefinierter Operator muss als `public` und als `static` vereinbart werden.
- ✓ Mindestens ein Operand muss als Typ die Klasse verwenden, in der der Operator definiert wird.
- ✓ Dem Ergebnistyp folgt das Schlüsselwort `operator` und die Angabe des Symbols des zu überladenden Operators. Danach folgen in Klammern die Operanden.
- ✓ Operatoren können nur als Wert übergeben werden (`ref` und `out` sind nicht erlaubt).
- ✓ Wenn als Ergebnis eine neue Instanz der Klasse zurückgegeben werden soll, ist innerhalb des Codeblocks mithilfe des Operators `new` ggf. eine neue Instanz zu erzeugen und entsprechend modifiziert zurückzugeben.

Beispiel: *OverloadOperator.sln*

In diesem Beispiel wird eine Klasse `Zeichen` deklariert, die als Feld ein Zeichen vom Typ `char` enthält. Der Operator `+` wird für den Fall überladen, dass zwei Instanzen der Klasse `Zeichen` mit seiner Hilfe verknüpft werden. In diesem Fall werden die beiden Zeichen zu einem String zusammengesetzt. Der String wird als Ergebnis zurückgegeben. Der Operator `*` wird ebenfalls überladen. Er bewirkt, dass ein String erzeugt wird, in dem das Zeichen mit einer bestimmten Anzahl wiederholt wird.

```
class Zeichen
{
    ① private char wert;
    public Zeichen(char c)
    {
        wert = c;
    }
    ② public static string operator +(Zeichen c1, Zeichen c2)
    {
        return "" + c1.wert + c2.wert;
    }
    ③ public static string operator *(int anzahl, Zeichen c)
    {
        string s = "";
        for (int i = 0; i < anzahl; i++)
        { s += c.wert; }
        return s;
    }
}
class Program
{
    static void Main(string[] args)
    {
        ④ Zeichen einZeichen = new Zeichen('x');
        Zeichen nochEinZeichen = new Zeichen('o');

    }
}
```

```

⑤ Console.WriteLine(einZeichen + nochEinZeichen);
⑥ Console.WriteLine(20 * einZeichen);
}
}

```

- ① Die Klasse Zeichen wird deklariert. Sie erhält ein Feld `wert` vom Typ `char`.
- ② Der Operator `+` wird überladen. Wenn zwei Operanden vom Typ `Zeichen` mit diesem Operator verbunden werden, werden die in den Feldern `wert` enthaltenen Zeichen zu einem String zusammengefügt. Der String wird zurückgegeben.
- ③ Auch ein Operator `*` wird für die Klasse `Zeichen` deklariert. Wenn eine Zahl vom Typ `int` mit einem Objekt vom Typ `Zeichen` mit diesem Operator verbunden wird, wird das im Feld `wert` enthaltene Zeichen entsprechend oft in einen String eingefügt. Der String wird zurückgegeben.
- ④ Von der Klasse `Zeichen` werden zwei Instanzen erzeugt und über den Konstruktor initialisiert.
- ⑤ Die beiden Instanzen werden mithilfe des überladenen Operators `+` verknüpft. Das Ergebnis wird ausgegeben.
- ⑥ Die Zahl 20 wird mit der Instanz `einZeichen` mithilfe des überladenen Operators `*` verknüpft. Das Ergebnis wird ebenfalls ausgegeben.

Beachten Sie, dass die Verknüpfung `einZeichen * 20` **nicht definiert** ist. Für diesen Anwendungsfall müsste der Operator `*` erneut überladen werden.

Sie können einem Operator durch Überladen eine völlig neue Bedeutung geben. Es ist aber empfehlenswert, sich an die übliche Bedeutung eines Operators zu halten und diese Bedeutung direkt oder im übertragenen Sinn auf die entsprechenden Klassen anzuwenden. Der Quellcode wird dadurch leichter verständlich.



11.8 Übungen

Übung 1: Überschreiben/abstrakte Klassen

Übungsdatei: Datumsformate.sln

Ergebnisdatei: Datumsformate2.sln

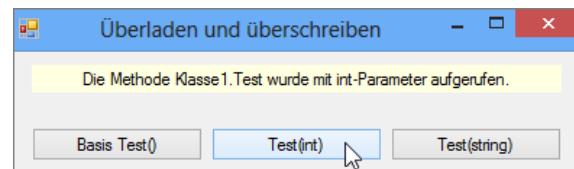
1. Erweitern Sie das in diesem Kapitel verwendete Beispiel `Datumsformate.sln` um eine weitere abgeleitete Klasse, die das aktuelle Datum mithilfe der Methode `ToOADate()` als Zahlenwert anzeigt.
2. Erzeugen Sie auch von dieser Klasse eine Instanz und rufen Sie die entsprechende Methode auf.

Übung 2: Überladen/Überschreiben

Übungsdatei: --

Ergebnisdatei: Overloading.sln

1. Erstellen Sie eine neue Windows-Anwendung.
2. Platzieren Sie im Formular drei Schaltflächen und eine Label-Komponente. Schalten Sie für die Label-Komponente die Eigenschaft `AutoSize` aus, wählen Sie eine andere Hintergrundfarbe (z. B. `Info`) und richten Sie den Label-Text innerhalb des Labels mit `MiddleCenter` aus. Entfernen Sie den Text `Label1`.
3. Implementieren Sie eine Basisklasse und eine davon abgeleitete Klasse in jeweils separaten Dateien.



4. Die Basisklasse erhält eine parameterlose überschreibbare Methode `Test()`, die unter gleichem Namen in der Basisklasse noch zweimal überladen wird und dabei einen `int`-Parameter bzw. einen `string`-Parameter übernimmt.
5. In der abgeleiteten Klasse wird die parameterlose Form der Methode überschrieben.
6. Alle Formen der Methode sollen eine Meldung über ihren Aufruf zurückgeben. Diese Meldung wird jeweils in dem Label angezeigt.
7. Erzeugen Sie in den Ereignismethoden (*Click*) der Schaltflächen eine Instanz der abgeleiteten Klasse und weisen Sie diese einer Objektvariablen vom Typ der Basisklasse zu. Weisen Sie jeder Schaltfläche eine entsprechende Form des Methodenaufrufs zu.

Übung 3: Operatoren überladen

Übungsdatei: --

Ergebnisdateien: Bruchzahlen.sln, BruchzahlenMitKuerzen.sln

1. Erzeugen Sie eine Konsolenanwendung.
2. Erstellen Sie in einer neuen Datei eine Klasse `Bruchzahl`, die für den Nenner und den Zähler entsprechende `int`-Felder mit den zugehörigen Eigenschaften enthält. (Nutzen Sie für die Programmierung der Felder und Eigenschaften den Code-Ausschnitt (Snippet) mit dem ShortCut Prop.)
3. Fügen Sie einen parameterlosen Konstruktor ein und erstellen Sie einen zweiten Konstruktor, der als Parameter Werte für den Zähler und den Nenner entgegennimmt.
4. Überschreiben Sie die Methode `ToString()`. Der String soll die Bruchzahl in der Schreibweise z/n enthalten, wobei z für den Zählerwert und n für den Nennerwert steht.
5. Definieren Sie für die Klasse `Bruchzahl` die Operatoren `+` und `*`, die als Ergebnis wiederum eine Bruchzahl ergeben. Die Berechnung soll mithilfe der nachfolgenden Formeln erfolgen. Beim Addieren wird zur Vereinfachung als gemeinsamer Hauptnenner das Produkt der beiden Nenner verwendet:

$$\frac{z_1}{n_1} + \frac{z_2}{n_2} = \frac{z_1 * n_2 + z_2 * n_1}{n_1 * n_2} \quad \text{und} \quad \frac{z_1}{n_1} * \frac{z_2}{n_2} = \frac{z_1 * z_2}{n_1 * n_2}$$

Um die Zahlenwerte klein zu halten, ist es sinnvoll, das kleinste gemeinsame Vielfache der beiden Nenner als Hauptnenner zu verwenden. Dies bleibt hier aber unberücksichtigt.

6. Deklarieren Sie in der Klasse `Program` innerhalb von `Main()` drei Bruchzahlen `bruchzahl1`, `bruchzahl2` und `ergebnis`. Weisen Sie `bruchzahl1` und `bruchzahl2` Instanzen zu, die Sie mit Beispielwerten initialisieren.
7. Wenden Sie die Operatoren `+` und `*` an. Speichern Sie das Ergebnis jeweils in der Bruchzahl `ergebnis` und geben Sie die Werte mithilfe der Methode `ToString()` aus.

$2/5 + 2/3 = 16/15$
$2/5 * 2/3 = 4/15$

Die Ausgabe des Programms



Als zusätzliche **Erweiterung** können Sie eine Methode implementieren, mit der Sie einen Bruch kürzen können (**BruchzahlenMitKuerzen.sln**). Dazu teilen Sie den Zählerwert und den Nennerwert durch ihren größten gemeinsamen Teiler. Eine Funktion, mit der Sie den größten gemeinsamen Teiler berechnen, wurde in Abschnitt 9.11 vorgestellt.

12 Schnittstellen (Interfaces)

12.1 Einführung zu Schnittstellen

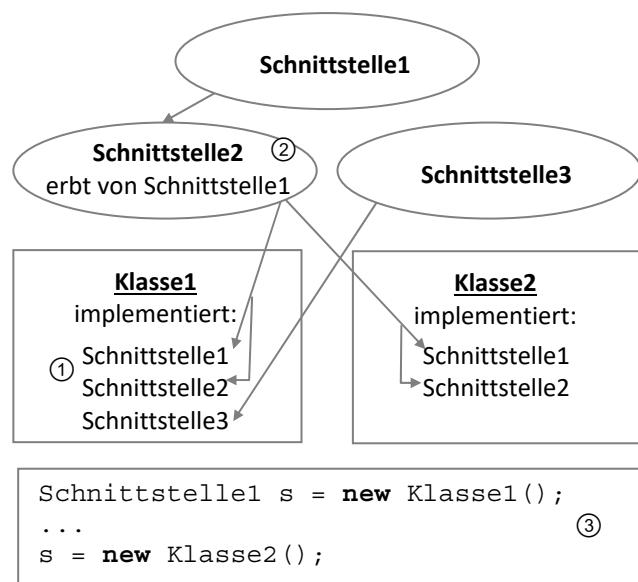
Schnittstellen (engl. Interfaces) definieren, ähnlich wie abstrakte Basisklassen, eine Grundfunktionalität. In der Regel ist das eine Gruppe von sachlich zusammengehörigen Eigenschaften, Methoden und Ereignissen.

- ✓ Schnittstellen enthalten im Gegensatz zu abstrakten Klassen **niemals** Implementierungen der einzelnen Member, sondern nur deren Deklaration.
- ✓ Klassen und auch Strukturen, die Sie im nachfolgenden Kapitel kennenlernen, können beliebig viele Schnittstellen implementieren ①.
- ✓ Eine Klasse erbt dadurch keine fertigen Member, sondern nur die Verpflichtung, alle in der Schnittstelle deklarierten Member zu implementieren. Ansonsten erhalten Sie eine abstrakte Klasse.
- ✓ Schnittstellen können von anderen Schnittstellen erben ②.
- ✓ Von Schnittstellen können Sie keine Instanzen bilden.
- ✓ Erst wenn eine Klasse alle Member einer Schnittstelle bzw. aller Schnittstellen implementiert, können Sie von dieser Klasse Instanzen bilden.

Durch Hinzufügen weiterer Schnittstellen lassen sich Klassen bzw. Klassenhierarchien ohne tiefe Eingriffe in ihre Struktur relativ leicht um bestimmte Funktionen erweitern.

Schnittstellen-Zuweisungskompatibilität

Durch die Verwendung von Schnittstellen können Sie über eine Variable vom Typ der Schnittstelle auf Objekte unterschiedlicher Klassen zugreifen ③. Voraussetzung dafür ist, dass diese Klassen die entsprechenden Schnittstellen implementiert haben. Über die Variable können Sie aber nur auf die in den Klassen implementierten Methoden der Schnittstelle zugreifen.



Abstrakte Klassen und Schnittstellen im Vergleich

Wenn Sie eine abstrakte Klasse definieren, die nur Methodendeklarationen besitzt, können Sie stattdessen auch eine Schnittstelle (Interface) einsetzen. Die Verwendung von Interfaces hat den Vorteil, dass Sie zusätzlich zur Implementierung eines Interfaces die Klasse noch von einer anderen Klasse ableiten können. Mit abstrakten Klassen ist dies nicht möglich, da in C# nur Einfachvererbung möglich ist. Außerdem kann eine Klasse mehrere Interfaces implementieren. Abstrakte Klassen haben jedoch den Vorteil, dass Sie hier auch Felder und vollständig implementierte Methoden und Eigenschaften einfügen können. Dies ist bei Interfaces nicht möglich.

12.2 Schnittstellen deklarieren

Bevor eine Schnittstelle in einer Klasse implementiert werden kann, muss sie deklariert werden. Die Deklaration einer Schnittstelle ist einer Klassendeklaration ähnlich. Schnittstellen können auch vererbt werden. Eine Schnittstelle erbt von ihrer Vorfahrschnittstelle die Deklaration aller Schnittstellenmember.

Syntax zur Deklaration einer Schnittstelle

```
[public|internal] interface IName [: Interface1[, Interface2, ...]] ①
{
    Classmember1
    Classmember2
    ...
}
```

- ✓ Für die Deklaration einer Schnittstelle wird das Schlüsselwort `interface` verwendet.
- ✓ Sie können eine Schnittstellendeklaration mit einem Zugriffsmodifizierer, z. B. `public`, versehen.
- ✓ Zur Kennzeichnung als Schnittstelle beginnt der Bezeichner einer Schnittstelle mit einem großen I, auf das ein Name folgt, der die Schnittstelle benennt, z. B. `IMeineSchnittstelle`.
- ✓ Eine Schnittstelle kann von Schnittstellen erben, die nach einem Doppelpunkt durch Kommata getrennt angegeben werden ①.
- ✓ Schnittstellen können Eigenschaften, Methoden und Ereignisse enthalten. Diese werden aber nicht vollständig implementiert, sondern wie abstrakte Methoden nur als Deklaration (Signatur) in dem Interface aufgeführt. Die explizite Angabe von `abstract` ist dabei unzulässig.
- ✓ Alle Member einer Schnittstelle sind automatisch als `public` deklariert. Die Angabe von Sichtbarkeitsattributen, auch `public`, ist unzulässig.
- ✓ Schnittstellen besitzen keine Konstruktoren oder Destruktoren. Von Schnittstellen können keine Instanzen erzeugt werden.

Eine Schnittstellendatei erstellen

- ▶ Öffnen Sie das Projekt, in das Sie die Schnittstelle einfügen möchten.
- oder Erzeugen Sie ein neues Projekt.
- ▶ Rufen Sie den Menüpunkt *Projekt - Neues Element hinzufügen* auf.
- ▶ Wählen Sie die Vorlage *Schnittstelle* und vergeben Sie einen Namen für das Interface, z. B. `IMeineSchnittstelle`.

In das Projekt wird eine neue eigenständige Datei mit dem Programmgerüst für ein Interface eingefügt.

12.3 Schnittstellen implementieren

Eine Schnittstelle kann nach der Deklaration in Klassen oder Strukturen implementiert werden. Um Instanzen dieser Klasse bilden zu können, müssen alle Member der Schnittstelle implementiert werden.

Syntax zur Implementierung von Schnittstellen

- ✓ Die zu implementierenden Schnittstellen werden nach einem Doppelpunkt mit Kommas getrennt aufgeführt.
- ✓ Falls die Klasse außerdem von einer anderen Klasse erbt, so muss der Name dieser Basisklasse vor den Schnittstellen angegeben werden.

Syntax zur einfachen Implementierung der Schnittstellenmember

```
class Classname : [BaseClass,] Interface1[, Interface2, ...]
{
    ...
    public InterfaceMember1 // z.B. aus Interface1
    { ... }
    public InterfaceMember2 // z. B. entweder aus Interface1
    { ... }                // oder aus Interface2
}
```

- ✓ Bei der Implementierung deklarieren Sie den Interface-Member mit der Signatur, die in der Schnittstelle angegeben ist, und dem Modifizierer `public`.
- ✓ Die Implementierung von Schnittstellenmembern kann entfallen, wenn die Implementierung von einer anderen Klasse geerbt wurde.
- ✓ Die Klassendeklaration kann noch weitere (nicht mit der Implementierung von Schnittstellen zusammenhängende) Member enthalten.

Syntax zur expliziten Implementierung der Schnittstellenmember

Durch die zusätzliche Angabe des Schnittstellennamens werden die Schnittstellenmember explizit implementiert.

```
class Classname : [BaseClass,] Interface1[, Interface2, ...]
{
    ...
    Interface1.InterfaceMember1 // z.B. aus Interface1
    { ... }
    Interface1.InterfaceMember2 // z.B. aus Interface1 ①
    { ... }
    Interface2.InterfaceMember2 // z.B. aus Interface2 ②
    { ... }
}
```

- ✓ Bei der **expliziten** Implementierung geben Sie vor dem Membernamen – abgetrennt durch einen Punkt – den Namen der Schnittstelle an, die die Implementierung erfordert.
- ✓ Bei der expliziten Implementierung werden auch die Member unterschieden, die mit gleichlautender Signatur (Name und Parameterliste) in mehreren Interfaces vorhanden sind (① und ②).
- ✓ Möglicherweise werden in mehreren Interfaces Member mit gleichlautender Signatur verwendet, die unterschiedliche Funktionen haben sollen. In diesem Fall darf höchstens einer dieser Member nicht explizit implementiert werden.

Bei der **expliziten** Implementierung werden die Member in der abgeleiteten Klasse **verborgen**. Es dürfen daher auch keine Zugriffsmodifizierer angegeben werden. Die Schnittstellenmember können nur über eine Instanz der Schnittstelle angesprochen werden.

Unterstützung bei der Eingabe des Programmcodes

- Deklarieren Sie die Klasse und geben Sie dahinter eine Schnittstelle an (`class ClassName : IZugriff`).

Unter dem Schnittstellennamen erscheint ein Smarttag-Anzeiger (eine Lampe) mit einer Fehlerbeschreibung und dem Angebot für mögliche Korrekturen.



Schnittstellenmember implementieren

- Öffnen Sie in dem Smarttag die möglichen Korrekturen.
- Wählen Sie den Befehl *I Zugriff-Schnittstelle implementieren*, um die in der Schnittstelle definierten Member zu implementieren.
oder Wählen Sie den Befehl *I Zugriff-Schnittstelle explizit implementieren*, um die in der Schnittstelle definierten Member explizit zu implementieren.

Automatisch wird das Grundgerüst für alle zu implementierenden Member eingefügt.

Über das Kontextmenü des Schnittstellennamens stehen Ihnen diese Befehle im Untermenü des Befehls *Schnittstelle implementieren* ebenfalls zur Verfügung.

Kennzeichnung als Codeblock

Der automatisch erzeugte Programmcode wird in einen Codeblock `#region ... #endregion` eingeschlossen. Dieser so gekennzeichnete Block kann im Editor mit den Symbolen aus und wieder eingeblendet werden. Diese Block-kennzeichnung bleibt vom Compiler unberücksichtigt.

```
#region Comment
...
#endregion
```

Beispiel: Personen.sln

Das folgende Beispiel zeigt, wie eine Schnittstelle deklariert, in einer Klasse implementiert, vererbt und für den Zugriff auf Klasseninstanzen genutzt wird. In Ihren Projekten sollten Sie Schnittstellen und Klassen jeweils in eigenen Dateien unterbringen.

①	<code>interface IZugriff</code>
	<code>{</code>
	<code> string Zugriff { get; set; }</code>
	<code> void Ausgabe();</code>
	<code>}</code>
②	<code>class Person : IZugriff</code>
	<code>{</code>
③	<code> private string name;</code>
	<code> string IZugriff.Zugriff</code>
	<code> {</code>
④	<code> get { return name; }</code>

```

⑤     set { name = value; }
}
⑥ public void Ausgabe()
{
    Console.WriteLine(this.name);
}
}

⑦ class Test : Person
{ }

class Program
{
    static void Main(string[] args)
    {
        IZugriff einePerson = new Person();
        einePerson.Zugriff = "Hans Mustermann";
        einePerson.Ausgabe();

        einePerson = new Test();
        einePerson.Zugriff = "Willi Muster";
        einePerson.Ausgabe();

        Person nochEinePerson = new Person();
        // nochEinePerson.Zugriff = "Klaus Meier"; nicht zulässig!! Daher Kommentar
        ((IZugriff)nochEinePerson).Zugriff= "Klaus Meier"; // über cast zulässig
        nochEinePerson.Ausgabe();                         // zulässig!!
    }
}

```

- ① Die Schnittstelle `IZugriff` wird deklariert. Sie enthält die Eigenschaft `Zugriff`, die einen Schreib-/Lesezugriff auf ein Feld vom Typ `string` ermöglichen soll, und eine Methode `Ausgabe()`.
- ② Die Klasse `Person` dient zum Speichern des Namens einer Person. Die Klassendeklaration enthält die Angabe über die zu implementierende Schnittstelle.
- ③ An dieser Stelle wird die von der Schnittstelle geforderte Eigenschaft `Zugriff` explizit implementiert.
- ④-⑤ Die Eigenschaft regelt den Schreib-/Lesezugriff auf das Feld `name`.
- ⑥ Die Methode `Ausgabe()` wird hier zur Demonstration nicht explizit implementiert.
- ⑦ Die von der Klasse `Person` abgeleitete Klasse `Test` erbt die gesamte Schnittstellenimplementierung.
- ⑧ Eine Objektvariable vom Typ der Schnittstelle `IZugriff` wird deklariert. Ihr wird eine Instanz der Basis-Klasse `Person` zugewiesen. Über die implementierte Eigenschaft `Zugriff` wird das Feld `name` gefüllt. Anschließend erfolgt die Ausgabe.
- ⑨ Durch die Zuweisungskompatibilität ist es möglich, der Objektvariablen vom Typ der Schnittstelle eine Instanz der abgeleiteten Klasse `Test` zuzuweisen. Die abgeleitete Klasse hat die Implementierung der Schnittstelle von der Klasse `Person` geerbt. Deshalb kann mithilfe dieser Instanz ebenfalls der Zugriff auf die Eigenschaft `Zugriff` durchgeführt werden.
- ⑩ Ein Objekt der Klasse `Person` wird erzeugt.
- ⑪ Für dieses Objekt kann die Eigenschaft `Zugriff` nicht angesprochen werden, da `Zugriff` explizit implementiert wurde und somit nur über die Schnittstelle verwendet werden kann.

Hans Mustermann
Willi Muster
Klaus Meier

Die Ausgabe des Programms

- ⑫ Durch einen expliziten Cast der Variablen in den Typ der Schnittstelle können auch die verborgenen Schnittstellenmethoden aufgerufen werden.
- ⑬ Die Methode `Ausgabe()` wurde nicht explizit implementiert und kann daher auch über die Instanz der Klasse angesprochen werden.

12.4 Member einer Schnittstelle verdecken

Sie können die Implementierung eines Schnittstellelements innerhalb einer abgeleiteten Klasse durch eine erneute Deklaration verdecken bzw. überschreiben. Ein explizites Auszeichnen des Überschreibens der Implementierung mithilfe des Schlüsselworts `override` ist jedoch bei Methoden unzulässig.

Beispiel: *Personen2.sln*

Die Eigenschaft `Zugriff` des letzten Beispiels wird hier in der abgeleiteten Klasse verdeckt. Die neue Implementierung speichert den übergebenen Namen in Großbuchstaben.

```
interface IZugriff
{
    string Zugriff { get; set; }
    void Ausgabe();
}

class Person : IZugriff
{
    protected string name;
    string IZugriff.Zugriff
    {
        get { return name; }
        set { name = value; }
    }
    public void Ausgabe()
    {
        Console.WriteLine(this.name);
    }
}

class Test : Person, IZugriff
{
    string IZugriff.Zugriff
    {
        get { return name; }
        set { name = value.ToUpper(); }
    }
}
```

```

④ new public void Ausgabe()
{
    Console.WriteLine("Test: " + this.name);
}

class Program
{
    static void Main(string[] args)
    {
        IZugriff einePerson = new Person();
        einePerson.Zugriff = "Hans Mustermann";
        einePerson.Ausgabe();
        einePerson = new Test();
        einePerson.Zugriff = "Willi Muster";
        einePerson.Ausgabe();
    }
}
...

```

- ① Die Eigenschaft Zugriff wird in der Basisklasse explizit implementiert.
- ② Auch die abgeleitete Klasse implementiert das Interface IZugriff.
- ③ Die Eigenschaft Zugriff wird neu implementiert und verdeckt somit die Implementierung der Basisklasse. Der set-Teil der Eigenschaftsdeklaration wandelt den übergebenen String mithilfe der Methode ToUpper() in Großbuchstaben um und speichert das Ergebnis im Feld name.
- ④ Da die Methode Ausgabe() in der Basisklasse nicht explizit implementiert wurde, kann sie in der abgeleiteten Klasse neu deklariert werden und verbirgt somit die Methode der Basisklasse. Um zu kennzeichnen, dass dieses Verbergen bewusst geschieht, stellen Sie der Deklaration das Schlüsselwort new voran.

Hans Mustermann
Test: WILLI MUSTER

Die Ausgabe des Programms

12.5 Typprüfung und -konvertierung

Wie bei Klassen können Sie mithilfe der Operatoren `as` und `is` Typabfragen und -umwandlungen durchführen.

Beispiel: *Inter.sln*

Das folgende Beispiel zeigt anhand einer Schnittstelle `ITest` und einer implementierenden Klasse `Test` einige Zuweisungs- und Konvertierungsmöglichkeiten in Verbindung mit Schnittstellen.

```

① interface ITest
{
    void Ausgabe();
}

② class Test : ITest
{
}
```

```

③  void ITest.Ausgabe()
{
    Console.WriteLine("Die Methode \"Ausgabe()\" wurde aufgerufen.");
}
④  class Program
{
    static void Main(string[] args)
    {
        ⑤     ITest instanz1, instanz2;
        ⑥     Test instanz3 = new Test();
        ⑦     instanz1 = instanz3;
        ⑧     if (instanz1 is ITest)
            Console.WriteLine("Mit der Schnittstelle ITest kompatible Instanz");
        ⑨     instanz2 = instanz3 as ITest;
        ⑩     if (instanz2 != null)
            instanz2.Ausgabe();
        ((ITest)instanz3).Ausgabe();
    }
}

```

- ① Die Schnittstelle `ITest` mit ihrer Methode `Ausgabe()` wird deklariert.
- ② Hier wird die Klasse `Test` vereinbart. Sie implementiert die Schnittstelle `ITest`.
- ③ Wie in der Schnittstelle gefordert, wird die Methode `Ausgabe()` implementiert.
- ④ Die beiden Variablen `instanz1` und `instanz2` vom Schnittstellentyp `ITest` werden deklariert.
- ⑤ Eine Instanz der Klasse `Test` wird erzeugt und der Objektvariablen `instanz3` zugewiesen.
- ⑥ Durch die Zuweisungskompatibilität ist es möglich, der Objektvariablen `instanz1` vom Typ `ITest` ein Objekt der Klasse `Test` zuzuweisen.
- ⑦ Die Instanz `instanz1` wird einer Typprüfung unterzogen. Liefert der Ausdruck den Wert `true`, wird eine entsprechende Meldung ausgegeben.
- ⑧ Hier erfolgt eine Konvertierung des Objekts `instanz3` vom Typ `Test` in ein Schnittstellenobjekt vom Typ `ITest`, das der Variablen `instanz2` vom Typ `ITest` zugewiesen wird.
- ⑨ War die Konvertierung und Zuweisung erfolgreich, d. h., es existiert ein Objekt `instanz2`, dann wird die Ausgabemethode dieses Objekts aufgerufen.
- ⑩ Abschließend wird die Objektvariable `instanz3` vom Typ `Test` explizit in ein Schnittstellenobjekt vom Typ `ITest` konvertiert. Die konvertierte Instanz wird als Qualifizierer für den Aufruf der Methode `Ausgabe()` verwendet.

Mit der Schnittstelle ITest kompatible Instanz
 Die Methode "Ausgabe()" wurde aufgerufen.
 Die Methode "Ausgabe()" wurde aufgerufen.

Die Ausgabe des Programms

12.6 Übungen

Übung 1: Medien

Übungsdatei: --

Ergebnisdatei: Media.sln

1. Erzeugen Sie eine Konsolenanwendung *Media.sln*.
2. Erstellen Sie in dem Projekt als eigenständige Datei ein Interface **IMedia** mit drei Methoden:
 - ✓ string *DisplayMedia()*;
 - ✓ string *PlayMedia()*;
 - ✓ string *StopMedia()*;
3. Erstellen Sie ebenfalls als eigenständige Dateien drei Klassen **Picture**, **Video**, **Audio**, die das Interface implementieren.
4. Implementieren Sie die Methoden. Alle Methoden geben den Namen der Klasse und anschließend einen Text als String zurück: Beispielsweise gibt die Methode *PlayMedia()* der Klasse **Video** den Namen der Klasse und den Text *Video gestartet* als String zurück (vgl. Abbildung).
5. Den Namen der Klasse bestimmen Sie, indem Sie mit der Methode *GetType()* die Klasse ermitteln und von dieser Klasse das Feld **Name** auslesen: *GetType().Name*
6. Nutzen Sie in der Klasse **Program** die drei Klassen.

Medium1:
Picture: Bild wird angezeigt.
Picture: PlayMedia() wird nicht unterstützt.
Picture: Bild ausgeblendet
Medium2:
Video: Videobild wird angezeigt.
Video: Video gestartet
Video: Video gestoppt
Medium3:
Audio: DisplayMedia() wird nicht unterstützt.
Audio: Audio gestartet
Audio: Audio gestoppt

Die Ausgabe des Programms

Übung 2: Einfache Kontoführung

Übungsdatei: --

Ergebnisdatei: Bankkonto.sln

1. Erstellen Sie eine Konsolenanwendung, um ein Programm für eine einfache Kontoführung zu schreiben.
2. Entwickeln Sie in einer separaten Datei eine Klasse **Konto**. Als einziges Feld soll diese Klasse eine als **private** deklarierte Variable für den Kontostand besitzen.
3. Deklarieren Sie in einer weiteren separaten Datei eine Schnittstelle, in der eine Eigenschaft zum Schreib-/Lesezugriff auf den Kontostand deklariert ist.
4. Importieren Sie die Schnittstelle durch die Klasse **Konto** und implementieren Sie dort die von der Schnittstelle deklarierte Eigenschaft.
5. Implementieren Sie einen Konstruktor, über den Sie den Kontostand mit einem Wert von 5000 initialisieren.
6. Das Programm sollte so aufgebaut sein, dass sowohl zu Beginn als auch nach jeder Ein- bzw. Auszahlung der aktuelle Kontostand angezeigt wird. Durch die Eingabe einer Zahl 1, 2 oder 0 soll der Anwender die Möglichkeit haben, eine Einzahlung (1) oder eine Auszahlung (2) vorzunehmen oder das Programm mit der Eingabe der Zahl 0 zu beenden. (Tipp: switch-case-Anweisung)

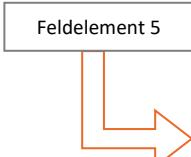
13 Komplexe Datentypen

13.1 Eindimensionale Arrays

Was sind Arrays?

Über **Arrays** (auch Datenfelder oder kurz Felder genannt) können Sie mehrere Daten eines Typs in einer einzigen Struktur speichern. Über deren **Index** besitzen Sie einen wahlfreien Zugriff auf die einzelnen Daten, die sogenannten **Array-Elemente (Feldelemente)**.

- ✓ Die Abbildung zeigt ein **eindimensionales Array**. Alle Elemente sind unabhängig voneinander und können über einen Index erreicht werden. Ein eindimensionales Array wird auch als Vektor bezeichnet.
- ✓ Der **Index** ist ein ganzzahliger Wert, der jeweils zu einem Feldelement gehört. Das erste Feldelement besitzt den Index **0**, für jedes weitere Element erhöht sich der Index jeweils um 1.
- ✓ Die Größe kann nachträglich nicht mehr geändert werden.



0	122
1	56
2	145
3	157
4	87
5	57
6	145
7	155
8	68
9	23

Ein eindimensionales Array mit zehn Elementen vom Typ `int`

Syntax für die Deklaration und Erzeugung eines eindimensionalen Arrays

- ✓ Sie deklarieren eine Array-Variable, indem Sie an den Datentyp ein Klammernpaar **()** anfügen ①. Beachten Sie, dass die eckigen Klammern hier eingegeben werden müssen und keine optionalen Eingaben darstellen.
- ✓ Um eine Array-Instanz zu erzeugen, verwenden Sie das Schlüsselwort `new`. Hinter dem Datentyp geben Sie in eckigen Klammern **()** die Anzahl der Elemente an, die in dem Array gespeichert werden sollen ②. Sie können die Arraygröße nachträglich nicht mehr ändern.
- ✓ Die Deklaration und die Erzeugung einer Instanz können Sie in einer Anweisung durchführen ③.
- ✓ Bei der Erzeugung werden die Array-Elemente automatisch mit Standardwerten initialisiert. So erhalten z. B. Array-Elemente vom Typ `int` den Wert 0.

```
type[] identifier; ①
identifier = new type[size]; ②
type[] identifier1 = new type[size]; ③
```

Tipps zur Benennung



Die Benennung erfolgt nach den Regeln für Bezeichner, jedoch sollten Sie jeweils den Plural (Mehrzahl) verwenden, um deutlich zu machen, dass es sich um mehrere Elemente handelt.

Heißt der Datentyp z. B. `Person`, dann könnten Sie das Array dieses Datentyps mit `Personen` benennen.

Array-Literale: Feldelemente individuell initialisieren

```
type[] identifier1 = new type[size] {value0, value1, ..., valueN}; ①
type[] identifier2 = new type[] {value0, value1, ..., valueN}; ②
type[] identifier3 = {value0, value1, ..., valueN}; ③
```

- ✓ Bereits bei der Erzeugung des Arrays können Sie die Feldelemente mit Daten initialisieren.
- ✓ Geben Sie nach der Erzeugung des Arrays mit new in geschweiften Klammern `{ }` die gewünschten Werte als Literale ein. Trennen Sie dabei die einzelnen Werte durch Kommata.
- ✓ Wenn die Größe angegeben wird ①, dann muss diese exakt mit der Anzahl der angegebenen Werte übereinstimmen.
- ✓ Wenn Sie die Größe nicht angeben ②, wird diese aus der Anzahl der angegebenen Werte ermittelt.
- ✓ Die Initialisierung mit new kann entfallen.

Über den Index auf Array-Elemente zugreifen

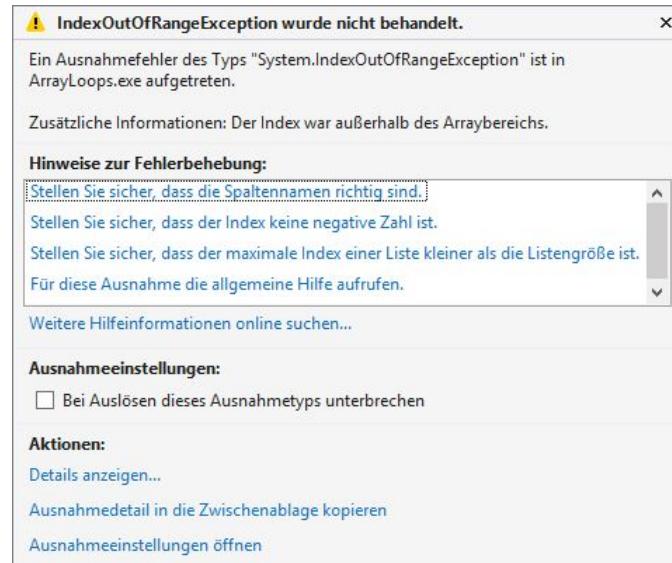
Wie Variablen einfacher Datentypen können Sie auch Array-Elementen Werte zuweisen und die Inhalte auslesen.

- ✓ Der Zugriff auf Feldelemente erfolgt durch den Index, der bei 0 beginnt und entsprechend für das letzte Feldelement den Wert Feldlänge - 1 besitzt.
- ✓ Der Index kann ein beliebiger Ausdruck sein, der ein Ergebnis von ganzzahligem Typ liefert.
- ✓ Der Indexwert wird in eckige Klammern `[]` gesetzt, die üblicherweise ohne Leerzeichen direkt hinter den Bezeichner geschrieben werden.
- ✓ Sie können einem über den Index selektierten Array-Element einen Wert zuweisen ①.
- ✓ Über den Index (`intExpression`) können Sie den Wert des entsprechenden Array-Elements auslesen ② und beispielsweise in einer anderen Variablen speichern.

```
identifier[intExpression] = expression; ①
identifier1 = identifier[intExpression]; ②
```

Fehlerhafte Indexangaben

Der Wertebereich für den Index erstreckt sich von 0 für das erste Element bis zu Feldlänge - 1 für das letzte Element. Wenn Sie einen ungültigen Index (außerhalb dieses Bereichs) angeben, wird dieser Fehler bei der Ausführung erkannt und es wird eine sogenannte Exception `IndexOutOfRangeException` erzeugt (vgl. Kap. 14). Ein entsprechender Hinweis in der Konsole wird ausgegeben. Bei der Verwendung des Debuggers erscheint das nebenstehende Fenster.



Die Anzahl der Array-Elemente ermitteln

Jedes Array besitzt die Eigenschaft `Length`, mit der Sie die Größe des Arrays, d. h. die **Anzahl** der Array-Elemente, bestimmen können.

```
int identifier = arrayIdentifier.Length;
```

Beispiele für die Arbeit mit Arrays: *UseArray.sln*

```
class Program
{
    static void Main(string[] args)
    {
        int a = 2;
        int b = 6;

①     int[] field = { 3, 4, 5, a + b };
②     field[2] = 7;
③     Console.WriteLine("Das Feld enthält {0} Elemente: ", field.Length);
        for (int i = 0; i < field.Length; i++)
        {
④         Console.WriteLine("Element {0}: {1}", i, field[i]);
        }
    }
}
```

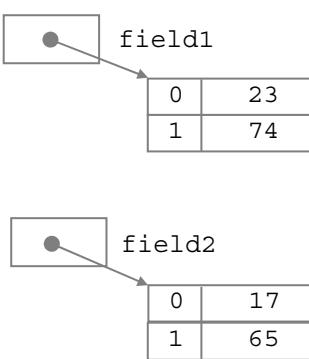
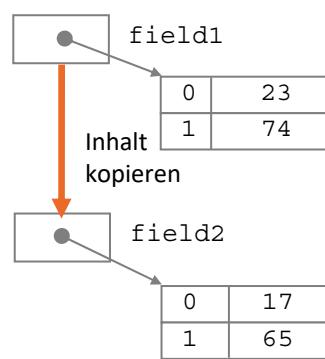
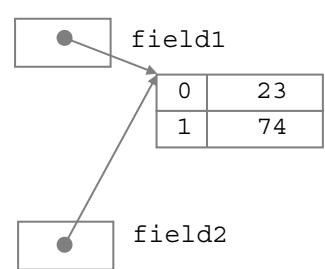
- ① Definition, Erzeugung und Initialisierung eines Arrays mit individuellen Werten
- ② Wertzuweisung an das dritte Feldelement
- ③ Die Anzahl der Feldelemente (entspricht Größe des Arrays) wird mit der Eigenschaft `field.Length` ermittelt und ausgegeben.
- ④ Für jedes Array-Element werden der Index und der Wert ausgegeben.

Das Feld enthält 4 Elemente:
Element 0: 3
Element 1: 4
Element 2: 7
Element 3: 8

Die Ausgabe des Programms

Zuweisungen bei Array-Variablen

Array-Variablen sind Referenzvariablen. Daher können Sie ein Array nicht kopieren, indem Sie einer Array-Variablen eine andere Array-Variable zuweisen. Bei dieser Zuweisung wird lediglich die Referenz kopiert. Beide Array-Variablen verweisen anschließend auf dasselbe Array. Das zweite Array wird nicht mehr referenziert und somit freigegeben.

Ausgangszustand	Zuweisung	Ergebnis
		
<pre>int[] field1 = {23, 74}; int[] field2 = {17, 65};</pre>	<pre>field2 = field1;</pre>	<p>Beide Array-Variablen verweisen auf dasselbe Array.</p>

Arrays in Schleifen bearbeiten

Häufig werden Arrays innerhalb von Schleifen bearbeitet. Beispielsweise sollen alle Elemente eines Arrays auf einen bestimmten Wert gesetzt werden ① oder Sie möchten alle Werte eines Arrays ausgeben ②.

```
int[] squareNumbers = new int[10];
for (int i = 0; i < squareNumbers.Length; i++)
{
    squareNumbers[i] = i * i;      ①
}
for (int i = 0; i < squareNumbers.Length; i++)
{
    Console.WriteLine(squareNumbers[i]);  ②
}
```

Beispiel: „ArrayLoops.sln“

Die **foreach**-Schleife

Die Schreibweise für das Auslesen von Arrays in Schleifen können Sie abkürzen und als sogenannte **foreach-Schleife** schreiben, wenn ...

- ✓ Array-Elemente nur ausgelesen und nicht verändert werden,
- ✓ beginnend beim ersten Element alle Elemente nacheinander durchlaufen werden sollen,
- ✓ nur **ein** Array durchlaufen wird.

Syntax der **foreach**-Schleife

- ✓ Nach dem Schlüsselwort **foreach** definieren Sie im Schleifenkopf eine Variable (**element**) vom Typ eines Array-Elements.
- ✓ Nach dem Schlüsselwort **in** geben Sie den Namen des Arrays an, das durchlaufen werden soll.
- ✓ Auch diese Art der Schleife lässt sich mit **break** vorzeitig abbrechen.

```
foreach (type element in Field)
{
    ...
    [break;]
    ...
}
```

Für die schnelle Erstellung einer **foreach**-Schleife steht Ihnen in Visual Studio der Code-Ausschnitt **foreach** zur Verfügung.

Beispiel

In jedem Schleifendurchlauf wird eine Variable (**sq**) vom Typ der Array-Elemente definiert (hier: **int**) ①. Dieser Variablen wird bei jedem Durchlauf der Wert des entsprechenden Array-Elements zugewiesen. Sie können diese Variable direkt verwenden und beispielsweise in einer Ausgabeanweisung ausgeben ②.

```
...
foreach (int sq in Square)      ①
{
    Console.WriteLine(sq);       ②
}
...
```

Beispiel: „ArrayLoops.sln“

Die Verwendung der **foreach**-Schleife bietet sich an, wenn alle vorhandenen Elemente nacheinander angeprochen werden sollen. Zur Adressierung ausgewählter Elemente eignet sie sich weniger, da kein Index zur Verfügung steht.



13.2 Mehrdimensionale und verzweigte Arrays

Mehrdimensionale Arrays

Ein Array kann mehrdimensional sein. Ein zweidimensionales Array wird auch als Matrix bezeichnet und lässt sich in einer Tabelle veranschaulichen. Beachten Sie, dass mit jeder Dimension der Speicherbedarf erheblich zunimmt.

Sie möchten beispielsweise für drei Messstationen (Index 0 bis 2) die monatlichen Durchschnittstemperaturen (Index 0 bis 11) eines Jahres speichern. Die Temperatur einer Messstation ist zum einen abhängig vom Monat. Zum anderen ist die Durchschnittstemperatur in einem Monat abhängig vom Standort (Messstation).

	0	1	2	3	4	5	6	7	8	9	10	11
0	1	0	4	8	12	16	18	19	15	8	5	1
1	2	1	10	12	17	24	23	26	17	9	9	6
2	1	1	8	10	14	20	21	22	15	7	8	4

Ein zweidimensionales Array mit insgesamt 36 Elementen
(3 * 12) vom Typ int

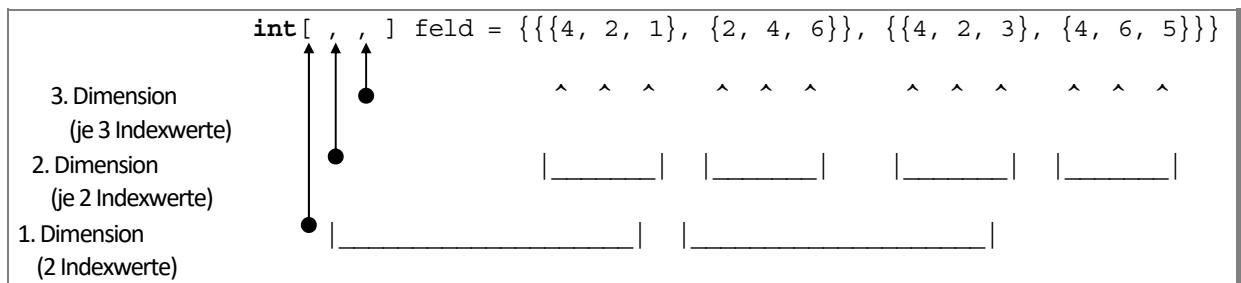
Syntax für die Deklaration und Erzeugung mehrdimensionaler Arrays

```
type [, , ...] identifier; ①
identifier = new type[size, size1, ...]; ②
type [, , ...] identifier = new type[size, size1, ...]; ③
```

- ✓ Bei der Deklaration eines mehrdimensionalen Arrays trennen Sie innerhalb der Klammern ① ② die Dimensionen durch Kommata ①.
- ✓ Bei der Initialisierung geben Sie in dem Klammerpaar ① ② für jede Dimension die Anzahl der Elemente an ②.
- ✓ Deklaration und Initialisierung können in einer Anweisung erfolgen ③.

Array-Literale: mehrdimensionale Arrays initialisieren

- ✓ Bei mehrdimensionalen Arrays schreiben Sie innerhalb der geschweiften Klammern für alle Indexwerte der ersten Dimension weitere Paare geschweifter Klammern ① ② und trennen diese durch Kommata.
- ✓ Sofern das Array mehr als zwei Dimensionen besitzt, werden innerhalb dieser geschweiften Klammern für jeden Indexwert der zweiten Dimension wiederum Paare geschweifter Klammern gesetzt und durch Kommata getrennt.
- ✓ Diese Schachtelung wird fortgesetzt. Lediglich für die letzte Dimension schreiben Sie entsprechend den Indexwerten, durch Kommata getrennt, die Arraywerte.



Verzweigte (unregelmäßige) Arrays

Arrays, die als Elemente wiederum Arrays enthalten, werden als **verzweigte Arrays** (engl. jagged arrays) bezeichnet. Da die enthaltenen Arrays unterschiedlich groß sein können, werden sie häufig auch **unregelmäßige Arrays** genannt.

```
type[][] identifier;
type[][] identifier = new type[size][];
```

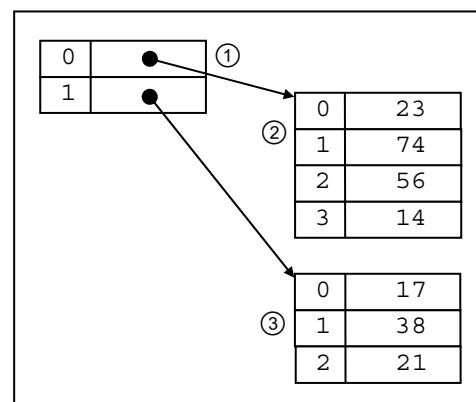
- ✓ Bei verzweigten Arrays schreiben Sie für jede Dimension ein eigenes Klammerpaar $\square \square$.
- ✓ Nur für die erste Dimension kann bei der Erzeugung der Instanz mit `new` die Größe angegeben werden.
- ✓ In den Feldern des "äußeren" Arrays werden nur Referenzen auf die enthaltenen Arrays gespeichert.
- ✓ Für diese Arrays muss zunächst über das Schlüsselwort `new` Speicherplatz reserviert werden.

Das nachfolgende Beispiel zeigt die Definition eines zweidimensionalen verzweigten (unregelmäßigen) Arrays.

```
(1) int[][] field = new int[2][];
(2) field[0] = new int[] { 23, 74, 56, 14 };
(3) field[1] = new int[] { 17, 38, 21 };
```

Beispiel: „VerzweigtesArray.sln“

- ① Ein Array wird erzeugt, in dem zwei Arrays mit `int`-Werten gespeichert werden sollen.
- ② Das erste enthaltene Array wird erzeugt. Das Array soll die Werte 23, 74, 56 und 14 enthalten. Über das Schlüsselwort `new` wird Speicherplatz für das Array mit 4 Elementen vom Typ `int` reserviert und dieses mit den vier Werten initialisiert.
- ③ Das zweite enthaltene Array besteht nur aus drei Elementen: Ein verzweigtes Array
17, 38 und 21.



13.3 Mit Arrays arbeiten

Funktionen, Methoden und Eigenschaften im Zusammenhang mit Feldern: *ArrayMethods.sln*

Die folgende Tabelle zeigt eine Auswahl von Methoden und Eigenschaften für die Arbeit mit Arrays. Als Ausgangsbasis für die in der Tabelle angeführten Beispiele werden zwei Array-Variablen definiert:

```
int i;
int[] ar = new int[24];
int[] ar2 = new int[30];
```

Methode/Eigenschaft	Beschreibung und Beispiel
Rank	Die Eigenschaft Rank enthält die Anzahl der Dimensionen des Feldes. <code>i = ar.Rank;</code> <code>i</code> erhält den Wert 1
GetUpperBound()	Die Methode GetUpperBound() liefert den Index des höchsten Elements einer Dimension. <code>i = ar.GetUpperBound(0);</code> <code>i</code> erhält den Wert 23
GetLowerBound()	Die Methode GetLowerBound() liefert den Index des niedrigsten Elements einer Dimension. <code>i = ar.GetLowerBound(0);</code> <code>i</code> erhält den Wert 0

Methode/Eigenschaft	Beschreibung und Beispiel
GetLength()	Die Methode GetLength() liefert die Länge des Feldes für die als Parameter angegebene Dimension. i = ar.GetLength(0); i erhält den Wert 24
Length	Die Eigenschaft Length gibt die Anzahl der Elemente aller Dimensionen an. i = ar.Length; i erhält den Wert 24
Clear()	Diese Methode setzt die Feldelemente auf 0 bzw. null. Das Feld bleibt jedoch bestehen. System.Array.Clear(ar, 0, 24);
CopyTo()	Kopiert ein Array ar ab der angegebenen Position (zweiter Parameter, hier: 4) in ein Array ar2. ar.CopyTo(ar2, 4);
Reverse()	Kehrt die Reihenfolge der Elemente oder eines Teils der Elemente in einem eindimensionalen Feld um System.Array.Reverse(ar);
SetValue()	Mit dieser Methode kann einem Feldelement (hier: dem dritten Element) ein bestimmter Wert (hier: 123) zugewiesen werden. ar.SetValue(123, 2);
Sort()	Ein Feld wird aufsteigend sortiert. System.Array.Sort(ar);
BinarySearch()	Diese Methode sucht einen Wert in einem sortierten Feld und gibt den Index zurück. i = System.Array.BinarySearch(ar, 123);

13.4 Parameter-Arrays

Was sind Parameter-Arrays?

Mit einem sogenannten Parameter-Array (**params**) können Sie einer Methode beliebig viele Parameter **eines Typs** übergeben. Da auch der Typ **object** verwendet werden kann, sind auch "gemischte" Parameterlisten möglich. Ein Parameter-Array stellt eine spezielle Parameterart dar.

- ✓ Innerhalb der Parameterliste einer Methode kann nur **ein** Parameter-Array verwendet werden.
- ✓ Wenn einer Methode mehrere Parameter übergeben werden, muss das Parameter-Array als letzter Parameter aufgeführt werden.

Syntax für die Verwendung eines Parameter-Arrays

```
type|void identifier([parameterlist], params parameterType[] identifier1)
{
    ...
}
```

- ✓ Parameter-Arrays werden mit dem Schlüsselwort **params** gekennzeichnet. Parameter-Arrays sind immer optional, da das Array auch kein Element enthalten kann.
- ✓ Parameter-Arrays müssen immer als letzter Parameter aufgeführt werden. Sie können nicht als Referenz (**ref**) oder als Ausgabeparameter (**out**) angegebenen werden.
- ✓ Parameter-Arrays sind immer eindimensional.
- ✓ Mit einem Parameter-Array können Sie einer Methode bei jedem Aufruf eine andere Anzahl von Parametern übergeben. Als Parameter können Sie entweder ein Datenfeld oder eine durch Kommata getrennte Liste von Variablen oder Konstanten übergeben.

Beispiel: AverageOfParamArray.sln

Der Methode `Average()` wird ein Parameter-Array beliebiger Größe übergeben. Die Methode berechnet den Mittelwert der übergebenen Zahlenwerte und gibt das Ergebnis zurück.

```

① static double Average(params double[] numbers)
{
    double sum = 0;
    foreach (double d in numbers)
    {
        sum += d;
    }
    ③ return sum / numbers.Length;
}

static void Main(string[] args)
{
    ④ Console.WriteLine(Average(0, 1, 3, 5));
    Console.WriteLine(Average(2, 4));
    Console.WriteLine(Average(1, 1, 3, 5, 7));
}

```

- ① Die Methode `Average()` wird deklariert. Ihr kann als Parameter-Array ein Datenfeld aus beliebig vielen Werten vom Typ `double` übergeben werden.
- ② In der `foreach`-Schleife werden alle Werte des Arrays aufsummiert.
- ③ Zur Berechnung des Mittelwertes wird die Summe durch die Anzahl der übergebenen Werte geteilt. Das Ergebnis wird zurückgegeben.
- ④ Die mit einem Parameter-Array definierte Methode kann mit einer beliebigen Anzahl von Parametern aufgerufen werden.

2,25
3
3,4

Die Ausgabe des Programms

Anstelle der einzelnen Werte kann auch ein Datenfeld (Array) übergeben werden.

Beispiel: Adresszeilen.sln

Die Methode `Split()` zerlegt einen String in Teilzeichenfolgen. Die Trennzeichen werden als Parameter-Array ① übergeben. Ein String-Array mit den Teilzeichenfolgen wird zurückgegeben:

```

static void Main(string[] args)
{
    string txt = "Name,Vorname,1234 Ort,Straße 5,1234/56789";
    string[] adr = txt.Split(new char[] { ',', ' ', ' ', '/' });
    foreach (string zeile in adr) ①
    {
        Console.WriteLine(zeile);
    }
}

```

Name
Vorname
1234
Ort
Straße
5
1234
56789

Programmcode

Programmausgabe

Rein aus C# heraus gibt es kaum Gründe für die Einführung von Parameter-Arrays. Man kann stattdessen jederzeit als Alternative ein Array oder ein Objekt vom Typ einer Collection als Parameter verwenden. Parameter-Arrays haben ihre Bedeutung hauptsächlich in Verbindung mit externen Technologien im .NET-Habitat und Visual Studio, da sie von der Aufrufseite mit einfachen Parametern zu füllen sind. Beachten Sie, dass die Verwendung von Parameter-Arrays in Verbindung mit dem Überladen von Methoden sogar zu Problemen mit der Eindeutigkeit führen kann. Wie bei optionalen Parametern können Situationen entstehen, bei denen der Compiler bei einem Methodenaufruf nicht eindeutig entscheiden kann, welche Methode aufgerufen werden soll. Das nachfolgende Beispiel verdeutlicht diese Problematik.

Beispiel: AufrufKonflikt.sln

Die Methode Test () wird in der Klasse Class1 in drei Varianten deklariert. Einmal ohne Parameter, einmal mit einem Parameter-Array und einmal mit einem optionalen Parameter. Die Methode wird also zweimal überladen.

```

① /// <summary>
/// Ohne Parameter
/// </summary>
internal void Test()
{
    Console.WriteLine("Methode 1");
}

② /// <summary>
/// Parameter-Array
/// </summary>
/// <param name="i"></param>
internal void Test(params int[] i)
{
    Console.WriteLine("Methode 2");
}

③ /// <summary>
/// Optionale Parameter
/// </summary>
/// <param name="i"></param>
internal void Test(int i=0)
{
    Console.WriteLine("Methode 3");
}
...
...

④ static void Main(string[] args)
{
    new Class1().Test();      // Methode ohne Parameter wird aufgerufen.
                            // Die anderen beiden theoretisch passenden
                            // Methoden sind nicht auswählbar.
    new Class1().Test(1);    // Methode mit optionalem Parameter wird
                            // aufgerufen. Die theoretisch passende Methode
                            // mit Parameter-Array ist nicht auswählbar.
}
...
...

```

- ① Die Methode Test () wird ohne Parameter deklariert.
- ② Eine weitere Methode Test () wird deklariert. Ihr kann als Parameter-Array ein Datenfeld aus beliebig vielen Werten vom Typ int übergeben werden.
- ③ Als dritte Überladung gibt es die Methode mit einem optionalen Parameter vom Typ int.
- ④ In der Programmklasse wird die Klasse Class1 instanziert. Die jeweiligen Aufrufe der Methode Test lassen keinen eindeutigen Schluss zu, welche der verfügbaren überladenen Methoden zu nutzen ist. Im ersten Aufruf passen alle drei Deklarationen, im zweiten Aufruf die letzten beiden Deklarationen.

Der Compiler muss eine Auswahl vornehmen, die aber prinzipiell mögliche Aufrufe dann ausschließt. Wenn Sie die jeweils standardmäßig gewählte Deklaration auskommentieren, sehen Sie, dass in dem Fall die andere Methode zum Aufruf kommt.

Prinzipiell sollte man beim Verwenden von Parameter-Arrays als auch optionalen Parametern in C# Vorsicht walten lassen. In den meisten Fällen ist die Angabe von einem Array als Parameter oder eines Collection-Typs die bessere Wahl.

Parameterübergabe an die `Main`-Methode

Die `Main()`-Methode einer Konsolenanwendung besitzt standardmäßig als Parameter ein Array vom Typ `String`. Alle Zeichenketten, die Sie in dem Befehl zur Ausführung des Programms hinter dem Klassennamen des Projektes anfügen, werden als Parameter der `Main()`-Methode übergeben.

Durch den Aufruf `ParameterInfo Param1 10 12.23`

werden der Methode `Main()` der Klasse `Program` die Parameterwerte `Param1`, `10` und `12.23` als `Strings` übergeben. Die Zeichenketten müssen durch Leerzeichen getrennt werden.

Beispiel: `ParameterInfo.sln`

Das Programm (`ParameterInfo`) zeigt alle Parameter an, die Sie zusätzlich zum Klassennamen übergeben.

Der Aufruf `ParameterInfo Hallo C#-Programmierer`

erzeugt beispielsweise die Ausgabe

```
Hallo
C#-Programmierer
```

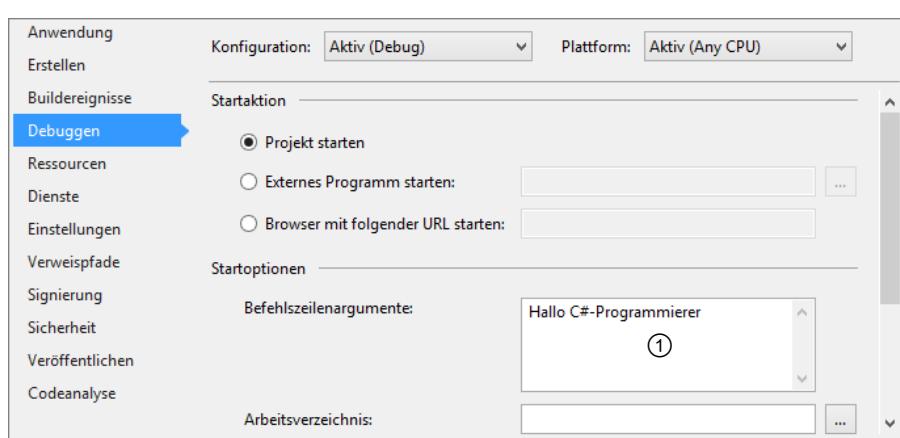
- ① Der `Main`-Methode werden alle Parameter, die Sie beim Programmaufruf hinter dem Programmnamen anfügen, als Feld von Typ `string` übergeben.
- ② Die Anzahl der Parameter können Sie über die Eigenschaft `Length` ermitteln. Haben Sie keine Parameter übergeben, wird die `for`-Schleife nicht durchlaufen.
- ③ Die Parameter werden zeilenweise ausgegeben.
- ④ Die Ausgabe ist mit einer `foreach`-Schleife ebenfalls möglich.

```
① static void Main(string[] args)
{
    ② for (int i = 0; i < args.Length; i++)
    {
        ③ Console.WriteLine(args[i]);
    }
    ④ foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```

Programmargumente (Parameter) in Visual Studio angeben

In Visual Studio haben Sie die Möglichkeit, wie in einer Kommandozeile Parameter beim Aufruf des Programms anzugeben.

- Öffnen Sie die *Projekt-eigenschaften* und wechseln Sie in das Register *Debuggen*.
- Geben Sie im Eingabefeld *Befehlszeilenargumente* ① die gewünschten Parameter an.



13.5 Auflistungen

Alternativ zu Arrays können Sie auch Auflistungen, sogenannte Collections, verwenden, um mehrere Daten eines Typs zu speichern und zu verwalten.

- ✓ In Auflistungen können einzelne Elemente beispielsweise flexibler hinzugefügt und entfernt werden als bei Arrays.
- ✓ Die Größe der Auflistung muss zuvor nicht festgelegt werden.
- ✓ Für viele Aktionen wird kein Index benötigt. Bei manchen Arten von Auflistungen können zur Suche Schlüsselwörter verwendet werden.
- ✓ Bei der Arbeit mit sich ständig ändernden Gruppen von Elementen sind Auflistungen zu empfehlen. Diese können sowohl bei einer kleinen als auch bei einer großen Elementanzahl nützlich sein.

Auch bei einigen Steuerelementen, wie z. B. `ListBox`, `ComboBox`, `ListView`, `TreeView`, werden Auflistungen eingesetzt.

Das .NET Framework stellt verschiedene Klassen für Auflistungen bereit, um beispielsweise Listen, Warteschlangen, Stapel, Wörterbücher oder sogenannte Hashtabellen zu erstellen. Nachfolgend werden einige dieser Auflistungen vorgestellt. Sie müssen dazu den Namespace `System.Collections` zu Beginn einbinden.

13.6 Die Klasse `ArrayList`

Eine häufig verwendete Klasse für Aufzählungen ist die Klasse `ArrayList`. Sie implementiert mehrere Interfaces, wie beispielsweise das Interface `IList` und das Basis-Interface `ICollection`. Die nachfolgende Tabelle zeigt einige ausgewählte Methoden und Eigenschaften der Klasse `ArrayList`.

Ausgewählte Methoden und Eigenschaften einer <code>ArrayList</code>	Bedeutung
<code>Count</code>	Gibt die Anzahl der Listenelemente zurück
<code>Capacity</code>	Gibt die Kapazität der Liste an. Standardmäßig wird beim Hinzufügen des ersten Elements zunächst Platz für 4 Elemente bereitgestellt. Wenn dies nicht ausreicht, wird die Kapazität jeweils verdoppelt.
<code>Add()</code>	Fügt am Ende der Liste ein neues Element hinzu
<code>Insert()</code>	Fügt an einer bestimmten Indexposition ein neues Element ein
<code>Clear()</code>	Löscht alle Elemente der Liste
<code>Remove()</code>	Sucht und entfernt das erste Element, das mit einem übergebenen Suchmuster übereinstimmt
<code>RemoveAt()</code>	Löscht ein Element an der übergebenen Indexposition

Beispiel: *Liste.sln*

Das folgende Beispiel zeigt den Umgang mit einer Auflistung vom Typ `ArrayList`. Das Programm fügt einer Auflistung 20 Elemente hinzu und löscht davon zwei Elemente mit verschiedenen Methoden. Dabei wird auch die Flexibilität der Liste gegenüber dem einfachen Array deutlich, denn in einem Array können keine einzelnen Elemente gelöscht werden.

```
using System.Collections;
...
static void Main(string[] args)
{
    ① ArrayList oneList = new ArrayList();
    ② for (int i = 0; i < 20; i++)
    {
        oneList.Add("Inhalt von Listenelement " + i.ToString());
    }
    ③ Console.WriteLine("Anzahl der Elemente: {0}", oneList.Count);
    Console.WriteLine("Kapazität der Liste: {0}", oneList.Capacity);
    ④ oneList.RemoveAt(10);
    ⑤ oneList.Remove("Inhalt von Listenelement 15");
    ⑥ foreach (string s in oneList)
    {
        Console.WriteLine(s);
    }
    ⑦ Console.WriteLine("Anzahl der Elemente: {0}", oneList.Count);
}
```

- ① Hier wird eine Auflistung `oneList` vom Typ `ArrayList` vereinbart.
- ② Mithilfe einer Schleife werden der Liste 20 Einträge hinzugefügt. Beim Hinzufügen des ersten Eintrags erhält die Liste Platz für 4 Elemente. Dieser Platz wird bei Bedarf jeweils automatisch verdoppelt, sodass die Kapazität beispielsweise auf 8 erhöht wird, wenn das 5. Element hinzugefügt wird.
- ③ Über die Eigenschaften `Count` und `Capacity` werden die aktuelle Anzahl der Listenelemente und die aktuelle Kapazität der Liste ausgegeben.
- ④ Das Listenelement mit dem Index 10 wird entfernt.
- ⑤ Das erste mit dem angegebenen Suchmuster übereinstimmende Element wird gelöscht.
- ⑥ Die Listenelemente werden mithilfe einer `foreach`-Schleife ausgegeben.
- ⑦ Abschließend wird noch einmal die Anzahl der Listenelemente ausgegeben. Sie hat sich nun um die zwei gelöschten Elemente verringert.

```
Anzahl der Elemente: 20
Kapazität der Liste: 32
Inhalt von Listenelement 0
Inhalt von Listenelement 1
Inhalt von Listenelement 2
Inhalt von Listenelement 3
Inhalt von Listenelement 4
Inhalt von Listenelement 5
Inhalt von Listenelement 6
Inhalt von Listenelement 7
Inhalt von Listenelement 8
Inhalt von Listenelement 9
Inhalt von Listenelement 10
Inhalt von Listenelement 11
Inhalt von Listenelement 12
Inhalt von Listenelement 13
Inhalt von Listenelement 14
Inhalt von Listenelement 15
Inhalt von Listenelement 16
Inhalt von Listenelement 17
Inhalt von Listenelement 18
Inhalt von Listenelement 19
Anzahl der Elemente: 18
```

Die Ausgabe des Programms

13.7 Auflistungsinitialisierer

Das Befüllen einer Auflistung erfolgt in der Regel über mehrere Aufrufe der `Add()`-Methode. Sollen zu Beginn gleich zahlreiche Elemente aufgenommen werden, sind dementsprechend viele Aufrufe der `Add()`-Methode notwendig. Durch den Einsatz von Auflistungsinitialisierern, die mit dem .NET Framework 3.5 eingeführt wurden, vereinfacht sich nun diese Aufgabe. Allen Auflistungen, die das Interface `IEnumerable` implementieren, können jetzt schon bei der Objekterstellung Elemente hinzugefügt werden.

```
ArrayList listName = new ArrayList {"Element1", "Element2", "Element3"};
```

Als Besonderheit werden beim Erstellen des Objekts keine runden Klammern mehr angegeben. Die Elemente werden einfach in geschweiften Klammern, mehrere durch Kommata getrennt, angegeben.

Beispiel: *AuflistungsInit.sln*

Eine `ArrayList` wird in diesem Beispiel gleich zu Beginn mit 3 Elementen initialisiert, die im Folgenden über eine `foreach`-Schleife ausgegeben werden.

```
① using System;
② using System.Collections;
namespace AuflistungsInit
{
    class Program
    {
        static void Main(string[] args)
        {
            ③ ArrayList fahrzeugTypen = new ArrayList {"PKW", "Motorrad", "Fahrrad"};

            foreach(string fahrzeug in fahrzeugTypen)
                Console.WriteLine(fahrzeug);
        }
    }
}
```

- ① Zur Verwendung der Klasse `ArrayList` wird der Namespace `System.Collections` eingebunden.
- ② Bei der Erzeugung des `ArrayList`-Objekts werden hier keine runden Klammern verwendet, da eine Initialisiererliste genutzt wird. Die Elemente werden dazu in geschweiften Klammern angegeben. Die Anweisung wird mit einem Semikolon beendet.
- ③ Die Listenelemente werden über eine `foreach`-Schleife ausgegeben.

13.8 Listen mit einem Enumerator durchlaufen

Alle Auflistungen und Arrays implementieren die Schnittstelle `IEnumerable`. Dadurch steht ein Positionszeiger, ein sogenannter **Enumerator** (heute meist **Iterator** genannt), zur Verfügung, mit dessen Hilfe Sie einen iterativen Lesezugriff auf die Auflistungs- oder Array-Elemente erhalten. Im Unterschied zur `foreach`-Schleife lässt sich der Enumerator mit der Methode `GetEnumerator()` ermitteln, mit `Reset()` jederzeit auf den Anfang der Liste zurücksetzen und mit `MoveNext()` zum jeweils nächsten Element bewegen.

Beispiel: *ListeMitEnumerator.sln*

- ✓ Eine Auflistung vom Typ ArrayList wird zunächst mit 10 Beispielwerten gefüllt. Anschließend wird der Enumerator der Auflistung ermittelt und mit dem Enumerator die Liste durchlaufen.

```

using System.Collections;
static void Main(string[] args)
{
    ArrayList oneArrayList = new ArrayList();
    for (int i = 0; i < 10; i++)
    {
        oneArrayList.Add("test " + i.ToString());
    }

    IEnumator e = oneArrayList.GetEnumerator();
    while (e.MoveNext())
    {
        Console.WriteLine(e.Current.ToString());
    }
}

```

- ① Eine Auflistung vom Typ ArrayList wird erzeugt.
- ② In einer for-Schleife wird die Liste mit 10 Beispielwerten gefüllt.
- ③ Der Enumerator der Auflistung wird ermittelt.
- ④ In einer while-Schleife wird der Enumerator jeweils zum nächsten Element bewegt. Beim ersten Aufruf wird der Enumerator auf das erste Element der Liste gesetzt.
- ⑤ Über die Eigenschaft Current wird das entsprechende Listenelement auf dem Bildschirm angezeigt.

Individuellen Iterator erstellen: *Iterator.sln*

Das folgende Beispiel zeigt, wie Sie auf einfache Weise für eine Klasse einen Iterator erstellen, der eine individuelle Ausgabereihenfolge ermöglicht. Dazu muss die Klasse, für die Sie den Iterator erstellen möchten, die Schnittstelle **IEnumerable** und somit die Methode **GetEnumerator()** implementieren.

```

public class MonthNames : IEnumerable
{
    private string[] months = {"Jan", "Feb", "Mar", "Apr", "Mai", "Jun",
                                "Jul", "Aug", "Sep", "Okt", "Nov", "Dez"};
    IEnumerator IEnumerable.GetEnumerator()
    {
        for (int i = 0; i < months.Length; i += 3)
        {
            yield return months[i];
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MonthNames quartalsAnfang = new MonthNames();

```

```

⑦ foreach (string month in quartalsAnfang)
{
    Console.WriteLine(month);
}
}

```

- ① Eine Klasse MonthNames wird deklariert. Damit die Elemente der Klasse mit einem Iterator durchlaufen werden können, wird das Interface `IEnumerable` implementiert.
- ② Die Klasse enthält ein Feld mit den Monatsnamen als `string`-Array.
- ③ Die Methode `GetEnumerator()` der Schnittstelle wird implementiert.
- ④ Der Iterator soll in diesem Beispiel beginnend bei dem 1. Wert "Jan" (index 0) jeden dritten Wert zurückgeben.
- ⑤ Für die Rückgabe wird das Schlüsselwort `yield` (dt. liefern) vor `return` gesetzt. Während die `foreach`-Schleife ⑦ abläuft, liefert die Methode `GetEnumerator()` über die Schlüsselwörter `yield return` den Listenwert. `yield` bewirkt, dass die `for`-Schleife ④ anschließend jedoch nicht gleich fortgesetzt wird. Die Schleife wartet, bis die `foreach`-Schleife ⑦ den nächsten Wert abruft.
- ⑥ Eine Instanz der Klasse MonthNames wird erzeugt und der Variablen `quartalsAnfang` zugewiesen.
- ⑦ In der `foreach`-Schleife werden die Werte ausgegeben. Automatisch wird dabei der neu definierte Iterator berücksichtigt: Es wird nur der jeweils dritte Monat ausgegeben (Quartalsanfang).

```

Jan
Apr
Jul
Okt

```

Die Ausgabe des Programms

13.9 Typsicherheit bei Auflistungen und Generics

Arrays sind sogenannte klassenlose Objekte. Wenn Sie die Deklaration eines Arrays anschauen, wird dabei **kein** Konstruktor einer Klasse verwendet. Auch wenn die Verwendung von `new` das fälschlicherweise suggeriert. Bei der Deklaration wird aber explizit ein Datentyp festgelegt, was als Inhalt in einem Array gespeichert werden darf. Arrays sind damit typsicher. Diese Festlegung eines Datentyps fehlt vollkommen bei den Auflistungen und verwandten Strukturen.

Im Gegensatz zu den nur semidynamischen Arrays (wenn die Größe einmal festgelegt ist, kann sie nicht mehr geändert werden) sind Auflistungen bzw. Collections jedoch dynamisch, was gerade auch schon demonstriert wurde. Diesen Vorteil erkauft man aber wie gesagt teuer mit dem Verzicht auf jedwede Typsicherheit.

Mit der Einführung von sogenannten **Generics** wurde dieses Problem reduziert. Generics führt das Konzept der Typparameter in .NET Framework ein. Damit können Klassen, Methoden und eben auch manche Auflistungen bzw. Collections (aber nicht alle) gekennzeichnet werden, dass sie nur bestimmte Datentypen akzeptieren sollen. Das Thema ist sowohl anspruchsvoll als auch umfangreich und sprengt den Rahmen des Buchs, soll aber zumindest beispielhaft für eine Collection demonstriert werden.

Eine typsichere Liste erstellen: *GenerischeListe.sln*

```

① using System;
using System.Collections;
using System.Collections.Generic;
namespace GenerischeListe
{
    class Program
        static void Main(string[] args)
        {

```

```

②    // int als Typ des Arguments
List<int> list = new List<int>();
③    for (int x = 0; x < 10; x++)
{
    list.Add(x);
}
foreach (int i in list)
{
    System.Console.Write(i + " ");
}
System.Console.WriteLine("\nFertig");
}
}
}

```

- ① Zur Verwendung von Generics mit `System.Collections.Generic` wird das Interface `IEnumerable` implementiert.
- ② In den spitzen Klammern wird bei der Deklaration der Liste der Datentyp festgelegt, den die Liste aufnehmen kann.
- ③ Die Methode `Add()` kann nur noch Elemente hinzufügen, die dem Datentyp entsprechen oder kompatibel dazu sind.

13.10 Indexer

Eng verwandt mit den Eigenschaften sind sogenannte Indexer. Mit ihrer Hilfe lassen sich Objekte wie Arrays indizieren. Wenn die Instanz einer Klasse z. B. intern ein Array von Zeichenketten verwaltet, haben Sie mehrere Möglichkeiten, den Zugriff auf die Array-Elemente zu realisieren.

- ✓ Sie deklarieren das Array als `public` und verwenden folgende Syntax:

```
objektvariable.Arrayname[index] = ...
```

- ✓ Sie schützen das Array als `private` und verwenden einen Indexer:

```
objektvariable[index] = ...
```

- ✓ Sie verwenden eine Methode zum Zugriff auf die Array-Elemente:

```
Zugriffsmethode(index) ...
```

Syntax für die Deklaration eines Indexers

- ✓ Die Deklaration des Indexers beginnt mit der optionalen Angabe eines Modifiers und des Datentyps.
- ✓ Danach folgt das Schlüsselwort `this` als Verweis auf das Objekt selbst, für das der Indexer gilt. Anschließend folgt ein Index oder eine Liste mehrerer Indizes als Parameter. Indexer müssen mindestens einen Parameter besitzen.
- ✓ Bei Bedarf kann ein Zugriffsmodifizierer vorangestellt werden. Ohne Angabe werden Indexer wie alle Klassenmember als `private` vereinbart. In der Regel sollten Indexer als `public` deklariert werden, da sie die Schnittstelle eines Objektes nach außen bilden.

```

[modifier] type this[indexType
index, ...]
{
    [modifier] get
    {
        ...
    }
    [modifier] set
    {
        ...
        ... = value; ②
    }
}

```

- ✓ In geschweiften Klammern `{ }` ① folgen die sogenannten Accessoren `get` und `set`. Das sind spezielle Methoden, die den Zugriff auf die entsprechenden Felder regeln. Wenn eine Nur-Lese-Eigenschaft gewünscht ist, kann der `set`-Block wegfallen, beim Nur-Schreib-Zugriff der `get`-Block.
- ✓ Im `set`- und `get`-Block lässt sich z. B. eine Kontrolle auf Über- oder Unterschreitung des Indexbereichs durchführen. Dazu können auch weitere Methoden aufgerufen werden.
- ✓ Im `get`-Block kann der Wert des vom Index bezeichneten Feldes bzw. Array-Elements zurückgegeben werden.
- ✓ Ein dem Indexer zugewiesener Wert kann mithilfe des Schlüsselworts `value` in das vom Index bezeichnete Feld bzw. Array-Element geschrieben werden. Dazu wird das Schlüsselwort `value` dem Feld bzw. Array-Element zugewiesen ②.
- ✓ Indexer lassen sich (wie Methoden und Eigenschaften) überladen.
- ✓ Wie bei Eigenschaften kann **einer** der Accessoren `get` oder `set` einen Zugriffsmodifizierer erhalten. Dieser Zugriffsmodifizierer kann die Zugriffsrechte, die für den Indexer festgelegt sind, nur einschränken, aber nicht erweitern.

Beispiel: *Index.sln*

Im folgenden Programm erfolgt der Zugriff auf das Zeichenkettenarray `meldungen` über einen Indexer. Auf diesem Weg werden zwei Elemente des Zeichenkettenfeldes mit Werten belegt. Anschließend werden alle fünf Elemente des Arrays auf dem Bildschirm ausgegeben.

```

class Zeichenketten
{
    ① private string[] meldungen = new string[5];
    ② public string this[int index]
    {
        ③     get
        {
            if ((index >= meldungen.GetLowerBound(0)) &&
                (index <= meldungen.GetUpperBound(0)))
                return meldungen[index];
            else
            {
                Console.WriteLine("ungültiger Index!");
                return null;
            }
        }
        ④     set
        {
            if ((index >= meldungen.GetLowerBound(0)) &&
                (index <= meldungen.GetUpperBound(0)))
                meldungen[index] = value;
            else
                Console.WriteLine("ungültiger Index!");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
    
```

```

⑤ Zeichenketten textZeilen = new Zeichenketten();
⑥ textZeilen[1] = "Hallo";
⑦ textZeilen[3] = "Welt";
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("textZeilen[{0}]: {1}", i, textZeilen[i]);
}
}
}

```

- ① In der Klasse `Zeichenketten` wird das Array `meldungen` deklariert und initialisiert. Es kann 5 Zeichenketten aufnehmen.
- ② Durch diesen Indexer wird sowohl der Lese- als auch der Schreibzugriff auf das Array `meldungen` gesteuert.
- ③ Mithilfe des `get`-Blocks wird die vom übergebenen Index bezeichnete Zeichenkette des Arrays an die aufrufende Anweisung zurückgegeben. Zuvor erfolgt noch eine Prüfung, ob der Index im erlaubten Bereich liegt. Wenn dies nicht zutrifft, wird eine Fehlermeldung ausgegeben.
- ④ Im `set`-Block wird der von der Variablen `value` repräsentierte Wert dem vom übergebenen Index bezeichneten Element des Arrays `meldungen` zugewiesen. Dies erfolgt allerdings nur, wenn sich der Index im erlaubten Bereich befindet; andernfalls wird eine Fehlermeldung angezeigt.
- ⑤ Eine Instanz der Klasse `Zeichenketten` wird erzeugt und der Objektvariablen `textZeilen` zugewiesen.
- ⑥ Hier erfolgt der Schreibzugriff auf das Array `meldungen` über den Indexer. Dem zweiten und dem vierten Feldelement wird jeweils eine Zeichenkette zugewiesen.
- ⑦ Mittels einer Schleife wird lesend über den Indexer auf das Array `meldungen` zugegriffen und dessen Elemente werden auf dem Bildschirm ausgegeben. Bei den Array-Elementen mit dem Index 1 und 3 werden die zugewiesenen Zeichenketten angezeigt. Bei den übrigen drei Array-Elementen wird nichts angezeigt, da sie bei der Erzeugung der Instanz mit dem Wert `null` initialisiert wurden.

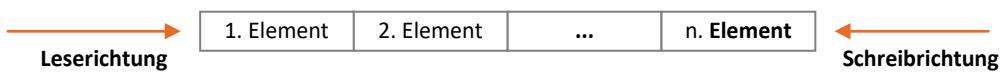
Sie können nur einen Indexer pro Klasse vereinbaren. Möchten Sie mehrere Indexer für mehrere Inhalte verwenden, können Sie mehrere Objekte solcher Klassen als Felder einer übergeordneten Klasse implementieren.

13.11 Warteschlangen

Eine besondere Form von Listen stellt die Warteschlange oder `Queue` (sprich `kju`) dar.

- ✓ Eine Queue arbeitet nach dem **FIFO-Prinzip (First In First Out)**.
- ✓ Neue Elemente können nur am Ende hinzugefügt werden.
- ✓ Die Entnahme von Elementen erfolgt immer am Anfang (Kopf) der Schlange.

Die Warteschlange eignet sich zur Zwischenspeicherung von Daten, die in der Reihenfolge ihres Eintreffens aufbewahrt oder ausgewertet werden sollen.



Beispiel: Schlange.sln

Das Beispiel zeigt die Verwendung einer Liste vom Typ Queue. Das Programm speichert 20 Elemente in einer Warteschlange und entfernt danach wieder zwei davon. Die Anzahl der Listenelemente und ihr Inhalt werden zur Kontrolle auf dem Bildschirm ausgegeben.

```
class Program
{
    static void Main(string[] args)
    {
        ① Queue simpleQueue = new Queue();
        ② for (int i = 0; i < 20; i++)
        {
            simpleQueue.Enqueue("Inhalt von Schlangenelement " +
                i.ToString());
        }
        ③ Console.WriteLine("Anzahl der Elemente: {0}", simpleQueue.Count);
        ④ Console.WriteLine(simpleQueue.Dequeue().ToString() + " entfernt");
        Console.WriteLine(simpleQueue.Dequeue().ToString() + " entfernt");
        ⑤ Console.WriteLine("Anzahl der Elemente: {0}", simpleQueue.Count);
        ⑥ foreach (string s in simpleQueue)
        {
            Console.WriteLine(s);
        }
    }
}
```

- ① Eine Variable `simpleQueue` vom Typ `Queue` wird deklariert.
- ② In einer Schleife werden der Warteschlange mit der Methode `Enqueue()` 20 Einträge jeweils **am Ende hinzugefügt**.
- ③ Über die Eigenschaft `Count` wird die aktuelle Anzahl der Warteschlangenelemente ermittelt und ausgegeben.
- ④ Zwei Elemente werden mithilfe der Methode `Dequeue()` **vom Anfang der Warteschlange entfernt** und zur Kontrolle ausgegeben.
- ⑤ Die Anzahl der Elemente wird erneut ermittelt und ausgegeben.
- ⑥ Die verbliebenen Listenelemente werden mithilfe einer `foreach`-Schleife ausgegeben.

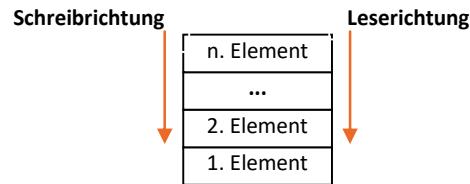
Anzahl der Elemente: 20 ③
 Inhalt von Schlangenelement 0 entfernt ④
 Inhalt von Schlangenelement 1 entfernt ⑤
 Anzahl der Elemente: 18 ⑤
 Inhalt von Schlangenelement 2 ⑥
 Inhalt von Schlangenelement 3
 Inhalt von Schlangenelement 4
 ...
 Inhalt von Schlangenelement 19

Die Ausgabe des Programms

13.12 Stapel

Ein Stapel bzw. Stack (sprich st&ack) ist ebenfalls eine besondere Form der Auflistung.

- ✓ Der Stack arbeitet im Gegensatz zur Queue nach dem LIFO-Prinzip (**Last In First Out**).
- ✓ Neue Elemente werden immer oben auf dem Stapel abgelegt (hinzugefügt) und auch oben wieder entnommen.
- ✓ Die Elemente werden durch diese Form der Speicherung in ihrer Reihenfolge umgekehrt.



Ein Stapel eignet sich zur Zwischenspeicherung von Daten, die in umgekehrter Reihenfolge ihres Eintreffens ausgewertet oder weiterverwendet werden sollen.

Beispiel: *Stacke.sln*

Das Beispiel zeigt die Verwendung einer Liste vom Typ Stack. Das Programm speichert 20 Elemente auf einem Stapel und entfernt anschließend zwei Elemente. Die Anzahl der Listenelemente und ihr Inhalt werden zur Kontrolle auf dem Bildschirm ausgegeben.

```
class Program
{
    static void Main(string[] args)
    {
        ① Stack simpleStack = new Stack();
        ② for (int i = 0; i < 20; i++)
        {
            simpleStack.Push("Inhalt von Stapelelement " + i.ToString());
        }
        ③ Console.WriteLine("Anzahl der Elemente: {0}", simpleStack.Count);
        ④ Console.WriteLine(simpleStack.Pop().ToString() + " entfernt");
        ⑤ Console.WriteLine(simpleStack.Pop().ToString() + " entfernt");
        ⑥ Console.WriteLine("Anzahl der Elemente: {0}", simpleStack.Count);
        foreach (string s in simpleStack)
        {
            Console.WriteLine(s);
        }
    }
}
```

- ① Eine Variable simpleStack vom Typ Stack wird deklariert.
- ② In einer for-Schleife werden mit der Methode Push() 20 Einträge auf dem Stapel abgelegt.
- ③ Über die Eigenschaft Count wird die aktuelle Anzahl der Stapelelemente ermittelt und ausgegeben.
- ④ Zwei Elemente werden mithilfe der Methode Pop() vom Stapel genommen und zur Kontrolle ausgegeben.
- ⑤ Die Anzahl der Stapelelemente wird erneut ermittelt und ausgegeben.
- ⑥ In einer foreach-Schleife werden die verbliebenen Listenelemente ausgegeben.

Anzahl der Elemente: 20	③
Inhalt von Stapelelement 19	entfernt
Inhalt von Stapelelement 18	entfernt
Anzahl der Elemente: 18	⑤
Inhalt von Stapelelement 17	
Inhalt von Stapelelement 16	
Inhalt von Stapelelement 15	
Inhalt von Stapelelement 14	⑥
Inhalt von Stapelelement 13	
...	
Inhalt von Stapelelement 1	
Inhalt von Stapelelement 0	

Die Ausgabe des Programms

13.13 Hash-Tabelle

Eine weitere spezielle Form der Auflistung ist die Hash-Tabelle bzw. Hashtable.

- ✓ Die Elemente der Liste werden mithilfe eines Schlüssels verwaltet. Die Reihenfolge der Elemente in einer Hashtable ist daher belanglos.
- ✓ Der Schlüssel kann einen beliebigen Datentyp besitzen. Gewöhnlich werden Schlüssel vom Typ `string` oder `int` verwendet. Allerdings wird intern nicht der Schlüssel selbst gespeichert, sondern sein sogenannter Hashcode, daher die Bezeichnung Hashtable.
- ✓ Ein Hashcode ist eine Zahl vom Typ `int`, die durch einen speziellen Algorithmus für jedes Objekt errechnet wird.
- ✓ Den Hashcode eines Objekts können Sie mithilfe der Methode `GetHashCode()` ermitteln.
- ✓ Beachten Sie, dass verschiedene Objekte den gleichen Hashcode besitzen können, da dieser immer nur durch einen `int`-Wert repräsentiert wird. In einer Hash-Tabelle werden in diesem Fall diese Objekte noch einmal einzeln überprüft. Allerdings tritt ein solcher Fall eher selten ein.

Lese-/Schreibrichtung	Hashcode n	n. Element

	Hashcode 2	2. Element
	Hashcode 1	1. Element

Beispiel: `Hash.sln`

Das Beispiel zeigt den Umgang mit einer Liste vom Typ `Hashtable`. Das Programm speichert 3 Einträge mit Schlüsseln und Werten. Danach wird die Anzahl der Listeneinträge auf dem Bildschirm ausgegeben. Abschließend wird demonstriert, wie über den Schlüssel auf den Wert eines Eintrags zugegriffen werden kann.

```
class Program
{
    static void Main(string[] args)
    {
        ①      Hashtable ht = new Hashtable();
        ②      ht.Add("Schlüssel_1", "Wert1");
        ③      ht.Add("Schlüssel_2", "Wert2");
        ④      ht.Add("Schlüssel_3", "Wert3");
        ⑤      Console.WriteLine("Schlüssel_1".GetHashCode());
        ⑥      Console.WriteLine("Schlüssel_2".GetHashCode());
        ⑦      Console.WriteLine("Schlüssel_3".GetHashCode());
        ⑧      Console.WriteLine("Einträge: {0}", ht.Count);
        ⑨      Console.WriteLine(ht["Schlüssel_2"]);
    }
}
```

- ① Eine Variable `ht` vom Typ `Hashtable` wird vereinbart.
- ② Der `Hashtable` werden drei Einträge mit Schlüsseln und Werten hinzugefügt.
- ③ Zur Information wird der Hashcode der drei Schlüssel ermittelt und ausgegeben.
- ④ Mithilfe der Eigenschaft `Count` wird die aktuelle Anzahl der Elemente ermittelt und ausgegeben.
- ⑤ Ein Element der Hash-Tabelle wird über einen Indexer angesprochen. Der Schlüssel wird als Index angegeben und der Wert des Elements wird ausgegeben.

-1394059095	③
1334824260	
-231259681	④
Einträge: 3	
Wert2	⑤

Die Ausgabe des Programms

13.14 Mit Aufzählungstypen arbeiten

Was sind Wertemengen?

Häufig werden in Programmen sogenannte Wertemengen benötigt, beispielsweise für

- ✓ Wochentage
- ✓ Werkstage
- ✓ Monate
- ✓ Jahreszeiten

In C# lassen sich Wertemengen als Aufzählungstypen, sogenannte Enumerations, definieren. Eine Enumeration fasst mehrere gleichartige Konstanten zu einer Gruppe zusammen. Man muss aber festhalten, dass Enumerations eher historisch von Bedeutung sind und aus einer Zeit stammen, wo man noch nicht objektorientiert programmiert hat. Man kann alternativ jederzeit echte objektorientierte Strukturen wie Auflistungen oder Klassen verwenden.

Syntax für die Deklaration eines Aufzählungstyps

- ✓ Die Definition beginnt mit dem Schlüsselwort `enum`, gefolgt von dem Namen der Aufzählung.
- ✓ Anschließend kann, abgetrennt durch einen Doppelpunkt `:`, der Datentyp der Aufzählungswerte folgen. Wird der Datentyp nicht angegeben, wird standardmäßig der Datentyp `int` verwendet. Als Datentyp können Sie ganzzahlige Datentypen angeben (wie `byte`, `short`, `int`, `long`).
- ✓ Bei Bedarf können Sie der Deklaration Modifizierer voranstellen. Ohne die Angabe eines Modifizierers wird der Aufzählungstyp als `internal` deklariert.
- ✓ Jedem Element dieser Enumeration ist automatisch eine Ordinalzahl (abzählbare Zahl, 0, 1, 2, ...) zugeordnet, die dieses Element kennzeichnet. Das erste Element erhält die Ordinalzahl 0, das zweite die 1 usw. Mit der Wertzuweisung können Sie diese Reihenfolge überschreiben.

```
[modifier] enum identifier [: Type]
{
    identifier1 [= value1],
    [identifier2 [= value2],
     ...]
}
```

Aufzählungen anwenden

- ✓ Aufzählungen werden innerhalb des .NET Frameworks und in Visual Studio sehr häufig verwendet. Wenn Sie sich die Eigenschaften einer Komponente z. B. mithilfe des Eigenschaftenfensters ansehen, so handelt es sich bei vielen auswählbaren Wertegruppierungen um Enumerationen.
- ✓ Mithilfe der Methode `Enum.GetNames()` können Sie die einzelnen Konstanten und mit der Methode `Enum.GetValues()` die Ordinalwerte einer Aufzählung ermitteln. Dabei muss der Typ der Aufzählung übergeben werden:
`Enum.GetNames(typeof(identifier))`

FlatStyle	Standard	▼
Font	Flat	
ForeColor	Popup	
GenerateMember	Standard	
Image	System	
ImageAlign	MiddleCenter	

Beispiel: `EnumGroesse.sln`

Die folgende Deklaration vereinbart den Aufzählungstyp `Groesse` mit den Werten `Klein`, `Mittelgross` und `Gross` und gibt anschließend die Konstanten und Ordinalwerte aus.

```
(1) enum Groesse
{
(2)     Klein = 5,
        Mittelgross = 10,
```

```

③ Gross = Klein + Mittelgross
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Wert von {0}: {1}", Groesse.Gross, (int)Groesse.Gross);
        Console.WriteLine("Wert von {0} + 1: {1}", Groesse.Gross, 1 + Groesse.Gross);
        foreach (string s in Enum.GetNames(typeof(Groesse)))
        {
            Console.WriteLine(s);
        }
        foreach (int i in Enum.GetValues(typeof(Groesse)))
        {
            Console.WriteLine(i);
        }
    }
}

```

- ① Die Aufzählung `Groesse` wird deklariert.
- ② Die Ordinalwerte der Elemente werden mit Wertzuweisungen individuell festgelegt.
- ③ Der Ordinalwert für die Konstante `Gross` wird aus den Ordinalwerten anderer Elemente der Aufzählung berechnet.
- ④ Name und Ordinalwert der Konstanten `Gross` werden ausgegeben. Beachten Sie, dass für die Ausgabe des Wertes eine Typumwandlung erforderlich ist.
- ⑤ Hier kann die Typumwandlung entfallen, da der Wert in einen Berechnungsausdruck eingebunden ist.
- ⑥ Mit der Methode `GetNames()` werden die Namen der Konstanten als Array zurückgegeben. Die Elemente dieses Arrays sind vom Typ `string` und werden in einer `foreach`-Schleife ausgegeben.
- ⑦ Mit der Methode `GetValues()` werden dementsprechend die Ordinalwerte der Elemente der Aufzählung ermittelt und ausgegeben.

Wert von Gross: 15
 Wert von Gross + 1: 16
 Klein
 Mittelgross
 Gross
 5
 10
 15

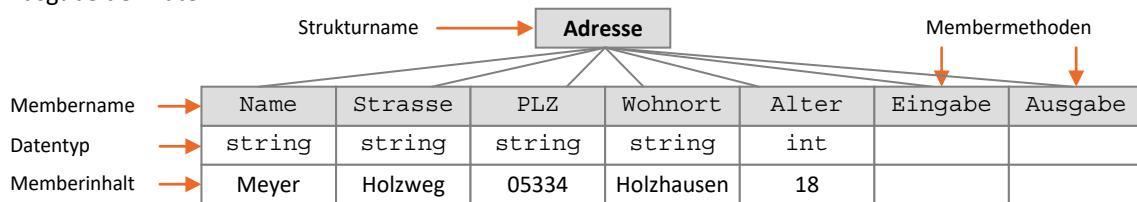
Die Ausgabe des Programms

13.15 Strukturen

Mit **Strukturen** haben Sie die Möglichkeit, zusammengehörige Daten mit unterschiedlichen Datentypen unter einem Bezeichner zu speichern. Jedes Element (**Member**) eines solchen Datenverbunds hat einen Namen und einen Datentyp. Außerdem können Strukturen Methoden für das Arbeiten mit den gespeicherten Daten enthalten. Strukturen gehören zu den benutzerdefinierten Datentypen. Nach der Deklaration einer Struktur können Sie Variablen vom Datentyp dieser Struktur erzeugen und einsetzen. Unterschiede zwischen Klassen und Strukturen werden in Abschnitt 13.16 erläutert. Aber schon jetzt der Hinweis, dass Strukturen so etwas wie der Vorläufer von Klassen in nicht objektorientierten Welten (etwa in C) waren. Die reine Deklaration ist – bis auf das einleitende Schlüsselwort – so gut wie identisch.

Beispiel für eine Struktur **Adresse**

Nachfolgend wird der Aufbau einer Adressdaten-Struktur gezeigt, die die wesentlichen Bestandteile einer Adresse in verschiedenen Datentypen speichert. Zusätzlich enthält die Struktur noch Methoden für die Ein- und Ausgabe der Daten.



Syntax für die Definition einer Struktur

- ✓ Die Deklaration einer Struktur beginnt mit dem Schlüsselwort **struct** und dem Bezeichner.
- ✓ Bei Bedarf kann ein Modifizierer vorangestellt werden. Standardmäßig sind Strukturen mit dem Modifizierer **internal** versehen.
- ✓ Es folgt die Deklaration der in der Struktur enthaltenen Variablen, die ebenfalls als Member (Mitglieder) bezeichnet werden. Die Member können unterschiedliche Datentypen besitzen. Eine einfache Deklaration ohne Zugriffsmodifizierer entspricht einer Deklaration mit **private**.
- ✓ Datenmember einer Struktur können nicht bei der Deklaration initialisiert werden. Dazu bedarf es einer zusätzlichen Wertzuweisung.
- ✓ Strukturen können in Klassen und Schnittstellen, nicht aber in Methoden deklariert werden.
- ✓ Im Unterschied zu anderen Programmiersprachen können Strukturen in C# auch Methoden als Member enthalten. Sie erhalten dadurch eine Ähnlichkeit mit Klassen.
- ✓ Strukturen können nicht geschachtelt werden und kennen keine Vererbung.

```
[modifier] struct Name
{
    [modifier] type member1;
    [modifier] type member2;
    ...
    [modifier] void|type Methodname(Parameterlist)
    {
        ...
    }
    ...
}
```

Auf die Member einer Struktur zugreifen

- ✓ Wie auch von einer Klasse werden Objekte vom Typ einer Struktur erzeugt. objektname.Membername...
- ✓ Der Zugriff auf die Member einer Struktur erfolgt wie bei Klassen über die Objektvariable, einen Punkt und den Namen des öffentlichen Members.

Beispiel: **Adresse.sln**

Das Programm speichert zusammengehörige Adressdaten in einer Struktur. Die Daten werden durch die Angabe **private** vor einem Zugriff von außen geschützt. Nur die beiden Membermethoden **Eingabe()** und **Ausgabe()** sind von außen zugänglich und erlauben den Zugriff auf die Daten.

```

① struct Adresse
{
    private int nummer, alter;
    private string name, strasse, plz, ort;
② public void Eingabe(int nummer, string name, string strasse, string
    plz,
                      string ort, int alter)
    {
        this.nummer = nummer;
        this.name = name;
        this.strasse = strasse;
        this.plz = plz;
        this.ort = ort;
        this.alter = alter;
    }
④ public void Ausgabe()
{
    Console.WriteLine("Laufende Nummer: " + nummer);
    Console.WriteLine("Name: " + name);
    Console.WriteLine("Strasse: " + strasse);
    Console.WriteLine("PLZ: " + plz);
    Console.WriteLine("Ort: " + ort);
    Console.WriteLine("Alter: " + alter);
}
⑤ }

class Program
{
    static void Main(string[] args)
    {
⑥     Adresse adr = new Adresse();
        int neuesAlter;
        string neuerName, neueStrasse, neuePlz, neuerOrt;
⑦     Console.WriteLine("Bitte geben Sie Ihren Namen ein:");
        neuerName = Console.ReadLine();
        Console.WriteLine("Bitte geben Sie Ihre Strasse ein:");
        neueStrasse = Console.ReadLine();
        Console.WriteLine("Bitte geben Sie Ihre Postleitzahl ein:");
        neuePlz = Console.ReadLine();
        Console.WriteLine("Bitte geben Sie Ihren Wohnort ein:");
        neuerOrt = Console.ReadLine();
        Console.WriteLine("Bitte geben Sie Ihr Alter ein:");
        neuesAlter = Convert.ToInt32(Console.ReadLine());
        adr.Eingabe(1, neuerName, neueStrasse, neuePlz, neuerOrt, neuesAlter);
        adr.Ausgabe();
    }
}

```

- ① Der Datentyp **Adresse** wird als Struktur deklariert. In die Struktur werden die entsprechenden Variablen zur Speicherung einer Adresse aufgenommen.
- ② Die Membermethode **Eingabe()** speichert die übergebenen Adressdaten in den Membervariablen.
- ③ In diesem Beispiel wurden für die Parameter der Methode die gleichen Namen wie für die Membervariablen verwendet. Um in der Methode die Membervariablen, d. h. die Variablen der **Struktur**, anzusprechen, verwenden Sie das Schlüsselwort **this**. Der Vorteil dieser (meist optionalen) Schreibweise ist, dass damit deutlicher wird, dass es sich um Membervariablen des betreffenden Objekts handelt.

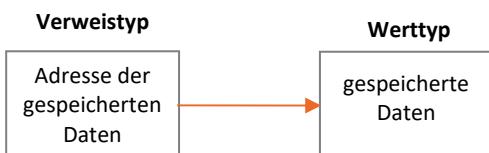
- ④ Mithilfe der Membermethode `Ausgabe()` ist es möglich, die in der Struktur gespeicherten Daten auf dem Bildschirm auszugeben.
- ⑤ Hier endet die Definition der Struktur.
- ⑥ Eine Variable `adr` vom Typ der Struktur `Adresse` wird deklariert und initialisiert.
- ⑦ In der Hauptmethode werden alle für die Adresse notwendigen Angaben abgefragt und in entsprechenden Variablen zwischengespeichert.
- ⑧ Mit der Methode `Eingabe()` werden die Adressdaten an die Struktur übergeben.
- ⑨ Die Ausgabe der Adresse erfolgt mit der Methode `Ausgabe()`.

13.16 Speicherverwaltung

Die Datentypen von C# werden grundsätzlich unterschieden in Werttypen und Verweistypen. Die folgende Tabelle zeigt, welche Datentypen zu welchem dieser beiden Typen gehören.

Werttypen	Verweistypen (Referenztypen)
<code>byte, short, int, long</code>	<code>string, object</code>
<code>float, double, decimal</code>	Arrays
<code>bool, char, enum, struct</code>	<code>class, interface, delegate</code>

- ✓ Ein Datentyp ist ein **Werttyp**, wenn die Daten in der eigenen Speicherreservierung einer Variablen dieses Datentyps enthalten sind.
- ✓ Ein **Verweistyp** enthält dagegen die Adresse eines anderen Speicherorts, an dem die Daten gespeichert sind.



- ✓ Die beiden Typen unterscheiden sich auch hinsichtlich des Speicherorts. Werttypen werden auf dem sogenannten **Stack** (Stapel) des Rechners, Verweistypen auf dem sogenannten **Heap** (Haufen) angelegt. Stack und Heap sind spezielle vom jeweiligen Programm reservierte Bereiche des Hauptspeichers eines Rechners. Aktionen auf dem Heap können sehr flexibel gestaltet werden, sind aber speicherintensiv. Aktionen auf dem Stack haben den Vorteil einer höheren Ausführungsgeschwindigkeit.



Speicherbereiche eines Programms

- ✓ Sie können einer Variablen vom Datentyp `object` sowohl einen Verweistyp als auch einen Werttyp zuweisen. Die Variable verhält sich bei Zuweisung eines Werttyps so, als enthielte sie eigene Daten. Dies wird erreicht, indem bei der Aufnahme des Werttyps eine Datenbox auf dem Heap eingerichtet wird. Werden diese Daten dann wieder einem Werttyp zugewiesen, wird die Box entfernt. Der ganze Vorgang wird auch als **Boxing** und **Unboxing** bezeichnet.

Garbage Collection

Für alle Verweistypen wird dynamisch der Arbeitsspeicher auf dem Heap reserviert und verwaltet. Dazu gehört auch eine automatische Speicherfreigabe durch die sogenannte Garbage Collection (dt. Müllabfuhr) für nicht mehr benötigte Objekte. Der Garbage Collector verfolgt Objektverweise und spürt Objekte auf, die vom Programmcode nicht mehr erreicht werden können, weil sie nicht mehr referenziert werden. Der Garbage Collector arbeitet mit einer gewissen Verzögerung, da die Verfolgung und Kontrolle aller Verweise in Abhängigkeit von der Programmgröße Zeit in Anspruch nimmt. Außerdem wartet der Garbage Collector auf einen Zeitpunkt, an dem das Programm möglichst wenig beschäftigt ist. Nur in Ausnahmefällen kann es durch diese Verzögerung zu Speicherplatzmangel kommen.

Es ist jedoch möglich, in die automatische Garbage Collection einzutreten, um die Speicherfreigabe in bestimmten Fällen zu erzwingen bzw. zu verhindern. Für derartige Aufgaben gibt es die Methoden der Klasse GC. Einige dieser Methoden sind nachfolgend dargestellt:

Methode	Einsatzmöglichkeit und Beispiel
Collect()	Erzwingt eine Garbage Collection. GC.Collect();
GetTotalMemory()	Gibt die ungefähre Anzahl der auf dem Heap reservierten Bytes zurück. Ist das Argument true, wird vorher eine Garbage Collection durchgeführt. Console.WriteLine(GC.GetTotalMemory(true));
KeepAlive()	Stellt sicher, dass der Speicher für das referenzierte Objekt bis zum Aufruf dieser Zeile nicht vom Garbage Collector freigegeben wird. GC.KeepAlive(Variable);

13.17 Strukturen und Klassen

Zwischen Strukturen und Klassen gibt es sehr viele Gemeinsamkeiten. Daten, Methoden zur Verarbeitung und zum Zugriff auf die Daten, Ereignisse und Schnittstellen lassen sich wie bei Klassen zusammenfassen und kapseln. Allerdings gibt es auch wichtige Unterschiede:

- ✓ Strukturen sind Werttypen, Klassen dagegen Verweistypen. Das heißt, Strukturen werden auf dem **Stack** des Rechners angelegt, Klasseninstanzen dagegen auf dem **Heap**. Dadurch kann es sein, dass die Lösung einfacher Aufgaben mithilfe von Strukturen aus Gründen der höheren Programmablaufgeschwindigkeit unter Umständen einer Lösung mithilfe von Klassen vorzuziehen ist. Größere und komplexere Aufgaben mit hohem und wechselndem Speicherbedarf lassen sich besser mit Klassen lösen.
- ✓ Im Unterschied zu Werttypen (z. B. Strukturen) lassen sich Verweistypen (z. B. Klasseninstanzen) nicht durch einfache Zuweisung kopieren, da bei der Zuweisung nur eine Referenz erstellt wird. Manche Verweistypen besitzen zum Kopieren die Methode `Copy()`. In anderen Fällen müssen Sie selbst eine Kopiermethode schreiben, in der eine weitere Instanz des entsprechenden Typs erstellt wird. Die Werte der einzelnen Felder müssen anschließend in die neue Instanz übertragen werden.
- ✓ Der wichtigste Nachteil von Strukturen ist, dass sie **keine Vererbung** und deshalb keinen Polymorphismus kennen. Deshalb ist eine wirklich objektorientierte Entwicklung mit Strukturen nicht möglich.
- ✓ Einmal angelegte Instanzen von Strukturen können durch die Garbage Collection nicht wieder beseitigt werden, da die Garbage Collection nur den Speicher auf dem Heap freigibt.

13.18 Übungen

Übung 1: Enumerationen

Übungsdatei: --**Ergebnisdatei:** Aufzaehlung.sln

1. Erstellen Sie eine Konsolenanwendung, die mithilfe einer Aufzählung mit dem Namen Farben Konstanten für die Farben Rot, Gelb, Grün und Blau erzeugt.
2. Ermitteln Sie mit der Anweisung `Enum.GetNames(typeof(Farben))` die Namen der einzelnen Konstanten und geben Sie diese unter Verwendung einer `foreach`-Schleife auf dem Bildschirm aus.

Übung 2: Arrays und Strukturen

Übungsdatei: Adresse.sln**Ergebnisdatei:** VieleAdressen.sln

1. Erweitern Sie das in diesem Kapitel behandelte Beispiel `Adresse.sln` so, dass es möglich ist, mehrere Adressen einzugeben.
2. Legen Sie mittels einer Konstanten die Anzahl der zu speichernden Adressen auf den Wert 3 fest.
3. Speichern Sie die eingegebenen Adressen in einem Array, das aus Elementen der Struktur `Adresse` besteht.
4. Geben Sie nach der Eingabe alle Adressen zur Kontrolle auf dem Bildschirm aus.

Übung 3: Arrays kopieren

Übungsdatei: --**Ergebnisdatei:** Klonen.sln

1. Weil Arrays Referenztypen sind, können Sie sie nicht einfach durch eine Zuweisung (z. B. `neuesArray = altesArray`) kopieren, denn dabei wird nur eine weitere Referenz auf das bereits vorhandene Array erstellt. Um ein Array wirklich zu kopieren, können Sie z. B. die Methode `Copy()` der Klasse `Array` verwenden. Erstellen Sie zur Demonstration eine Konsolenanwendung.
2. Erzeugen Sie ein Array, füllen Sie es mit Werten und kopieren Sie es mithilfe der Methode `Copy()`.
3. Geben Sie beide Arrays auf dem Bildschirm aus, um zunächst zu zeigen, dass beide Arrays die gleichen Elemente bzw. Werte besitzen.
4. Löschen Sie nun die Werte des Originalarrays mithilfe der Methode `Clear()`. Geben Sie beide Arrays noch einmal zur Kontrolle auf dem Bildschirm aus.

Übung 4: Eine Auflistung verwenden

Übungsdatei: --**Ergebnisdatei:** EineListe.sln

1. Erstellen Sie ein neues Projekt als Windows-Anwendung.
2. Platzieren Sie auf dem Formular eine `ListBox`-, eine `TextBox`-, vier `Button`- und eine `CheckBox`-Komponente. Stellen Sie die Eigenschaft `Appearance` der Checkbox `Sortiert` auf den Wert `Button`, damit die Checkbox wie eine Schaltfläche aussieht.
3. In der Ereignismethode für die Schaltfläche `Hinzufügen` übernehmen Sie die in die Textbox eingegebene Zeichenkette mithilfe der Methode `Items.Add()` in die Listbox.
4. In der Ereignismethode der Schaltfläche `Alle löschen` rufen Sie die Methode `Items.Clear()` der Listbox auf, um die Liste zu löschen.

5. Innerhalb der Ereignismethode der Schaltfläche *Löschen* soll das markierte Element der Listbox mithilfe der Methode `Items.RemoveAt()` entfernt werden. Den Index des markierten Elements erhalten Sie durch die Eigenschaft `SelectedIndex`.
6. Setzen Sie in der Ereignismethode der Checkbox *Sortiert* die Eigenschaft `Sorted` der Listbox auf den Wert `true` bzw. `false`, je nachdem, ob die Schaltfläche der Checkbox betätigt wurde oder nicht. Den Zustand dieser Schaltfläche können Sie über deren Eigenschaft `Checked` (betätigt → `true`, nicht betätigt → `false`) abfragen. Auf diese Weise werden die hinzugefügten Zeichenketten je nach Zustand der Checkbox alphabetisch in die Liste sortiert oder am Ende angefügt.



Übung 5: Ein Parameter-Array nutzen

Übungsdatei: --

Ergebnisdatei: Zufallszahlen.sln

1. Erstellen Sie ein neues Projekt als Windows-Anwendung.
2. Platzieren Sie zwei Button-, eine Label- und eine ListBox-Komponente auf dem Formular.
3. Erzeugen Sie in der Ereignismethode der Schaltfläche *Start* ein Feld aus Zufallszahlen vom Typ `int` mit einer zufälligen Größe. Zufallszahlen können Sie folgendermaßen erzeugen:

```
int anzahl;
Random randomNumbers = new Random();
anzahl = randomNumbers.Next(20); // Zahl >= 0 und Zahl < 20
...
zahl = randomNumbers.Next(1, 100); // Zahl >=1 und Zahl < 100
```

4. Übergeben Sie das dynamische Feld mithilfe eines Parameter-Arrays an eine Methode, die die einzelnen Elemente des Parameter-Arrays in der Listbox darstellt. Wie Sie der Listbox eine Zeile mit einem Eintrag aus einem Array hinzufügen können, ist nachfolgend dargestellt:

```
listBox1.Items.Add(field[index].ToString());
```

5. Mithilfe der Label-Komponente zeigen Sie an, wie viele Parameter (Elemente des Parameter-Arrays) bei jedem Betätigen der Schaltfläche *Start* übergeben wurden.

14 Fehlerbehandlung und Fehlersuche

14.1 Fehlerarten

Während Sie den Programmcode eingeben, Ihr Programm übersetzen oder Ihr Programm testen, können verschiedene Fehler auftreten, die sich in drei Gruppen einteilen lassen:

Syntaxfehler

Syntaxfehler entstehen, wenn Sie sich nicht an die Syntax von C# halten, wenn Sie sich vertippen oder beispielsweise Programmierwörter und Methoden in einem falschen Zusammenhang verwenden.

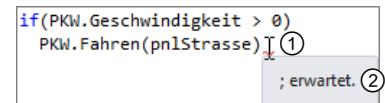
Syntaxfehler vermeiden

Visual Studio hilft Ihnen auf unterschiedliche Weise bei der Vermeidung von Syntaxfehlern:

- ✓ Mithilfe der **Wortvervollständigung**, der **IntelliSense-Auswahllisten**, der **QuickInfo** und der **Parameterlisten** zu Methoden werden Sie direkt bei der Eingabe des Programmcodes unterstützt. Visual Studio verringert den Tippaufwand und reduziert somit das Risiko einer fehlerhaften Programmcodeeingabe.
- ✓ Mit **Code-Ausschnitten** wird automatisch Programmcode erzeugt. Dadurch entstehen korrekte, vollständige Code-Grundgerüste, z. B. für Kontrollstrukturen.

Syntaxfehler erkennen und beheben

Sobald Sie eine Anweisung mit dem Semikolon abschließen oder einen Block mit der geschweiften Klammer } beenden, wird die Zeile im Gesamtzusammenhang geprüft. **Fehlerhafte** Teile einer Programmzeile werden mit einer **roten Wellenlinie** unterstrichen.



- Zeigen Sie mit der Maus auf den mit einer Wellenlinie ① gekennzeichneten Programmcode.
Sie erhalten eine kurze Beschreibung des Fehlers ②.



Sofern möglich, bietet Ihnen Visual Studio Vorschläge zur Korrektur an. Dazu wird in der Regel ein Smarttag mit dem Lampensymbol angezeigt. Dieses finden Sie auch am rechten Rand des Editors.

- Wenn das Smarttag nicht geöffnet ist, klicken Sie auf das Lampensymbol ③, um es zu öffnen. Sie erhalten eine Beschreibung des Problems und einen Hinweis auf mögliche Korrekturen.
- Wählen Sie aus den Korrekturvorschlägen einen Befehl im Smarttag, um die Änderung ④ durchzuführen.

Warnungen

Eine **grüne Wellenlinie** kennzeichnet eine **Warnung**. Das Programm wird jedoch trotzdem übersetzt. Sie haben beispielsweise eine lokale Variable deklariert, die niemals benötigt wird, oder Sie greifen auf ein Laufwerk zu, wobei ein Laufzeitfehler auftreten kann, der bei der Programmierung nicht berücksichtigt wurde. Laufzeitfehler werden im nachfolgenden Abschnitt behandelt.

Das Fenster *Fehlerliste* nutzen

- ✓ Wenn bei dem Versuch, das Projekt mit **[Strg] [F5]** auszuführen, ein Fehler auftritt, erhalten Sie eine Rückmeldung ①. Brechen Sie den Vorgang mit Nein ab. Das Fenster *Fehlerliste* wird dann automatisch geöffnet.
- ✓ Wenn beim Erstellen des Projekts über das Menü *Erstellen* ein Fehler auftritt, wird das Fenster *Fehlerliste* ebenfalls automatisch geöffnet.
- ✓ Sie können das Fenster *Fehlerliste* aber auch jederzeit einblenden:
- Rufen Sie den Menüpunkt *Ansicht - Fehlerliste* auf.
Alternative: **[Strg] [W]** und dann **E**



Im Fenster *Fehlerliste* werden standardmäßig alle Fehler, Warnungen und andere Meldungen aufgelistet.

- ✓ Mit den Schaltflächen ① können Sie einzelne dieser drei Gruppen aus- bzw. einblenden.
- ✓ Mit einem Doppelklick auf einen Eintrag ② gelangen Sie zur entsprechenden Programmzeile im Programmcode.

Fehlerliste				
	1 Fehler	1 Warnung	0 Meldungen	1
Beschreibung	Datei	Zeile	Spalte	Projekt
⚠ 1 Die Variable 'nochEinArray' ist deklariert, wird aber nie verwendet.	Program.cs	10	16	ArrayListRange
✖ 2 Verwendung der nicht zugewiesenen lokalen Variablen 'einArray'	Program.cs	14	9	ArrayListRange

Über die Eigenschaften des Projekts legen Sie im Register *Erstellen* fest, unter welchen Umständen ein Fehler gemeldet bzw. eine Warnung erzeugt wird.

Laufzeitfehler

Nachdem Sie die Syntaxfehler beseitigt haben, lässt sich das Programm ausführen. Visual Studio kompiliert dazu das Programm automatisch, sollte dies notwendig sein. Aber auch **während** der Ausführung eines Programms können Fehler auftreten, sogenannte **Laufzeitfehler**:

- ✓ Beispielsweise soll der Anwender eine Zahl eingeben; stattdessen gibt er aber Text ein. Bei der Konvertierung des eingelesenen Strings in eine Variable vom Typ `int` tritt ein Fehler auf.
- ✓ Sie möchten z. B. eine Datei einlesen, aber die Datei ist nicht vorhanden.

Laufzeitfehler lassen sich voraussehen und im Programmcode so behandeln, dass eine weitere Ausführung der Anwendung möglich ist. Die Behandlung von Fehlern während der Laufzeit wird auch als Ausnahmebehandlung bezeichnet, weil beim Auftreten eines Fehlers von der Laufzeitumgebung (CLR) eine sogenannte **Ausnahme** (engl. **Exception**) erzeugt wird.

Logische Fehler

Logische Fehler können von Visual Studio nicht erkannt werden. Sie entstehen beispielsweise durch fehlerhafte Berechnungsformeln oder Fehler in der Anwendungslogik.

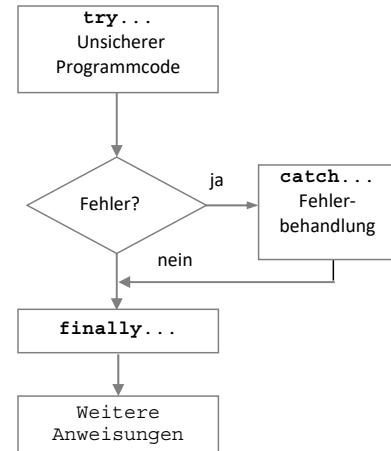


Testen Sie Ihr Programm mit verschiedenen Beispieldaten und überprüfen Sie, ob die Ergebnisse korrekt sind. Nutzen Sie den Debugger (siehe Abschnitt 14.4 ff), um beispielsweise Zwischenergebnisse zu überprüfen.

14.2 Strukturierte Fehlerbehandlung

Die strukturierte Ausnahmebehandlung ermöglicht während der Laufzeit eine stabile und umfassende Fehlerbehandlung, die in einer speziellen Kontrollstruktur mittels der `try`-, `catch`- und `finally`-Anweisungen erfolgt.

- ✓ Mithilfe der `try`-Anweisung wird der **unsichere Programmcode**, d. h. der Programmcode, der möglicherweise zu einem Fehler (oder allgemeiner einer Ausnahme) führen kann, gekapselt.
- ✓ Diesem Codeblock werden dann ein oder mehrere `catch`-Blöcke zur Fehlerbehandlung (oder allgemeiner zur Behandlung der Ausnahme) zugewiesen, die beim Eintreten eines bestimmten Fehlers (z. B. Datei nicht gefunden) abgearbeitet werden sollen.
- ✓ In einem abschließenden `finally`-Block, der in jedem Fall ausgeführt wird (also bei einem Fehler wie auch bei fehlerfreier Ausführung und insbesondere auch dann, wenn in einem `catch`-Block eine Sprunganweisung steht), lassen sich beispielsweise Bereinigungsaufgaben wie das Freigeben von Speicher oder das Schließen von Dateien durchführen.



Syntax für eine `try-catch-finally`-Struktur

- ✓ Eine `try-catch-finally`-Struktur wird mit dem Schlüsselwort `try` begonnen. Danach folgt in geschweiften Klammern `{ } der Programmcode, bei dessen Ausführung eine Ausnahme auftreten kann.`
- ✓ Mit einem oder mehreren `catch`-Blöcken können Ausnahmen gezielt behandelt werden. Dabei lässt sich ein bestimmter Ausnahmetyp herausfiltern, z. B. `ArithmeticException`. Die Basisklasse aller Ausnahmetypen ist die Klasse `Exception`. Durch davon abgeleitete Klassen existiert eine Hierarchie der Ausnahmetypen.

```

try
{
    ...
}
[catch (ExceptionType ExceptionName)
{
    ...
}]
[finally
{
    ...
}]
  
```

- ✓ Auch wenn keine Ausnahme aufgetreten ist, wird abschließend immer der `finally`-Block ausgeführt, bevor die gesamte Ausnahmebehandlungsstruktur verlassen wird.
- ✓ Der `finally`-Block enthält Aktionen, die auf jeden Fall vor dem Verlassen des Blocks (oder des Programms) durchgeführt werden sollten, z. B. das Schließen von Dateien oder das Freigeben von Objekten. Der `finally`-Block kann auch ohne einen `catch`-Block verwendet werden und umgekehrt.
- ✓ Sie können mehrere `try-catch`-Strukturen in einer Methode integrieren.
- ✓ Sie können `try-catch`-Strukturen schachteln, indem Sie in einem `try`-Block eine weitere `try-catch`-Struktur programmieren.

Tipps für die Ausnahmebehandlung

- ✓ Verwenden Sie die Ausnahmebehandlung überall dort, wo von Operatoren oder Methoden Ausnahmen ausgelöst werden können, also z. B. beim Zugriff auf externe Daten- oder Speicherbereiche. Eine nicht behandelte Ausnahme wird jeweils an die aufrufende Methode weitergegeben, wo sie auch behandelt werden kann. Eine unbehandelte Ausnahme führt dazu, dass das Programm mit einer Fehlermeldung abgebrochen wird.
- ✓ Schließen Sie nicht **einzelne** Anweisungen in `try-catch`-Strukturen ein. Dies vermindert die Lesbarkeit des Programms. Zudem können Probleme in der Abfolge von Anweisungen auftauchen, wenn bestimmte Anweisungen aufeinander aufbauen und nur zusammen sinnvoll sind. Verwenden Sie stattdessen mehrere `catch`-Blöcke, die die jeweiligen Ausnahmen abfangen.



Unterstützung bei der Eingabe des Programmcodes

- ▶ Geben Sie in die Programmzeile `try` ein.
- ▶ Betätigen Sie zweimal die -Taste.
 - ✓ Automatisch wird das Grundgerüst für die try-catch-Struktur eingefügt. Einen finally-Block müssen Sie ggf. ergänzen.
 - ✓ Wenn Sie `tryf` statt `try` eingeben, wird Code generiert, der einen `try`- und einen `finally`-Block enthält.

Exception-Hierarchie

Die Ausnahmen (Exceptions) stellen Klassen dar, die hierarchisch von der Klasse `System.Exception` abgeleitet sind. Durch die Vererbung sind viele Exceptions bereits behandelt, wenn Sie die Exception einer Basisklasse in einem `catch`-Block verarbeitet haben. Dies sollten Sie bei der Reihenfolge der `catch`-Blöcke beachten.

Beispiel: `Ausnahme.sln`

In diesem Beispiel wird aufgrund einer Division durch die Zahl 0 eine Ausnahme erzeugt, die abgefangen und auf dem Bildschirm ausgegeben wird.

```
class Program
{
    static void Main(string[] args)
    {
        ① int x = 10;
        L ② int y = 0;
        ③ try
        {
            ④ x /= y;
        }
        ⑤ catch (DivideByZeroException e)
        {
            ⑥ Console.WriteLine(e.Message);
        }
        ⑦ finally
        {
            Console.WriteLine("Der finally-Block wurde aufgerufen.");
        }
    }
}
```

- ① Zwei `int`-Variablen mit den Werten 10 und 0 werden deklariert.
- ② Hier beginnt der `try`-Block.
- ③ Es wird versucht, eine Division (hier absichtlich durch 0) durchzuführen, die eine Ausnahme auslöst.
- ④ Die Ausnahme wird abgefangen und der Variablen `e` zugewiesen. Als Typ wird die Klasse `DivideByZeroException` angegeben. In diesem Fall könnten auch die abgeleiteten Klassen `ArithmetiException` oder `Exception` angegeben werden. Allerdings sollten Sie versuchen, immer die Exception-Klassen zu verwenden, die den möglicherweise auftretenden Exceptions direkt entsprechen. Dadurch können Sie gezielter auf ein Problem reagieren.
- ⑤ Die mit der Ausnahme verbundene Meldung wird auf dem Bildschirm ausgegeben.
- ⑥ Der `finally`-Block wird zum Schluss abgearbeitet und bestätigt dies mit einer Meldung.

14.3 Eigene Ausnahmen erzeugen

Wenn Sie eigene Klassen oder Komponenten erstellen, kann es in bestimmten Situationen auch sinnvoll sein, eigene Ausnahmen zu erzeugen, die Sie dann entsprechend behandeln. Diese können erweiterte Parameterlisten im Konstruktor enthalten und bieten durch ihren Namen die Möglichkeit, individueller auf Fehler zu reagieren.

Syntax zur Auslösung einer Ausnahme

- ✓ Eine Ausnahme wird mithilfe des Schlüsselworts `throw` ausgelöst.
- ✓ In Klammern übergeben Sie die Meldung, die von der Ausnahme geliefert werden soll. Wenn Sie keine Meldung übergeben, wird eine Standardfehlermeldung verwendet.
- ✓ Bei der Exceptionklasse kann es sich, je nach Einsatzgebiet, um eine bereits vorhandene oder um eine von Ihnen selbst deklarierte Exceptionklasse handeln.
- ✓ Wenn die Anweisung `throw` innerhalb eines `catch`-Blocks ohne Angabe eines Exceptionobjekts erfolgt, wird die aktuell im `catch`-Block behandelte Ausnahme erneut ausgelöst.

```
throw [new NameOfException(Messagetext)];
```

Benennung von Exceptionklassen



Zur Benennung Ihrer eigenen Ausnahmen sollten Sie einen Namen verwenden, der mit einer Beschreibung der Funktionalität beginnt und mit dem Wort `Exception` abschließt, um die Klasse als Exceptionklasse zu kennzeichnen (beispielsweise `IndexOutOfBoundsException`).

Die Konstruktoren einer Exceptionklasse

Innerhalb der Exceptionklasse sollten immer drei Konstruktoren deklariert werden, um die Instanzen mit unterschiedlichen Parametern initialisieren zu können.

```
public ExceptionClassName() : base() { }
public ExceptionClassName(string message) : base(message) { }
public ExceptionClassName(string message, Exception ex)
    : base(message, ex)
{ }
```

Beispiel: *EigeneException.sln*

In diesem Projekt wird eine eigene Exceptionklasse `EmptyStringException` erzeugt. Eine weitere Klasse `Test` hat die Aufgabe, zwei Zeichenketten aneinanderzuhängen. Falls eine der beiden übernommenen Zeichenketten leer ist, wird eine Ausnahme vom Typ `EmptyStringException` erzeugt und behandelt.

```
① public class EmptyStringException : ApplicationException
{
    ② public EmptyStringException() : base() { }
    public EmptyStringException(string meldung) : base(meldung) { }
    public EmptyStringException(string meldung, Exception ex)
        : base(meldung, ex)
    {
    }
    class Test
    {
        ③ public string Plus(string txt1, string txt2)
        { }
```

```

④    if (txt1.Length == 0 || txt2.Length == 0)
        throw new EmptyStringException("Mindestens einer der Strings ist leer!");
    else
⑤        return txt1 + txt2;
    }
}
class Program
{
⑥    static void Main(string[] args)
    {
        Test eineInstanz = new Test();
        try
        {
            Console.WriteLine(eineInstanz.Plus("Zusammen", ""));
        }
        catch (EmptyStringException e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            Console.WriteLine("finally-Block ausgeführt.");
        }
    }
}

```

- ① Die Ausnahmeklasse `EmptyStringException` erbt von der Basisklasse `ApplicationException`, die speziell für benutzerdefinierte Ausnahmen zur Verfügung steht.
- ② Alle drei empfohlenen Konstruktoren für keinen, einen oder zwei Parameter werden implementiert. Der Konstruktor übernimmt ggf. die Parameter und ruft damit den entsprechenden Konstruktor der Basisklasse auf.
- ③ Die Methode `Plus()` der Klasse `Test` übernimmt zwei Zeichenketten.
- ④ Wenn mindestens eine der beiden Zeichenketten leer ist, wird eine Exception vom Typ `EmptyStringException` ausgelöst.
- ⑤ Andernfalls werden die beiden Zeichenketten zusammengesetzt und das Ergebnis wird zurückgegeben.
- ⑥ Eine Instanz der Klasse `Test` wird erzeugt und der Objektvariablen `eineInstanz` zugewiesen.
- ⑦ Innerhalb eines `try`-Blocks wird die Methode `Plus()` der Instanz aufgerufen. Durch die Übergabe einer leeren Zeichenkette wird eine Ausnahme ausgelöst.
- ⑧ Die Behandlung der Ausnahme besteht lediglich in der Ausgabe einer Fehlermeldung auf dem Bildschirm.
- ⑨ Im `finally`-Block wird zur Demonstration des Ablaufs eine Meldung ausgegeben.

14.4 Fehler aufspüren und beseitigen

Vor allem bei logischen Programmfehlern, wenn ein Programm zwar läuft, aber falsche Ergebnisse liefert oder unerwünscht reagiert, aber auch bei unbehandelten Ausnahmen können Sie die von Visual Studio angebotenen Hilfsmittel zur Fehlersuche verwenden. Das Aufspüren und Beseitigen solcher Fehler wird auch als Debuggen (Entwirren) bezeichnet. In Visual Studio ist dazu ein sogenannter **Debugger** integriert.

Programm mit Debugger-Unterstützung starten

Der Debugger benötigt spezielle Debug-Informationen, die in einer Datei mit der Erweiterung `*.pdb` im Unterordner `bin\Debug` des Projekts abgelegt werden, um die entsprechenden Passagen des Quelltextes zuordnen und anzeigen zu können. Diese Informationen werden automatisch beim Kompilieren erzeugt, wenn Sie das geöffnete Projekt in Verbindung mit dem Debugger starten:

- Rufen Sie den Menüpunkt *Debuggen - Debugging starten* auf.

Alternativen: Symbol oder **F5**

Programmausführung abbrechen

- Um eine Debug-Sitzung zu beenden, wählen Sie den Menüpunkt *Debuggen - Debugging beenden*.

Alternativen: Symbol oder **F5**

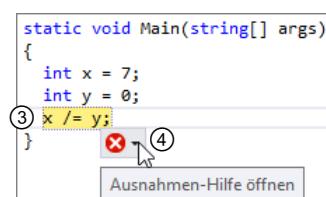
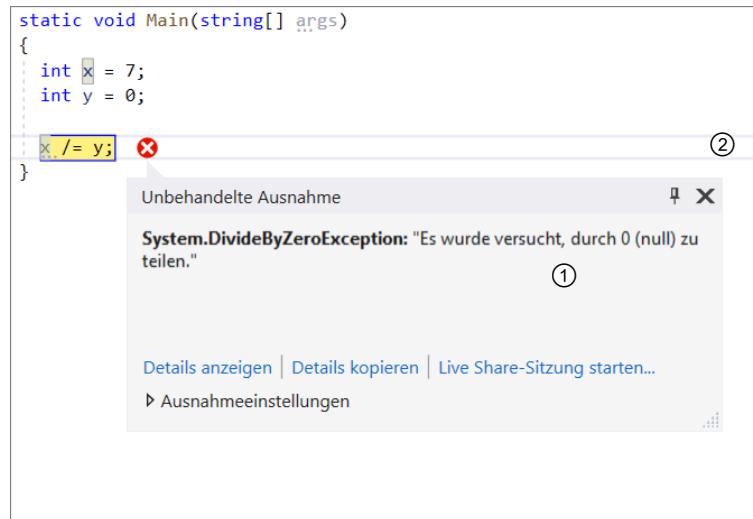
Reaktion auf Ausnahmen

Bei den Standardeinstellungen wird der Programmablauf automatisch angehalten (unterbrochen), wenn eine Ausnahme (Exception) eintritt, die nicht mit einer `catch`-Anweisung abgefangen wurde. So führt beispielsweise die Integer-Division durch null zu einer `DivideByZeroException`.

Über die angezeigten Links ① erhalten Sie in der Hilfe Unterstützung zur Behebung des Fehlers.

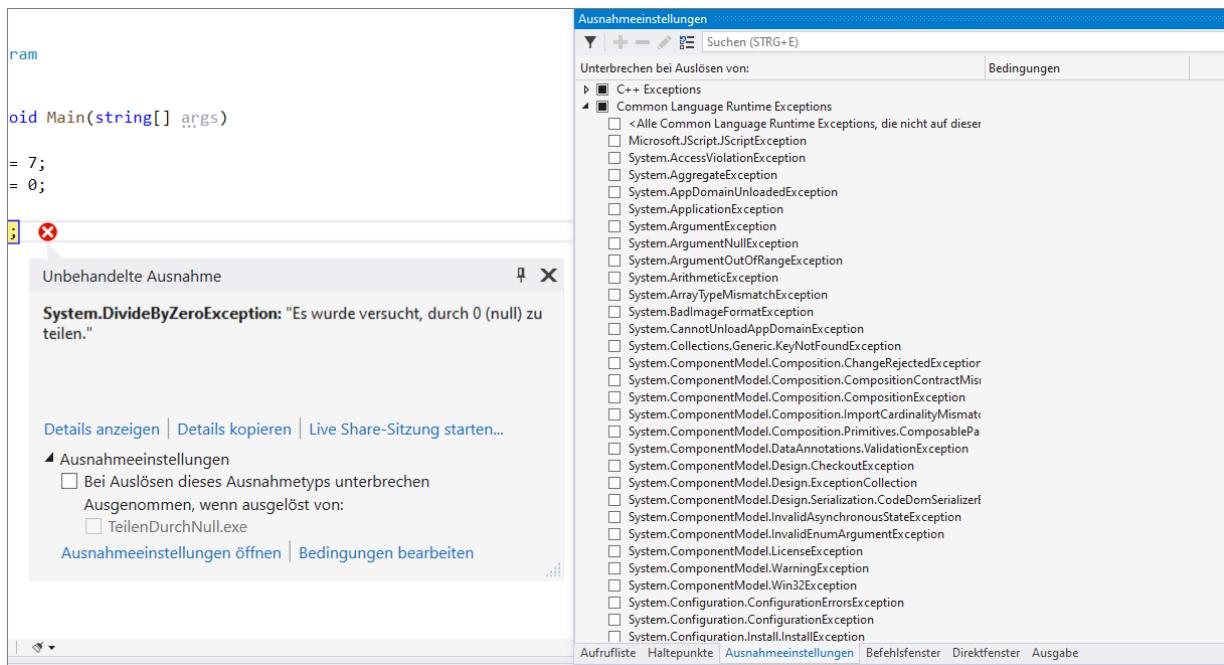
Auch wenn Sie das Meldungsfenster über das Schließfeld ② schließen, bleibt die Anweisung, die ausgeführt werden soll, gelb hinterlegt ③ und wird mit einem gelben Pfeil gekennzeichnet.

Zeigen Sie auf die fehlerhafte Anweisung, wird ein Smarttag ④ angezeigt, über das Sie das Meldungsfenster erneut öffnen können.



„TeilenDurchNull.sln“

Einstellungen zu Ausnahmen



Ausnahmeeinstellungen

- Rufen Sie den Menüpunkt *Debuggen - Fenster - Ausnahmeeinstellungen* auf. Auch das Klicken auf *Ausnahmeeinstellungen* unter Fehlersymbol beim Debuggen öffnet den Einstellungsdialog.
Alternative: **Strg Alt** und dann **E**
Legen Sie hier ggf. das Verhalten für die jeweilige Ausnahme fest.

14.5 Programmablauf kontrollieren

Das Programm anhalten

Ein Programm, das Sie im Debug-Modus (z. B. mit **F5**) gestartet haben, wird nicht nur automatisch, beispielsweise durch un behandelte Exceptions, unterbrochen, Sie können es auch jederzeit anhalten. Der aktuelle Zustand (Variablenwerte etc.) wird „eingefroren“ und die nächste auszuführende Anweisung wird gekennzeichnet. Dabei dürfen aber zu dem Zeitpunkt weder ein Laufzeitfehler noch eine Ausnahme aufgetreten sein.

- Rufen Sie zum gewünschten Zeitpunkt den Menüpunkt *Debuggen - Alle unterbrechen* auf.
Alternativen: Symbol oder **Strg Alt Pause**

Anschließend können Sie den Programmablauf mit dem Menüpunkt *Debuggen - Weiter* bzw. mit dem Symbol fortsetzen oder Sie führen schrittweise die nächsten Anweisungen aus.

Das Unterbrechen gelingt auf diese Weise meist nur bei Programmen, die eine längere Laufzeit haben. Sinnvoll ist das Unterbrechen, wenn sich das Programm beispielsweise in einer Endlosschleife befindet oder Sie ein Programm mit grafischer Oberfläche und einer Interaktion mit dem Anwender vorliegen haben und einen aktuellen Zustand von Variablen auswerten wollen.

Programm schrittweise ausführen

Die Anweisungen des Programms lassen sich in Einzelschritten ausführen. Dies entspricht dem Anhalten nach einzelnen Anweisungen.

Ausführen aller Einzelschritte und die Verzweigung in Methoden, die dann ebenfalls schrittweise ausgeführt werden	► <i>Debuggen - Einzelschritt</i> Alternativen:  oder 
Ausführen aller Einzelschritte: Aufgerufene Methoden werden ohne Unterbrechung abgearbeitet.	► <i>Debuggen - Prozedurschritt</i> Alternativen:  oder 
Die gerade ausgeführte Methode wird ohne Unterbrechung weiter abgearbeitet. Anschließend wird das Programm angehalten.	► <i>Debuggen - Ausführen als Rücksprung</i> Alternativen:  oder  

Der **gelbe Pfeil** am linken Rand des Editorfensters zeigt auf die Anweisung, die als Nächstes ausgeführt wird. In vielen Fällen können Sie den Pfeil innerhalb eines Blocks verschieben und so beispielsweise eine Anweisung noch einmal ausführen. Beachten Sie dabei, dass z. B. Variablenwerte möglicherweise zwischenzeitlich verändert wurden.



14.6 Programme an einer bestimmten Stelle anhalten

Haltepunkte setzen

Um den Programmablauf gezielt zu beobachten, können Sie das Programm mithilfe von Haltepunkten (Breakpoints) an bestimmten Programmcode-Zeilen anhalten.

- Klicken Sie auf den linken Rand in der Zeile ①, vor deren Ausführung das Programm anhalten soll.
Der Programmcode der Zeile wird farbig hinterlegt und die Zeile am Rand mit einem entsprechenden farbigen Kreis gekennzeichnet.

Alternativen: *Debuggen - Haltepunkt umschalten* oder 

```
public bool IsPrime(int value)
{
    if (value <= 1)
        return false;
    else
    {
        if (value > 2)
        {
            for (int i = 2; i <= (value / 2) + 1; i++)
                if (value % i == 0)
                    return false;
        }
        return true;
    }
}
```

Über das Kontextmenü des farbigen Markierungspunkts können Sie spezielle Einstellungen vornehmen. Beispielsweise lassen sich dort Bedingungen definieren, unter denen der Haltepunkt wirksam ist, und Sie können festlegen, dass ein Haltepunkt nur beachtet wird, wenn er zum soundsovielten Mal erreicht wird.

Haltepunkte auflisten

Im Fenster *Haltepunkte* können Sie sich die festgelegten Haltepunkte mit ihren Einstellungen auflisten lassen, um sie beispielsweise zu deaktivieren oder zu löschen. Das Fenster *Haltepunkte* öffnen Sie über den Menüpunkt *Debuggen - Fenster - Haltepunkte*.

Haltepunkte entfernen

- Klicken Sie auf den farbigen Kreis, um den entsprechenden Haltepunkt wieder zu entfernen.
oder Wählen Sie im Kontextmenü des farbigen Kreises, den Eintrag *Haltepunkt löschen*.
Alternativen: *Debuggen - Haltepunkt umschalten* oder 

Über entsprechende Einträge im Kontextmenü lassen sich Haltepunkte auch deaktivieren und wieder aktivieren. Zudem gibt es im Menü *Debuggen* Menüpunkte, um alle Haltepunkte gleichzeitig zu deaktivieren oder zu entfernen.



Tipps für die Bearbeitung von Codezeilen

Lesezeichen setzen

Sie haben die Möglichkeit, an jeder beliebigen Stelle des Programms ein Lesezeichen zu setzen. Dadurch wird das Auffinden bestimmter Programmzeilen wesentlich erleichtert. Die Symbolleiste *Text-Editor* stellt Ihnen Symbole zur Arbeit mit Lesezeichen zur Verfügung.

- ▶ Setzen Sie den Cursor in die Zeile, in der Sie ein Lesezeichen setzen möchten.
 - ▶ Rufen Sie den Menüpunkt *Bearbeiten - Lesezeichen - Lesezeichen umschalten* auf.
- Alternativen: oder und dann

```
public bool IsPrime(int value)
{
    if (value <= 1)
        return false;
    else
    {
        if (value > 2)
    }
```

Die betreffende Zeile wird durch eine Markierung ① am linken Rand gekennzeichnet.

Mit Lesezeichen arbeiten

- ▶ Wählen Sie den Menüpunkt *Bearbeiten - Lesezeichen - Nächstes Lesezeichen*, um zur nächsten Lesezeichenposition zu wechseln.
- Alternativen: oder und dann
- oder* Wählen Sie den Menüpunkt *Bearbeiten - Lesezeichen - Vorheriges Lesezeichen*, um zur vorherigen Lesezeichenposition zu wechseln.
- Alternativen: oder und dann

Mithilfe weiterer Menüpunkte bzw. Symbole können Sie das Wechseln zum nächsten/vorherigen Lesezeichen auf das aktuelle Dokument und den aktuellen Ordner beschränken.

Lesezeichen löschen

- ▶ Setzen Sie den Cursor in die Zeile, in der Sie das gesetzte Lesezeichen löschen wollen.
 - ▶ Rufen Sie den Menüpunkt *Bearbeiten - Lesezeichen - Lesezeichen umschalten* auf.
- Alternativen: oder und dann
- oder* Wählen Sie den Menüpunkt *Bearbeiten - Lesezeichen - Lesezeichen löschen*, um alle Lesezeichen zu löschen.
- Alternativen: oder und dann

14.7 Prüfen und Korrigieren

Werte anzeigen, überwachen und ändern

Wenn Sie ein Programm angehalten haben (Haltemodus), beispielsweise im Einzelschrittmodus oder durch einen Haltepunkt, können Sie die aktuellen Werte von Variablen und Feldern anzeigen lassen und so die korrekte Arbeitsweise Ihres Programms überprüfen. Sie können den angezeigten Wert aber auch verändern und so die Auswirkungen im Programmablauf verfolgen.

- ▶ Zeigen Sie auf den Namen der gewünschten Variablen (hier: `value`) ①.
- In einem Hinweisfenster werden der Name und der Wert der Variablen angezeigt.
- ▶ Klicken Sie auf den angezeigten Wert ②, um den Wert zu ändern.

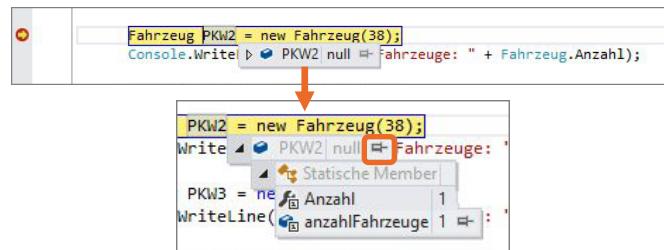
```
if (value > 2)
{
    for (int i = 2; i <= (value / 2) + 1; i++)
        if (value % i == 0)
            return value 8 +
}
```

- Klicken Sie ganz rechts auf das Sticker-Symbol (Pin, Stecknadel), wird das Hinweisfenster verankert und nicht mehr geschlossen, wenn Sie die Maus von der Variablen weg bewegen.

Auch die Eigenschaften und Felder von Objekten können Sie auf diese Weise anzeigen lassen und verändern.

Die Member (Eigenschaften und Felder) einer Klasse oder die Felder einer Struktur werden eingeblendet, sobald Sie auf das Plus-Symbol zeigen.

Wenn Sie das eingeblendete Sticker-Symbol klicken, können Sie die Anzeige der Variablenwerte dauerhaft aktivieren.



Weitere Überwachungsmöglichkeiten

Spezielle Fenster erlauben es Ihnen, Inhalte von Variablen, Feldern und Eigenschaften permanent zu überwachen.

Die Fenster Auto und Lokal

- Rufen Sie den Menüpunkt *Debuggen - Fenster - Auto* bzw. *Lokal* auf, während die Anwendung im Debugger angehalten wurde.

Lokal		
Name	Wert	Typ
args	{string[0]}	string[]
PKW1	{AnzahlAutos.Fahrzeug}	AnzahlAutos.Fahrzeug
Geschwindigkeit	45	int
aktuelleGeschwindigkeit	45	int
↳ Statische Member		
PKW2	null	AnzahlAutos.Fahrzeug
PKW3	null	AnzahlAutos.Fahrzeug

Das Fenster *Auto* zeigt die Variablen im aktuellen Abarbeitungskontext, während das Fenster *Lokal* die lokalen Variablen im aktuellen Gültigkeitsblock zeigt. Auch in diesen Fenstern können Sie Werte während des Debuggens verändern.

Sofern die Referenz *this* im aktuellen Kontext existiert, wird sie in beiden Fenstern ebenfalls angezeigt ①.

Die Fenster Überwachen 1 bis Überwachen 4

Die in diesen Fenstern zu überwachenden Variablen oder Ausdrücke können Sie selbst hinzufügen und entfernen. Gehen Sie dazu folgendermaßen vor (wenn sich der Debugger im Haltemodus befindet):

- Markieren Sie die entsprechende Variable im Quelltext.
- Öffnen Sie mit der rechten Maustaste das Kontextmenü und wählen Sie den Menüpunkt *Überwachung hinzufügen*.
- Über die Untermenüs *Debuggen - Fenster - Überwachen* können Sie zwischen vier Überwachungsfenstern wählen. Die automatisch eingefügten Variablen – wie eben gezeigt – wandern immer in das erste Fenster. Sie können aber Variablen auch per Drag & Drop in die anderen Fenster ziehen und auf diese Weise bestimmte zusammenhängende Werte in einer Ansicht konzentrieren.

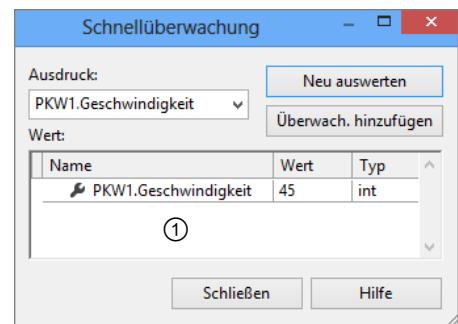
Zum Entfernen der Überwachung einer Variablen oder eines Ausdrucks markieren Sie diese(n) mit der rechten Maustaste und wählen im Kontextmenü den Eintrag *Überwachung löschen*.

Auch innerhalb der Überwachungsfenster lassen sich die Werte der Variablen ändern.

Das Dialogfenster Schnellüberwachung

- ▶ Rufen Sie den Menüpunkt Debuggen - Schnellüberwachung... auf.
 - ✓ Geben Sie den Namen einer Variablen ein, deren Wert Sie einsehen möchten. Die IntelliSense steht Ihnen auch hier zur Verfügung.
 - ✓ Geben Sie den Ausdruck ein, den Sie auswerten möchten.
- ▶ Betätigen Sie die Schaltfläche Neu Auswerten, um den Wert oder das Ergebnis des Ausdrucks im Bereich ① anzuzeigen.

Mit der Schaltfläche Überwach. hinzufügen können Sie die Variable bzw. den Ausdruck zur Überwachung in das Überwachungsfenster übernehmen.



Wenn Sie das Programm anschließend weiter ausführen möchten, müssen Sie das Fenster *Schnellüberwachung* zuvor schließen.

Programmcode bearbeiten

Im Haltemodus lässt sich der Programmcode (begrenzt) verändern und die Änderungen wirken sich, sofern die entsprechende Programmzeile nicht schon abgearbeitet wurde, direkt auf das laufende Programm aus.

14.8 Aufruferinformationen auswerten

Zur Fehlersuche können verschiedene Mechanismen hilfreich sein. Um beispielsweise zu ermitteln, wer eine bestimmte Methode aufgerufen hat, kann in der betreffenden Methode eine entsprechende Ausgabe auf die Konsole erfolgen.

Um diese Funktionalität zu nutzen, wird zuerst der Namespace `System.Runtime.CompilerServices` eingebunden. Innerhalb der Methodensignatur der aufgerufenen Methode werden drei zusätzliche Parameter hinzugefügt. Diese werden mit Vorgabewerten versehen und erhalten eines der Attribute `[CallerMemberName]`, `[CallerFilePath]` und `[CallerLineNumber]` vorangestellt. Darüber ermitteln Sie den Namen der aufrufenden Methode, den Namen der Source-Code-Datei, die diese Methode enthält, und die Zeilennummer innerhalb dieser Datei, in der die Methode aufgerufen wurde.

Ein Attribut wird in eckige Klammern geschrieben und dient als eine Art Markierung des nachfolgenden Programmierelements. Wenn Sie beispielsweise der Methode einen Parameter `[CallerMemberName] string name` hinzufügen, können Sie über den Parameter `name` den Namen der aufrufenden Methode ermitteln.

Beispiel: Aufruferinformationen.sln

Die Methode `ProtokollFunktion()` wird hier von der Methode `Main()` aufgerufen. Dieser Name kann über den Parameter `methodenName` ausgelesen werden. Des Weiteren werden der Name der Source-Datei, hier `Program.cs`, sowie die Zeilennummer, an der die Methode `ProtokollFunktion()` aufgerufen wird (im Beispiel in Zeile 10), ausgegeben.

```
class Program
{
    static void Main(string[] args)
    {
        ①     ProtokollFunktion(1000);
    }

    ②    private static void ProtokollFunktion(int zahl,
                                              [CallerMemberName] string methodenName = "",
                                              [CallerFilePath] string quellPfad = "",
                                              [CallerLineNumber] int zeilenNummer = 0)
    {
        ⑤     Console.WriteLine("Zahl: " + zahl);
        Console.WriteLine("Methode: " + methodenName);
        Console.WriteLine("C#-Datei: " + quellPfad);
        ⑥     Console.WriteLine("Zeilennummer: " + zeilenNummer);
    }
}
```

- ① Diese Zeile kennzeichnet die Zeilennummer, die später als `[CallerLineNumber]` ausgegeben wird, denn hier wird die Methode `ProtokollFunktion()` aufgerufen.
- ② Das Attribut `[CallerMemberName]` kennzeichnet den darauf folgenden Parameter dahin gehend, dass er den Namen der aufrufenden Methode beinhaltet.
- ③ Das nächste Attribut kennzeichnet den Wert des Parameters als vollständigen Dateinamen der Source-Code-Datei, die die aufrufende Methode beinhaltet (in diesem Beispiel liegen aufrufende und aufgerufene Methode in der gleichen Datei).
- ④ Mit dem Attribut `[CallerLineNumber]` wird abschließend der Parameter für die Zeilennummer markiert. Dies ist die Zeilennummer des Aufrufs ①.
- ⑤-⑥ Die Ausgabe der Parameterwerte erfolgt. Die Werte der letzten 3 Parameter werden automatisch befüllt.

14.9 Übungen

Übung 1: Ausnahmen abfangen

Übungsdatei: --

Ergebnisdatei: *ArrayIndexRange.sln*

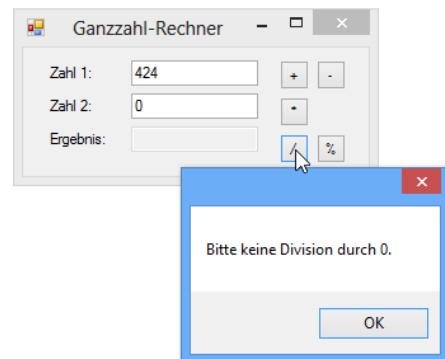
1. Erzeugen Sie in einer neuen Konsolenanwendung ein Array. Provozieren Sie Ausnahmen, indem Sie auf nicht vorhandene Array-Elemente zugreifen.
2. Fangen Sie die Ausnahmen mit einer `try-catch`-Konstruktion ab.
Tipp: Markieren Sie den zu überwachenden Programmcode und wählen Sie im Kontextmenü *Umschließen mit* und anschließend den Codeausschnitt `try`.

Übung 2: Integer-Rechner verbessern

Übungsdatei: *Kap09\IntegerCalcProp\IntegerCalcProp.sln*

Ergebnisdatei: *IntegerCalcExcept.sln*

1. Verändern Sie den Ganzzahl-Rechner *IntegerCalcProp.sln* so, dass Ausnahmen, die durch falsche Eingaben entstehen, abgefangen werden.
2. Benutzen Sie das Validating-Ereignis der TextBox-Komponente, das ausgelöst wird, wenn eine TextBox verlassen wird. Fangen Sie in einer Ereignisroutine für dieses Ereignis alle auftretenden Ausnahmen ab. Stellen Sie sicher, dass bei der Eingabe nur ganze Zahlen akzeptiert werden.
3. Nach dem Anzeigen einer Fehlermeldung soll die entsprechende TextBox gelöscht und eine erneute Eingabe möglich sein.
4. Fangen Sie bei der Division und der Modulo-Division Ausnahmen ab, die durch eine versuchte Division durch null entstehen.

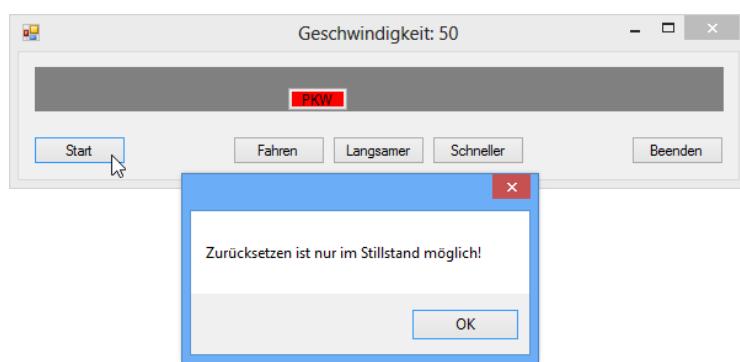


Übung 3: Eigene Ausnahmen erzeugen

Übungsdatei: *Kap10\Autofahren\Autofahren.sln*

Ergebnisdatei: *Autofahren2.sln*

1. Implementieren Sie in die Übung *Autofahren.sln* eine eigene Exception-Klasse. Lösen Sie eine Ausnahme dieses Typs aus, wenn versucht wird, das Fahrzeug an die Startposition durch erneutes Klicken auf *Start* zu setzen, während es fährt ($\text{Geschwindigkeit} > 0$).
2. Verhindern Sie in diesem Fall das Zurücksetzen und geben Sie eine entsprechende Meldung aus.



15 System-, Datei- und Laufwerkszugriffe

15.1 Systemzugriffe über Klassen des .NET Frameworks

Die Klassenbibliothek des .NET Frameworks enthält eine Vielzahl von Klassen, die in einer aufwendigen Hierarchiestruktur mit vielen Ebenen (Namensräumen) organisiert sind. Für die Suche nach geeigneten Klassen und deren Membern verwenden Sie den Objektkatalog und die Hilfe. Bei der Verwendung nutzen Sie die Unterstützung durch die IntelliSense.

Informationen zum verwendeten Computer erhalten

Einige spezielle Klassen des .NET Frameworks geben Ihnen die Möglichkeit, beispielsweise Eigenschaften und Einstellungen der Maus, des Bildschirms, der Tastatur oder des Betriebssystems zu prüfen. Die Ergebnisse können Sie dafür nutzen, dass Ihr Programm bei unterschiedlichen Computerkonfigurationen passend reagiert. Die folgende Übersicht zeigt eine Auswahl.

Sie möchten ...	
Informationen zum Betriebssystem erhalten	Mit <code>System.Environment.OSVersion</code> erhalten Sie ein Objekt der Klasse <code>OperatingSystem</code> , dessen Eigenschaft <code>VersionString</code> beispielsweise die Betriebssystemversion als Text liefert.
den Benutzernamen ermitteln	Über die Eigenschaft <code>Username</code> der Klasse <code>System.Environment</code> erhalten Sie den Benutzernamen, unter dem Sie an dem Computer angemeldet sind.
Bildschirmeinstellungen ermitteln	Z. B. über die Klasse <code>System.Windows.Forms.SystemInformation</code> erhalten Sie mit den Eigenschaften <code>ScreenOrientation</code> , <code>PrimaryMonitorSize</code> bzw. <code>WorkingArea</code> Informationen zur Monitorausrichtung, Bildschirmauflösung oder zur Größe des Arbeitsbereichs.
Einstellungen der Maus abrufen	Ebenfalls in der Klasse <code>System.Windows.Forms.SystemInformation</code> stehen Eigenschaften zur Verfügung, um z. B. zu prüfen, ob die verwendete Maus ein Scrollrad besitzt (<code>MouseWheelPresent</code>) oder wie viele Maustasten zur Verfügung stehen (<code>MouseButtons</code>).
Tastatureinstellungen ermitteln	Über die Eigenschaften <code>KeyboardDelay</code> und <code>KeyboardSpeed</code> der Klasse <code>System.Windows.Forms.SystemInformation</code> ermitteln Sie die Tastaturverzögerung und -wiederholgeschwindigkeit.
Laufwerke und das Systemverzeichnis abrufen	Die Methode <code>GetLogicalDrives()</code> der Klasse <code>System.Environment</code> liefert ein <code>string</code> -Array der Laufwerknamen. Die Eigenschaft <code>SystemDirectory</code> dieser Klasse informiert Sie über das aktuelle Systemverzeichnis.

Die Windows-Zwischenablage nutzen

Mithilfe der Windows-Zwischenablage kann der Anwender Daten von einer Anwendung in eine andere Anwendung kopieren bzw. verschieben. Auch in Ihren Visual C#-Programmen können Sie die Zwischenablage nutzen. Über die Klasse `System.Windows.Forms.Clipboard` stehen Ihnen Eigenschaften und Methoden zur Verfügung, mit denen Sie Daten verschiedener Art an die Zwischenablage übergeben oder aus dieser übernehmen können.

Viele Komponenten des Werkzeugkastens unterstützen die Zwischenablage standardmäßig. Der Anwender des fertigen Programms kann beispielsweise mit der Tastenkombination `Strg` `C` den Text eines Textfelds in die Zwischenablage kopieren oder mit `Strg` `V` Text aus einer anderen Anwendung in ein Textfeld einfügen.

15.2 Klassen für den Dateizugriff

Für die Arbeit mit Dateien stehen im .NET Framework im Namensraum `System.IO` spezielle Klassen bereit, z. B. `FileStream`, `BinaryReader`, `BinaryWriter`, `StreamReader`, `StreamWriter`, `File`, `FileInfo`, `Directory`, `DirectoryInfo` und `Path`. Mit deren Methoden lassen sich Dateioperationen durchführen, beispielsweise Dateien öffnen, Zugriffsarten festlegen, Dateien schließen, oder auch verschiedene Dateieigenschaften bzw. -zustände prüfen. Für den Umgang mit Laufwerken und Verzeichnissen existieren ebenfalls entsprechende Klassen und Methoden.

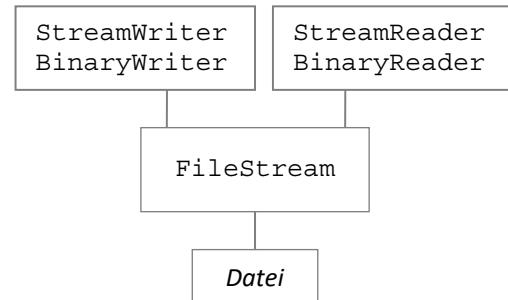
Die Dateioperationen der Klassen des .NET Frameworks basieren auf sogenannten Streams (Datenströmen), mit denen Daten sequenziell als eine Folge von Bytes gelesen und geschrieben werden können. Streams besitzen folgende Merkmale:

- ✓ Ein Datenstrom kann Daten von einer beliebigen Quelle lesen, z. B. von der Tastatur, aus einer Datei, einem String oder einer Netzwerkverbindung.
- ✓ Ein Stream kann Daten in ein beliebiges Ziel schreiben, z. B. auf den Bildschirm, in eine Datei oder einen String.
- ✓ Jeder Datenstrom kann unabhängig von seiner Quelle und von seinem Ziel nach dem gleichen Prinzip und mit denselben Methoden verarbeitet werden.
- ✓ Ein Datenstrom kann die Daten immer nur in eine Richtung senden.

Streams stellen also eine Verallgemeinerung konkreter Bytefolgen dar und erleichtern damit das Lösen von Datenkommunikationsaufgaben, weil betriebssystembedingte Einzelheiten keine Rolle spielen.

Für den Dateizugriff sind folgende Schritte notwendig:

- ✓ Ein `FileStream`-Objekt wird erzeugt und über den Dateinamen (und Pfad) mit der konkreten Datei verbunden.
- ✓ Mit dem `FileStream`-Objekt wird anschließend ein spezielles Reader- oder Writer-Objekt verbunden, das die Methoden für das Lesen aus dem Stream oder das Schreiben in den Stream bereitstellt.



15.3 Mit Laufwerken, Ordnern und Dateien arbeiten

Laufwerke, Ordner und Dateien verwalten

Über den Namespace `System.IO` stehen Ihnen Klassen zur Verfügung, mit deren Hilfe Sie schnell mit den installierten Laufwerken und deren Ordnerstruktur arbeiten können. Außerdem lassen sich Dateien einfach verwalten.

Bei der Zuweisung einer Datei zu einem Stream können Sie den vollständigen Pfad der Datei angeben. Die Datei wird dann an diesem Ort zum Lesen oder Schreiben geöffnet bzw. erzeugt. Wenn Sie nur den Dateinamen angeben, wird die Datei in dem Verzeichnis geöffnet oder erzeugt, in dem sich Ihre Anwendung befindet.

Sie möchten ...	
Laufwerknamen ermitteln	<code>Die Methode System.IO.DriveInfo.GetDrives()</code> liefert eine Collection mit den am Computer des Anwenders verfügbaren Laufwerken.
den aktuellen Ordner ermitteln	<code>System.IO.Directory.GetCurrentDirectory()</code>
prüfen, ob eine Datei/ein Ordner existiert	<code>System.IO.File.Exists()</code> bzw. <code>System.IO.Directory.Exists()</code>
eine Datei kopieren	<code>System.IO.File.Copy()</code>

Sie möchten ...	
eine Datei/einen Ordner verschieben bzw. umbenennen	System.IO.File.Move() bzw. System.IO.Directory.Move()
eine Datei/einen Ordner löschen	System.IO.File.Delete() bzw. System.IO.Directory.Delete()
die in einem Ordner gespeicherten Dateien ermitteln	System.IO.Directory.GetFiles()

Auf spezielle Ordner zugreifen

Über die Methode `System.Environment.GetFolderPath()` können Sie z. B. auf die Ordner *Eigene Dateien*, *Eigene Bilder*, *Eigene Musik* und *Desktop* des Anwenders zugreifen. Geben Sie dazu als Parameter den entsprechenden Wert der Enumeration `System.Environment.SpecialFolder` an. Die nebenstehende Tabelle zeigt einige Beispiele.

Mit dem folgenden Ausdruck wird beispielsweise der Pfad des Desktop-Ordners in der `string`-Variablen `name` gespeichert:

Ordner	Wert der Enumeration
Desktop-Ordner	DesktopDirectory
Arbeitsplatz	MyComputer
Eigene Dateien	MyDocuments
Eigene Musik	MyMusic
Eigene Bilder	MyPictures

```
string name = Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);
```

Der Namensraum `System` ist in dem Beispiel nicht angegeben (`System.Environment...`), da er standardmäßig eingebunden ist.

Die Klassen `FileInfo`, `DirectoryInfo` und `DriveInfo`

Weitere Möglichkeiten zur Arbeit mit Daten, Verzeichnissen und Ordnern erhalten Sie im Namensraum `System.IO` über die Klassen `FileInfo`, `DirectoryInfo` und `DriveInfo`. Beispielsweise können Sie Dateien mit den Methoden `OpenRead()` bzw. `OpenWrite()` ausschließlich zum Lesen bzw. ausschließlich zum Schreiben öffnen.

Um mit diesen Klassen arbeiten zu können, müssen Sie Instanzen für die jeweilige Datei, den jeweiligen Ordner oder das jeweilige Laufwerk erzeugen.

Die Konstante der Enumeration `FileMode`

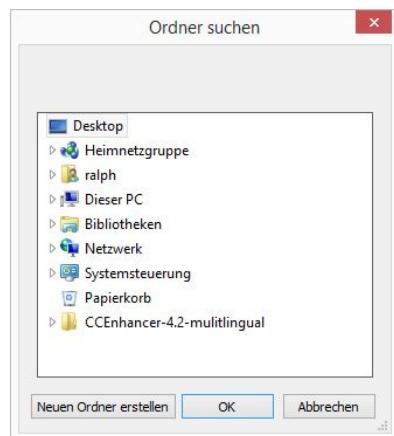
Wenn Sie eine Datei z. B. über ein `FileInfo`-Objekt öffnen, können Sie eine Konstante der Enumeration `System.IO.FileMode` angeben. Damit wird der Modus festgelegt, in dem die Datei geöffnet wird. Auch einige Konstruktoren der Klasse `FileStream` ermöglichen die Verwendung dieser Konstanten.

Konstante	Beschreibung
Append	Die Datei wird geöffnet oder erstellt, wenn sie nicht vorhanden ist. Die Schreibmarke wird an das Dateiende gesetzt. <code> FileMode.Append</code> kann nur für den Schreibzugriff verwendet werden.
Create	Die Datei wird erstellt. Wenn sie bereits vorhanden ist, wird sie überschrieben.
CreateNew	Die Datei wird erstellt. Wenn sie bereits vorhanden ist, wird eine Ausnahme ausgelöst.
Open	Eine Datei wird geöffnet. Wenn sie nicht vorhanden ist, wird eine Ausnahme ausgelöst.
OpenOrCreate	Eine Datei wird geöffnet. Wenn sie nicht vorhanden ist, wird sie erstellt.
Truncate	Eine vorhandene Datei wird geöffnet und ihr Inhalt entfernt. <code> FileMode.Truncate</code> kann nur für den Schreibzugriff verwendet werden.

Fertige Formularkomponenten für die Arbeit mit Ordnern und Dateien nutzen

Wenn Sie aus Formularen auf Laufwerke, Ordner und Dateien zugreifen möchten, können Sie Dialogfenster verwenden, die Sie auch aus anderen Windows-Anwendungen kennen.

Symbol	Name und Beschreibung
	<i>FolderBrowserDialog</i> Dialogfenster zur Auswahl eines Ordners (<i>Ordner Suchen</i>)
	<i>OpenFileDialog</i> Dialogfenster zum Öffnen einer Datei
	<i>SaveFileDialog</i> Dialogfenster zum Speichern einer Datei



Das Dialogfenster „Ordner suchen“

- ▶ Ziehen Sie eine entsprechende Komponente aus dem Werkzeugkasten (Bereich *Dialogfelder*) auf Ihr Formular.
- ▶ Blenden Sie das Dialogfenster an der gewünschten Stelle im Programmcode über den Aufruf der Methode `ShowDialog()` der Dialogkomponente ein, z. B.:


```
folderBrowserDialog1.ShowDialog()
```
- ✓ Die Methode `ShowDialog()` bewirkt, dass das Dialogfenster geschlossen werden muss, bevor der Anwender mit Ihrem Programm weiterarbeiten kann.
- ✓ Die Komponenten besitzen verschiedene Eigenschaften, mit denen Sie Voreinstellungen beispielsweise für den Pfad oder den Dateityp vornehmen können. Andere Eigenschaften geben z. B. den Namen des ausgewählten Ordners oder den Dateinamen einschließlich der Pfadangabe zurück.

15.4 Mit Textdateien arbeiten

Textdateien lesen und schreiben

Textdateien sind zeichenorientierte Dateien, die Sie mithilfe der Klassen `StreamReader` bzw. `StreamWriter` lesen bzw. schreiben. Mithilfe von `System.IO` stehen Ihnen die entsprechenden Methoden zur Verfügung. Die Codierung der Zeichen erfolgt dabei standardmäßig als Unicode im UTF-8-Format. Andere Codierungen (z. B. ASCII) können ebenfalls eingestellt werden. In Abhängigkeit vom Betriebssystem bietet die Klasse `FileStream` Methoden für den synchronen und den asynchronen Dateizugriff an. Beim synchronen Dateizugriff wird die Ausführung des Programms für die Zeit der Datenübertragung unterbrochen. Beim asynchronen Dateizugriff können Sie den Vorgang selbst steuern. In diesem Buch wird ausschließlich der einfachere, synchrone Dateizugriff behandelt.

Beispiel: *Textdateien.sln*

In diesem Beispiel wird eine Textdatei mithilfe der Klassen `StreamWriter` und `StreamReader` geschrieben und anschließend wieder ausgelesen.

```
class Program
{
    static void Main(string[] args)
    {
        ① string dateiname = @"c:\temp\test1.txt";
        ② StreamWriter sw;
```

```

③    sw = new StreamWriter(dateiname, false);
④    for (int i = 1; i < 10; i++)
{
    sw.WriteLine("Dies ist die {0}. Zeile.", i.ToString());
}
⑤    sw.Flush();
⑥    sw.Close();
⑦    StreamReader sr;
⑧    sr = new StreamReader(dateiname);
⑨    string txt = sr.ReadToEnd();
Console.WriteLine(txt);
//Alternative:
⑩   txt = File.ReadAllText(dateiname);
Console.WriteLine(txt);
}
}

```

- ① Für eine Textdatei wird der Dateiname einschließlich des Pfads vereinbart.
- ② Die `StreamWriter`-Instanz `sw` wird erzeugt.
- ③ Mit `new` wird über den Konstruktor die Datei mit dem angegebenen Dateinamen zum Schreiben geöffnet und der Objektvariablen `sw` zugewiesen. Wenn die Datei nicht vorhanden ist, wird sie erstellt. Der zweite Parameter der Methode gibt an, dass die Datei, falls sie vorhanden ist, nicht erweitert, sondern überschrieben wird.
- ④ Neun Textzeilen werden in den Stream geschrieben.
- ⑤ Zur Sicherheit wird die vollständige Ausgabe des Pufferspeichers erzwungen.
- ⑥ Der `StreamWriter` und der zugrunde liegende Stream werden geschlossen.
- ⑦ Die `StreamReader`-Instanz `sr` wird erzeugt.
- ⑧ Mit `new` wird über den Konstruktor die zuvor geschriebene Datei zum Lesen geöffnet und der Objektvariablen `sr` zugewiesen.
- ⑨ Mit der Methode `ReadToEnd()` wird der gesamte Inhalt der Datei in eine Variable vom Typ `string` eingelesen und anschließend auf dem Bildschirm ausgegeben.
- ⑩ Wird das `StreamReader`-Objekt im Programm nicht weiter benötigt, können Sie den Text in einer einzigen Anweisung auslesen. Dazu verwenden Sie die Methode `ReadAllText()`, die Sie über die Klasse `File` erreichen.

Beachten Sie, dass Sie ab Windows Vista nicht an beliebigen Stellen Dateien ablegen können, z. B. unter C:\, da dies durch die Sicherheitsbeschränkungen standardmäßig unterbunden wird.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamWriter`

Eigenschaft/Methode	Beschreibung
<code>BaseStream</code>	Gibt den zugrunde liegenden Stream an.
<code>Encoding</code>	Gibt die Codierung der Ausgabe an.
<code>Close()</code>	Schließt die <code>StreamWriter</code> -Instanz und den zugrunde liegenden Stream. Ein zusätzliches Schließen des Streams ist also nicht unbedingt erforderlich.
<code>Flush()</code>	Löscht den Puffer für die aktuelle <code>StreamWriter</code> -Instanz und veranlasst zuvor die Ausgabe aller im Puffer befindlichen Daten in den Stream.
<code>Write()</code>	Schreibt Daten fortlaufend in den Stream.
<code>WriteLine()</code>	Schließt das Schreiben von Daten mit einem Zeilenumbruch ab.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamReader`

Eigenschaft/Methode	Beschreibung
<code>BaseStream</code>	Gibt den zu Grunde liegenden Stream an.
<code>CurrentEncoding</code>	Ruft die Codierung ab, die von der aktuellen <code>StreamReader</code> -Instanz verwendet wird.
<code>Close()</code>	Schließt die <code>StreamReader</code> -Instanz und den zugrunde liegenden Stream.
<code>EndOfStream</code>	Besitzt den Wert <code>true</code> , wenn das Ende der Datei erreicht ist.
<code>Peek()</code>	Gibt das nächste Zeichen zurück, ohne den Lesezeiger weiterzubewegen. Am Dateiende (kein Zeichen mehr verfügbar) wird -1 zurückgegeben.
<code>Read()</code>	Liest das nächste Zeichen oder den nächsten Block von Zeichen aus dem Stream.
<code>ReadLine()</code>	Liest eine Zeile von Zeichen aus dem Stream.
<code>ReadToEnd()</code>	Liest den Stream bis zum Ende.

Ausnahmebehandlung bei Dateizugriffen

Sie sollten alle Schreib- und Lesezugriffe auf Dateien und Ordner über eine `try-catch-finally`-Konstruktion absichern. Auftretende Ausnahmen (Exceptions) werden im `catch`-Block behandelt. Im `finally`-Block sollten Sie Dateien bzw. Streams schließen, damit sie auch im Fehlerfall geschlossen werden. Auf die Darstellung dieser Absicherung wurde in den meisten Beispielen dieses Kapitels bewusst verzichtet, um sie kurz zu halten.

Ausgewählte Exceptions bei Dateizugriffen

Exception (in <code>System.IO</code>)	Beschreibung
<code>IOException</code>	Basisklasse für Ausnahmen, die beim Zugreifen auf Streams, Dateien und Verzeichnisse ausgelöst werden.
<code>DriveNotFoundException</code>	Wird ausgelöst, wenn ein Laufwerk nicht verfügbar ist.
<code>DirectoryNotFoundException</code>	Wird ausgelöst, wenn ein Verzeichnis nicht gefunden wurde.
<code>FileNotFoundException</code>	Wird ausgelöst, wenn eine angegebene Datei nicht gefunden wurde.
<code>EndOfStreamException</code>	Wird ausgelöst, wenn beim Zugriff versucht wird, das Dateiende zu überschreiten.

15.5 Übungen

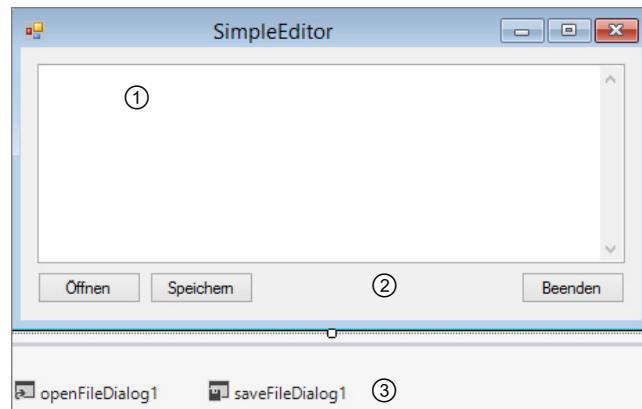
Übung 1: Ein einfacher Text-Editor

Übungsdatei: --

Ergebnisdatei: SimpleEditor.sln

1. Erstellen Sie eine neue Windows-Anwendung und platzieren Sie auf dem Formular eine TextBox-Komponente ①. Setzen Sie über das Smarttag der Komponente die Eigenschaft MultiLine auf den Wert `true` und blenden Sie über das Eigenschaftenfenster beide Symbolleisten (ScrollBars) ein.
2. Platzieren Sie drei Schaltflächen ② auf dem Formular (*Öffnen*, *Speichern* und *Beenden*).
3. Fügen Sie jeweils eine der Komponenten OpenFileDialog und SaveFileDialog ein, die beim Öffnen bzw. Speichern von Textdateien eingesetzt werden können.
4. Die Komponenten werden im Bereich unterhalb des Formulars angezeigt und wie andere Komponenten automatisch benannt ③. Unter der Eigenschaft `Filter` dieser Komponenten tragen Sie die folgende Zeile ein:

```
Textdateien (*.txt) | *.txt
```



5. Ersetzen Sie die Zeichenkette in der Eigenschaft `FileName` durch den Wert `*.txt`.
6. Die Dialoge können Sie wie am hier gezeigten Beispiel des Speichern-Dialogs aufrufen:

```
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
    ...
}
```

Auf diese Weise werden die Aktionen nicht ausgeführt, wenn das Dialogfenster mit *Abbrechen* geschlossen wird. Auf den ausgewählten Dateinamen und Pfad können Sie über die Eigenschaft `FileName` der Dialogbox-Komponente zugreifen.

7. Das Speichern und Laden von Texten realisieren Sie innerhalb der Ereignismethoden der Schaltflächen *Speichern* und *Öffnen* auf einfache Weise mit den Methoden, die Sie unter `System.IO.File` erreichen (vgl. Beispiel `Textdateien.sln` in diesem Kapitel).

Übung 2: Systeminformationen anzeigen

Übungsdatei: --

Ergebnisdatei: SystemInformationen.sln

1. Schreiben Sie eine Windows-Anwendung, in der Sie verschiedene Informationen zum verwendeten Computer ausgeben:
 - ✓ Betriebssystem
 - ✓ Benutzername
 - ✓ Mausrad vorhanden
 - ✓ Bildschirmauflösung
 - ✓ Die Pfade des aktuellen Ordners, des *Desktops*, des *Startmenüs* und der Ordner *Programme* und *Eigene Dateien*.
2. Verwenden Sie die Komponente DataGridView und verhindern Sie über deren Smarttag das Editieren, Hinzufügen und Löschen von Einträgen. Füllen Sie ebenfalls über das Smarttag die Tabelle im ganzen Fenster aus.
3. Blenden Sie die Spaltenköpfe (Eigenschaft RowHeadersVisible) aus und die Zeilenköpfe ein (Eigenschaft ColumnHeadersVisible).
4. Die Tabelle soll beim Laden des Formulars (Ereignis Load) gefüllt werden.
5. Legen Sie mit einer Anweisung die Spaltenanzahl (ColumnCount) auf den Wert 2 fest. Schalten Sie die automatische Spaltenbreite ein, indem Sie die Eigenschaft AutoSizeColumnsMode auf den Wert AllCells setzen.
6. Ermitteln Sie die erforderlichen Informationen und fügen Sie der Collection Rows mit der Methode Add (Beschreibung, Wert) jeweils eine neue Zeile hinzu.

The screenshot shows a Windows application window titled "SystemInformationen". The window contains a single DataGridView control displaying the following data:

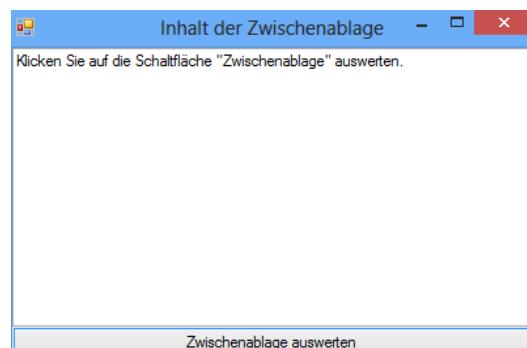
	System Information
Betriebssystem	Microsoft Windows NT 6.2.9200.0
Benutzername	Dirk
Mausrad	True
Maustasten	5
Bildschirmauflösung	1280 x 1024
Aktueller Ordner	E:\Herdt\Kap16\SystemInformationen\SystemInformationen\bin\Debug
Desktop	C:\Users\Dirk\Desktop
Startmenü	C:\Users\Dirk\AppData\Roaming\Microsoft\Windows\Start Menu
Programme	C:\Program Files (x86)
Eigene Dateien	C:\Users\Dirk\Documents

Übung 3: Inhalt der Zwischenablage anzeigen

Übungsdatei: --

Ergebnisdatei: Zwischenablage.sln

1. Schreiben Sie eine Windows-Anwendung, in der Sie einen Button am unteren Formularrand einfügen.
2. Platzieren Sie in dem verbleibenden Bereich eine TextBox (mehrzeilig) und blenden Sie diese über die Eigenschaft Visible aus.
3. Fügen Sie auf die gleiche Weise eine PictureBox-Komponente ein.
4. Beim Laden des Formulars sollen sowohl die TextBox als auch die PictureBox mit der Methode Hide () ausgeblendet werden.
5. Bei einem Klick auf die Schaltfläche *Zwischenablage auswerten* wird geprüft, ob es sich bei dem Inhalt der Zwischenablage um Text oder um ein Bild handelt. Abhängig davon wird der Inhalt der Zwischenablage in die TextBox bzw. die PictureBox kopiert und die entsprechende Komponente mit der Methode Show () eingeblendet.
6. Testen Sie das Programm:
 - ✓ Starten Sie die Anwendung *Zwischenablage*.
 - ✓ Kopieren Sie in einer beliebigen anderen Anwendung einen Text oder ein Bild in die Zwischenablage.
 - ✓ Wechseln Sie zurück zur Anwendung *Zwischenablage* und betätigen Sie die Schaltfläche *Zwischenablage auswerten*.



16 Anwendungen weitergeben

16.1 Voraussetzungen für die Weitergabe

Die Ausführungen dieses Kapitels beziehen sich auf die Weitergabe von Windows-Anwendungen bzw. Windows-Konsolenanwendungen (Deployment genannt).

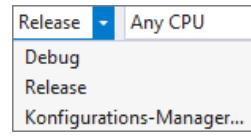
Bevor Sie eine Anwendung an andere Nutzer weitergeben können, müssen bestimmte Voraussetzungen erfüllt sein:

- ✓ Die lizenzirechtlichen Fragen müssen geklärt sein, das heißt, Sie müssen berechtigt sein, die entwickelte Software weiterzugeben. Wenn Sie mithilfe von Visual Studio entwickeln, erwerben Sie diese Lizenz beim Kauf des Entwicklungssystems. Wenn Sie nur mit dem zum freien Download angebotenen .NET Framework entwickeln, beachten Sie die Lizenzbestimmungen, die bei der Installation des Kits angezeigt werden.
- ✓ Es muss sichergestellt sein, dass auf jedem Zielrechner das .NET Framework installiert ist. Sie haben mehrere Möglichkeiten, das .NET Framework weiterzugeben. Bei Verwendung einer Installation über das Web ist die Installationsdatei nur wenige MB groß und lädt die benötigten Daten über das Internet. Führen Sie häufiger Installationen durch, ist eine Offline-Installation oftmals die bessere Wahl. Die Installationsdatei für die jeweils neueste Version des .NET-Frameworks steht auf den Webseiten von Microsoft, unter <https://docs.microsoft.com/de-de/> zum freien Download zur Verfügung. Allerdings können Sie aus Visual Studio heraus auch über die Veröffentlichungsdateien Ihrer Anwendung eine eventuell notwendige Nachinstallation des .NET-Frameworks automatisch veranlassen.
- ✓ Die Anwendung sollte umfangreich getestet sein.
- ✓ Bei der letzten Kompilierung sollten Sie ein sogenanntes **Release** erzeugen. Dadurch werden überflüssige Debug-Informationen entfernt und die Code-Ausführung optimiert.

Ein Release erstellen

Wenn Sie ein Projekt im Debug-Modus starten, erfolgt die Kompilierung in den Unterordner `bin\debug`. Um die kompilierten Daten ohne die Debug-Informationen zur Weitergabe zu speichern, erzeugen Sie ein Release. Das Projekt wird dabei in den Unterordner `bin\Release` kompiliert.

- Öffnen Sie das Projekt, für das Sie ein Release erstellen möchten.
- Wählen Sie im Listenfeld *Projektmappenkonfigurationen* den Eintrag *Release*.
- Rufen Sie den Menüpunkt *Erstellen - Projektmappe neu erstellen* auf, um für die gesamte Projektmappe ein vollkommen neues Release zu erstellen.
 - oder* Rufen Sie den Menüpunkt *Erstellen - Projektmappe erstellen* auf, um nur die Bestandteile zu erstellen, die seit dem letzten Release geändert wurden.
- ✓ Den Ordner, in dem das Release erstellt wird, können Sie in den Projekteigenschaften im Register *Erstellen* ändern.
- ✓ In einem Release sind keine Debugger-Informationen enthalten und der erzeugte Code ist optimiert.



Ein Release erstellen

16.2 Weitergeben durch Kopieren

Einer der Vorteile der .NET-Technologie ist die Möglichkeit, erstellte Assemblies (Anwendungen bzw. Klassenbibliotheken) durch einfaches **Kopieren** zu verbreiten. Deshalb wird in diesem Zusammenhang auch von **XCOPY-Deployment** gesprochen. Die entsprechenden Dateien mit den Erweiterungen `*.exe` bzw. `*.dll` sind selbstbeschreibend und enthalten in der Regel alle notwendigen Informationen. Eintragungen in die Systemregistrierung bzw. Typ- oder Versionsregistrierung sind nicht notwendig. Falls Ihre Anwendungen auf weitere Dateien oder Ressourcen zugreifen, müssen diese natürlich auch kopiert werden. In manchen Fällen ist es ratsam, das ganze Unterverzeichnis *Release* im *bin*-Ordner eines Projekts zu kopieren. In diesem Verzeichnis legt der Compiler die beim Kompilieren entstehenden Dateien sowie weitere von der Anwendung benötigte abhängige Dateien (z. B. weitere Assemblies) ab.

16.3 Anwendungen mit Visual Studio veröffentlichen

Visual Studio 2022 stellt eine integrierte Technik bereit, um Anwendungen auf einfache Weise weitergeben zu können. Insbesondere ist für deren Installation und Ausführung dann sehr wenig Benutzerinteraktion erforderlich. Visual Studio bietet vollständige Unterstützung zur Veröffentlichung und Aktualisierung von den unterschiedlichsten Typen an Anwendungen.

Diese ClickOnce-Bereitstellung genannte Technologie (wobei dieser Bezeichner ClickOnce nicht mehr so präsent verwendet wird wie früher) löst drei Hauptprobleme beim Deployment:

- ✓ Schwierigkeiten beim Aktualisieren von Anwendungen.
Automatische Bereitstellung von Updates. Dabei werden nur die Teile der Anwendung heruntergeladen, die geändert wurden. Anschließend wird die vollständige aktualisierte Anwendung von einem neuen parallelen Ordner aus neu installiert.
- ✓ Auswirkungen auf den Computer des Benutzers bei gemeinsam genutzten Komponenten und unterschiedlichen Versionen der Komponenten.
Dabei bleibt jede Anwendung völlig unabhängig und verursacht keine Konflikte mit anderen Anwendungen.
- ✓ Die Bereitstellung kann die Installation durch Benutzer ohne Administratorberechtigungen gestatten, ohne dass Sicherheitsprobleme entstehen. Es werden nur die Rechte für Codezugriffssicherheit gewährt, die für die Anwendung erforderlich sind.

Wenn Sie den Webpublishing-Assistenten in Visual Studio verwenden, werden alle notwendigen Kopiervorgänge automatisch ausgeführt. Sie können so vorgehen:

- ▶ Öffnen Sie die Projektmappe, in der sich das Projekt befindet, das Sie weitergeben wollen.
- ▶ Markieren Sie das gewünschte Projekt im Projektmappen-Explorer.
- ▶ Rufen Sie den Menüpunkt *Erstellen - [Projektname] veröffentlichen* auf.
Alternativ können Sie im Projektexplorer das Kontextmenü verwenden.
Der Webpublishing-Assistent wird gestartet. Damit kann man in einem vielfältig in den folgenden Schritten zu konfigurierenden Ordner publizieren. Alternativ gibt es noch einen Assistenten, über den Sie den Azure-Dienst von Microsoft (darauf wird im Buch nicht weiter eingegangen) nutzen können.
- ▶ Wenn Sie die Veröffentlichung in einem Ordner gewählt haben, geben Sie im ersten Schritt des Assistenten den Namen des Ordners an, in dem die Daten zusammengestellt werden sollen.
Standardmäßig wird der Ordner *publish* oder *veröffentlichen* als Unterordner des Projektordners verwendet.
Dabei kann sich diese Vorgabe über verschiedene Versionen und Einstellungen von Visual Studio aber auch ändern. Ist der Ordner nicht vorhanden, wird er ggf. neu erzeugt. Sollten Sie keine weiteren Einstellungen vornehmen wollen, können Sie bereit hier mit *Finish* bzw. *Fertig stellen* die Veröffentlichung abschließen.

Wo möchten Sie die Anwendung veröffentlichen?



Veröffentlichungsort für diese Anwendung angeben:

Durchsuchen...

Sie können die Anwendung auf einem FTP-Server oder in einem Dateipfad veröffentlichen.

Beispiele:

- Datenträgerpfad: c:\deploy\myapplication
- Dateifreigabe: \\server\myapplication
- FTP-Server: ftp://ftp.contoso.com/myapplication

[Informationen zum Testen Ihrer Anwendung in Azure](#)

[<< Zurück](#) [Weiter >](#) Fertig stellen [Abbrechen](#)

Zielordner des Veröffentlichungsassistenten

- In dem folgenden Schritt legen Sie fest, ob die Anwendung von einer Website, einem Dateipfad oder von einer CD oder DVD installiert werden soll. Die Auswahl macht dann u. U. jeweils spezifische Zusatzinformationen notwendig. Auch an der Stelle können Sie bereits den Assistenten beenden, wenn die erfassten Angaben ausreichen.

Wie werden Benutzer die Anwendung installieren?



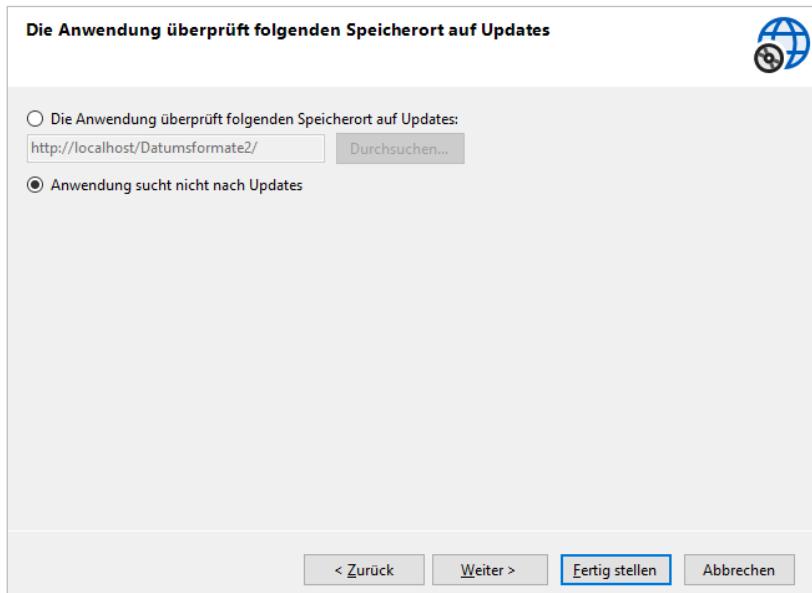
Von einer Website
URL angeben:
 Durchsuchen...

Von UNC-Pfad oder Dateifreigabe
UNC-Pfad angeben:
 Durchsuchen...

Von CD-ROM oder DVD-ROM

[<< Zurück](#) [Weiter >](#) Fertig stellen [Abbrechen](#)

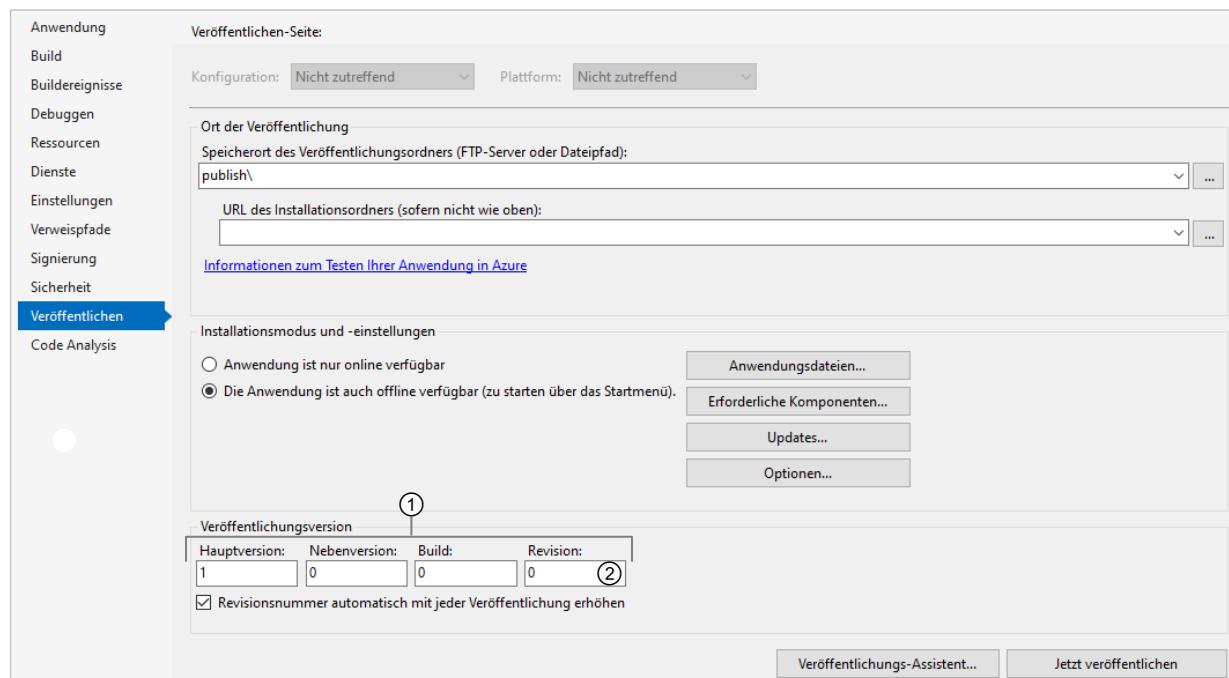
- Sofern Sie mit Klick auf *Weiter* den Assistenten fortsetzen, können Sie im folgenden Schritt Angaben zu Updates festlegen.



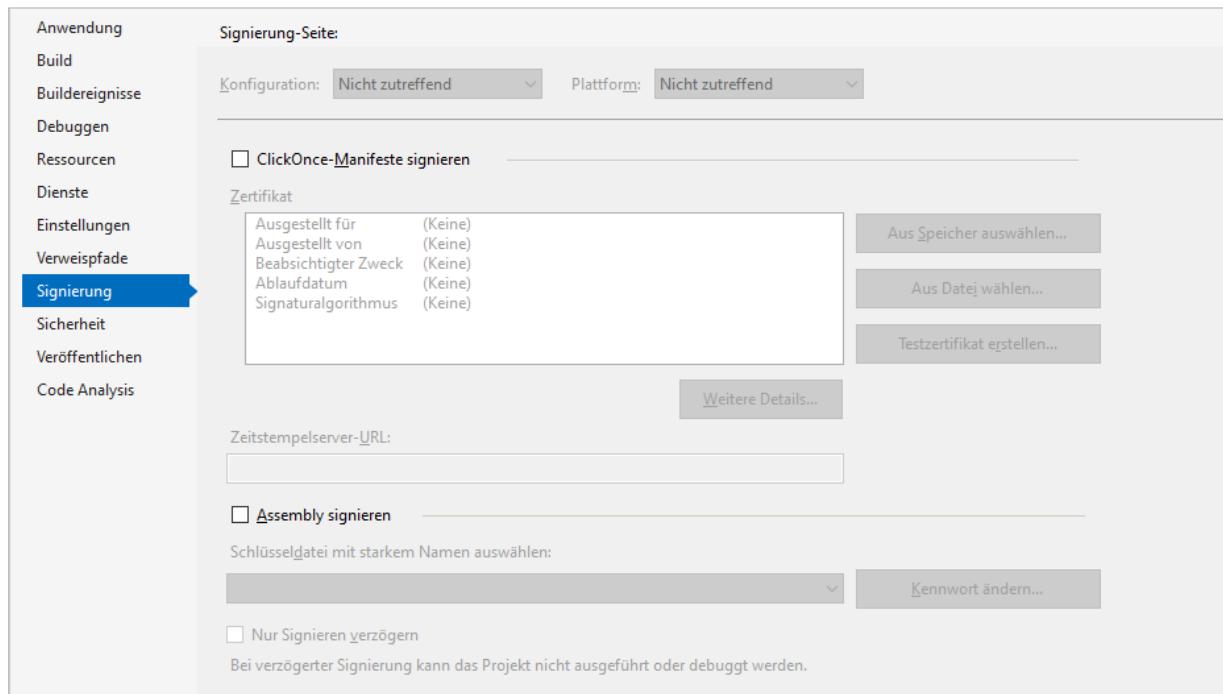
- Im abschließenden Dialog werden noch einmal alle Daten zum Deployment zusammengefasst und Sie können die Veröffentlichung abschließen.

Einstellungen ändern

- Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf den Namen des Projekts und wählen Sie im Kontextmenü den Befehl *Eigenschaften*. Diese Einstellungsmöglichkeiten wurden in Visual Studio 2022 gegenüber Vorgängerversionen massiv verändert und entschlackt. So fehlt etwa in dem nun geöffneten Fenster das Register *Paket*, worüber noch in Visual Studio 2019 viele Einstellungen für die Veröffentlichung festgelegt wurden. Mittlerweile sind diese Informationen erheblich reduziert und in das Register *Veröffentlichen* verlagert worden, wo die Eingaben des Veröffentlichungsassistenten zusammengefasst und erweitert werden.
- In dem Dialog können Sie die unterschiedlichsten Angaben spezifizieren, beispielsweise die Versionsnummer Ihrer Anwendung ①. Standardmäßig wird die letzte Stelle – die Revisionsnummer ② – bei jeder Veröffentlichung erhöht.



Beachten Sie, dass Sie unter Umständen ein Zertifikat für eine Veröffentlichung benötigen. Unter den Projekt-eigenschaften finden Sie im Bereich *Signierung* die Möglichkeit, ein vorhandenes Zertifikat auszuwählen oder ein Testzertifikat zu erstellen.



Eine Applikation sollte vor einer Veröffentlichung mit einem Zertifikat signiert werden

16.4 Übung

Windows-Anwendung weitergeben

Übungsdatei: Kap16\SimpleEditor\SimpleEditor.sln **Ergebnisdatei:** --

1. Öffnen Sie die Projektmappe *SimpleEditor.sln*.
2. Erzeugen Sie mithilfe des Webpublishing-Assistenten ein Setup für die Veröffentlichung auf CD oder DVD ohne Update-Funktionalität. Das Setup soll in einem separaten Ordner mit dem Namen *CDInstall* zusammengestellt werden.

Zeichen			
! (Nicht)	66	Arrays, jagged	173
!=	65	Arrays, mehrdimensionale	172
#endregion	162	Arrays, unregelmäßige	173
#region	162	Arrays, verzweigte	173
% (Modulo-Operator)	62	Array-Variablen	170
& (Und)	66	as	152, 165
.cs	15	Assemblies	9
.NET	6	Assoziativität, Operatoren	62, 64
.NET 6	7	Asynchroner Dateizugriff	214
.NET Core	7	Aufgabenkommentare einfügen	45
.NET-Klassenbibliothek	7	Aufgabenkommentare erstellen	45
.sln	14, 16	Auflistungen, Collections	178
::	127	Auflistungsinitialisierer	180
^ (Exklusiv-Oder)	66	Aufrufhierarchie anzeigen	95
(Oder)	66	Aufzählungstypen, Enumerationen	189
~ (Tilde, Destruktor)	120	Ausdruck, Expression	55, 61
<	65	Ausdrücke überwachen	207
<=	65	Ausgabeparameter	103
==	65	Ausnahme (Exception)	198
=>	115	Ausnahmebehandlung	198, 199, 216
>	65	Ausnahmebehandlung	204
>=	65	Ausnahmen, Reagieren auf	203
		Ausnahmen, selbst definierte	201
		Ausrichtungslinien	33, 35
		Ausrichtungslinien, Einstellungen	34
		Auswahl, einseitige	71
		Auswahl, mehrseitige	76
		Auswahl, mehrstufige	74
		Auswahl, zweiseitige	72
		Azure	220
A			
abstract	149, 151	Codeausschnitte	27, 28, 72, 85, 197, 210
Abstrakte Klassen	159	Codeausschnitte einfügen	28, 29
Accessoren	184	Codeblock	162
Add	178	Codieren	19
Aktivierungsreihenfolge anzeigen	36	Collect	194
Aktivierungsreihenfolge festlegen	36, 37	Collections, Auflistungen	178
Aktuelles Datum	150	Common Intermediate Language (CIL)	8
Anweisungen	23, 47	Bedingungen, Kontrollstrukturen	70
Anweisungsblöcke	47	Bedingungsoperator	73
Anwendung ausführen	25	benannte Parameter	99
Anwendungen weitergeben	219	Benutzeroberfläche	20
Anwendungarten	19	Benutzeroberfläche, grafische	11
App.config	15	Bezeichner	40
Append	213	Bezeichner ändern	55
ArithmeticeException	200	Bezeichner hervorheben	41
Arithmetische Operatoren	61	BigInteger	50
Array, Index	169	Bildschirmeinstellungen ermitteln	211
Array-Elemente	168	BinaryReader	212
Array-Elemente, Anzahl	169	BinarySearch	174
ArrayList	169	BinaryWriter	212
ArrayList	179	Bindung, Operatoren	62
ArrayList	168, 172	bool	50
ArrayList	168	Boolescher Datentyp	50, 52
ArrayList	168	BorderStyle	32
B			
BackColor	32	Boxing	193
base	134, 144	break	77, 171
BaseStream	215, 216	break-Anweisung	84
Basisklasse	132, 133, 134	Breakpoints	205
Bedingungen, Kontrollstrukturen	70	byte	49
Bedingungsoperator	73		
benannte Parameter	99		
Benutzeroberfläche	20		
Benutzeroberfläche, grafische	11		
Bezeichner	40		
Bezeichner ändern	55		
Bezeichner hervorheben	41		
BigInteger	50		
Bildschirmeinstellungen ermitteln	211		
BinaryReader	212		
BinarySearch	174		
BinaryWriter	212		
Bindung, Operatoren	62		
bool	50		
Boolescher Datentyp	50, 52		
BorderStyle	32		
C			
C# 10	6		
Call by reference	98		
Call by value	96, 97		
Capacity	178		
case	76		
case-Anweisung	76		
casting	57		
catch	199		
char	50		
checked	63		
CIL	8		
class	88		
Clear	174, 178		
Click	24		
ClickOnce	220		
Clipboard	211		
Close	215, 216		
CLR	7		
CLS	8, 48		
Code	19		
Codeausschnitte	27, 28, 72, 85, 197, 210		
Codeausschnitte einfügen	28, 29		
Codeblock	162		
Codieren	19		
Collect	194		
Collections, Auflistungen	178		
Common Intermediate Language (CIL)	8		
Common Language Runtime	7		
Common Language Specification			
(CLS)	8		
Common Type System (CTS)	8		
Complex	50		
Concat	105		
Containerelemente	31, 34		
continue-Anweisung	84		
Convert	57, 59, 104		
Copy	104		
Count	178		
Create	213		
CreateNew	213		
CTS	8, 48		
CurrentEncoding	216		

D		else	73	foreach	171
Dateioperationen	212	Empty	52	foreach, Arrays	171
Datentyp DateTime	50	Enabled	32	ForeColor	32
Datentyp Decimal	49	Encoding	215	Formatierungszeichen	59, 60
Datentyp, boolescher (logischer)	50, 52	Endlosschleifen	80	Formular gestalten	33
Datentypen für Zeichen(ketten)	50	EndOfStreamException	216	Formular hinzufügen	37
Datentypen, alphanumerische	50	Entwicklung, sprachübergreifende	6	Formular markieren	32
Datentypen, einfache	48	enum	189	Formular, Größe ändern	33
Datentypen, Gleitkomma-	49	Enum.GetNames	189	Formular, Name ändern	38
Datentypen, Integer-	48	Enum.GetValues	189	Formulardatei, Name ändern	38
Datentypen, nicht kompatibel	57	Enumerationen, Aufzählungstypen	189	Formulare	20, 23
Datentypen, numerische	48, 51	Enumerator	180	Formulare beschriften	20
Datentyp-System	8	Environment	211	Formulare gestalten	19
Datentypumwandlung	58	Ereignis, Standard-	23	Formularstruktur	31
DateTime, Datentyp	50	Ereignisbehandlung aufheben	24	Framework Class Library	7, 49
DateTime.Now	150	Ereignismethode	23	Freigabeversion	125
Datum, aktuelles	150	Ereignismethode entfernen	25	Funktionalität, Klassen	88, 92, 112
Debuggen, Debugger	202, 203	Ereignisse	22, 23	Fußgesteuerte Schleifen	79
Debugger beenden	17	Ereignisse behandeln	23		
Debugger starten	17	Erweiterungsmethoden	107		
Debug-Modus	30, 204	Escape-Sequenzen	52		
decimal	50	Exception	198, 200	G	
default	77	Exceptionklassen benennen	201	Garbage Collection	89, 120, 194
Deployment	219	Exceptions	216	Generate From Usage	27
Designer	19	explicit	154	Generics	182
DesktopDirectory	213	Expression, Ausdruck	55	Geschachtelte if-Anweisungen	75
Destruktoren	116, 119			get	113, 184
Directory	212	F		GetEnumerator	180
DirectoryInfo	212, 213	Fallauswahl	70, 76	GetLength	174
DirectoryNotFoundException	216	false	50	GetLowerBound	104, 173
Dispose	120	FCL	7	GetTotalMemory	194
DivideByZeroException	216	Fehler aufspüren	202	GetType	135
DivideByZeroException	200, 203	Fehler beseitigen	202	GetUpperBound	104, 173
do	81	Fehler, Laufzeit-	198	global	127
DoEvents()	142	Fehler, logische	198	goto-Anweisung	84
Dokumentation erstellen	45	Fehler, Syntax-	197	Grafische Benutzeroberfläche	11
Dokumentgliederung	31	Fehlerliste	198	Gruppieren, Steuerelemente	31
double	49	Felder	87, 112	Gültigkeitsbereich	52
do-while-Anweisung	81	Felder kapseln	113	Gültigkeitsbereich von Variablen	52
DriveInfo	213	Felder, Arrays	168		
		FIFO-Prinzip	185	H	
		File	212	Haltemodus	206
E		FileInfo	212, 213	Haltepunkte	205
Editor	24	FileMode	213	Hashcode	188
Eigenschaften	112, 117	FileNotFoundException	216	Hash-Tabelle, Hashtable	188
OOP	87	FileStream	212	Heap	193
Eigenschaften ändern	32	finally	199		
Eigenschaften festlegen	32	float	49	I	
Eigenschaften, abstrakte	149	Flush	215	IDE	10
Eigenschaften, ausgewählte	32	FolderBrowserDialog	214	Identifier	40
Einfachvererbung	132	Font	32	IDisposable	120
Einzelschritt	205	for	82		
		for-Anweisung	82, 83		

I				M	
IEnumerator	180	Klassen, abgeleitete	132	Main()	177
if	70, 71, 73	Klassen, abstrakte	149, 159	Managed Code	9
if-Anweisung	71	Klassen, Funktionalität	88, 92, 112	Manifest	9
if-Anweisung, geschachtelte	75	Klassen, Informationen zu	129	Margin	34
if-else-Anweisung	73	Klassen, Programmcode aufteilen	127	Markieren, Formular	32
implicit	154	Klassen, statische	123	Markieren, Steuerelemente	31
Index eines Arrays	169	Klassenbibliothek	7	Maus-Einstellungen abrufen	211
Index, fehlerhafter	169	Klassendatei hinzufügen	91	Mehrseitige Auswahl	76
Indexer	183, 188	Klassen-Designer	136	Mehrseitige Verzweigung	76
IndexOf	105	Klassendiagramme	136, 137	Member	88
Individuelle Iteratoren	181	Klassenelemente	121	Member anzeigen	139
Insert	104, 178	Klassenhierarchie	134, 135, 136	Member überladen	148
Instanzen	88	Klasseninstanzen erzeugen	89	Member überschreiben	145
Instanzen erzeugen	89	Klassenstruktur	129	Member verbergen	144
int	49	Kommentar, einzeiliger	44	Member, statische	121
Integrated Development Environment (IDE)	10	Kommentarblock	44	Memberliste	26, 27
IntelliSense	55	Kommentare erstellen	43, 44	Membervariablen, Zugriff auf	191
IntelliSense nutzen	25	Kommentare schachteln	44	MessageBox . Show	104
IntelliSense, Vorteile	25	Kommentare, Aufgaben-	45	Metadaten	9
interface	160	Kommentare, Dokumentations-	45	Methode, optionale Parameter	100
Interfaces deklarieren	160	Kompatibilität	143	Methoden	87, 88, 93, 112
Interfaces implementieren	160	Komponenten, visuelle	20	Methoden anwenden	93, 101
Intermediate Language	8	Konsolenanwendung	11, 19	Methoden erstellen	92, 101
internal	88, 89	Konsolenanwendung ausführen	30	Methoden generieren	95
IO	212	Konsolenanwendung codieren	29	Methoden mit Parametern	96, 97
IOException	216	Konsolenanwendung erzeugen	29	Methoden mit Rückgabewert	101
is	152, 165	Konsolenanwendung, Aufbau	43	Methoden überladen	105
IsArray	174	Konstanten verwenden	60	Methoden, abstrakte	149
Iterationen	70, 78, 108	Konstruktor erstellen	116	Methoden, benannte Parameter	99
Iterator	180	Konstruktor nutzen	118	Methoden, partielle	127
Iterator, individueller	181	Konstruktoren	116, 119, 123, 134	Methoden, vordefinierte	104
		Kontextschlüsselwörter	41, 42	Methodenstub	96
		Kontrollstrukturen	70, 71	Korrekturvorschläge	197
		Kopfgesteuerte Schleifen	79, 82	Microsoft Intermediate Language (MSIL)	8
		Kontrollstrukturen	169, 174	Modifizierer	88, 89
		Kontrollstrukturen	206	Modulo-Operator	62
		LIFO-Prinzip	187	MouseButtons	211
		LINQ	54	MouseDown	24
		Literele	51	MouseHover	24
		Location	32	MouseMove	24
		Locked	33, 36	MouseUp	24
		Logische Fehler	198	MouseWheelPresent	211
		Logischer Datentyp	50	MSIL	8
		Lokale Variablen	52	Multithreading	86
		long	49	MyComputer	213
		Loops (Schleifen)	70	MyDocuments	213
				MyMusic	213
				MyPictures	213
				Nachrichten zwischen Objekten	87
				Name, Eigenschaft	21, 33

Namen	40	Parameter, benannte	99	Projektvorlagen	11
Namen, Regeln	21	Parameter, optionale	100	protected	89
Namensräume	123, 129	Parameter, Übergabe als Verweis	98	protected internal	89
Namensräume einrichten	123	Parameter, Übergabe als Wert	97	public	89
Namensräume, Verweis entfernen	125	Parameter-Arrays	174		
Namensräume, Verweise anzeigen	124	Parameter-Arrays,			
Namensräume, Verweise hinzufügen	124	Konflikt beim Überladen	175	Q	
nameof	68	Parameterliste, aktuelle	97		
namespace	124	Parameterliste, formale	96	Qualifizierung	126
Neuen Typ generieren	27	Parameterübergabe als Verweis	96	Queue, Warteschlange	185
Neues Projekt konfigurieren	13	Parameterübergabe als Wert	96	QuickInfo	28, 97
new	89, 116, 165	Parameterübergabe, Art	96		
new, Modifizierer	144	Parameterübergabe, Main-Methode	177	R	
null	89	params	174	Rand, äußerer	34
		Parse	57	Rand, innerer	34
		partial	127	Random.Next	104
		Partielle Methoden	127	Rank	173
		Pascal-Schreibweise	41	Raster aktivieren	35
Object Management Group (OMG)	137	Path	212	Raster verwenden	35
Objektbrowser	129	Peek	216	Read	216
Objekte	88	Platzhalter	27, 28, 59	ReadLine	216
Objekte erzeugen	89	Polymorphismus	143, 159	readonly	88
Objektinitialisierer	118	PrimaryMonitorSize	211	ReadToEnd	216
Objektorientierte Programmierung	87	private	89	Redeklaration	144
Objektreferenz this	93	Programm	47	ref	96, 103
OMG	137	Programm anhalten	204	Referenztyp	89
Open	213	Programm schrittweise ausführen	205	Referenzvariable	89
OpenFileDialog	214	Programmablaufplan (PAP)	74	Referenzvariablen	91, 92, 170
OpenOrCreate	213	Programmargumente	177	Reguläre String-Literale	51
OperatingSystem	211	Programmaufbau	42	Reihenfolge, Aktivierungs-	36
operator	154, 156	Programmausführung abbrechen	203	Rekursion	108, 109
Operatoren	61	Programmcode bearbeiten	27	Release erstellen	219
Operatoren überladen	155	Programmcode ein- bzw.		Remove	104, 178
Operatoren, arithmetische	61	ausrücken	47, 48	RemoveAt	178
Operatoren, binäre	61	Programmcode generieren lassen	27	Replace	105
Operatoren, unäre	64	Programmcode kommentieren	43	Resize	24
Operatoren, Vergleichs-	65	Programmcode-Editor	24, 94	return	93, 101
Operatoren, Verkettungs-	64	Programmierung,		Reverse	174
Operatoren, Vorzeichen-	64	ereignisgesteuerte	22	Rundungsfehler	49
Operatoren, Zuweisungs-	67	Programmierung, objektorientierte	87		
Optionale Parameter, Konflikt beim Überladen	175	Programmierung, strukturierte	87	S	
Ordinalzahl	189	Projekt	11		
OSVersion	211	Projekt anlegen	19	SaveFileDialog	214
out	103	Projekt ausführen	17	sbyte	49
Overflow	63	Projekt erzeugen	12	Schleifen	70, 78
Overloading	105	Projekt öffnen	16	Schleifen verwenden	78, 79
override	146, 151, 164	Projekt speichern	12, 15	Schleifen, foreach-	171
		Projekt, Start-	11	Schleifen, fußgesteuerte	79, 81
		Projektausführung beenden	17	Schleifen, kopfgesteuerte	79, 82
		Projektmappe schließen	16	Schleifen, zählergesteuerte	82
		Projektmappen	11	Schleifenrumpf	79
		Projektmappenname	14	Schleifensteuerung	79, 84
Padding	34	Projektnamen	14	Schlüssel	188
PAP	74	Projekttypen	11, 13	Schlüsselwörter	40, 41
Parameter	96, 177				

Schlüsselwörter, Übersicht	41	Steuerelemente verschieben	33	TrimStart	104
Schnellüberwachung	208	Steuerelemente zueinander		true	50
Schnittstellen	159	ausrichten	34	Truncate	213
Schnittstellen deklarieren	160	Steuerelemente, Größe ändern	33	try	199
Schnittstellen implementieren	160	Steuerelemente, Hierarchie	31	Try-Catch-Finally	216
Schnittstellenmember explizit implementieren	161	Steuerelemente, mehrere markieren	32	typeof	189
Schnittstellenmember verdecken	164	StreamReader	212, 214	Typkompatibilität	57, 143
Schnittstellen-Zuweisungskompatibilität	159	Streams	212	Typkonversion	57
sealed	133	StreamWriter	212, 214	Typkonvertierung	165
Selektor	76	string	50	Typparameter	182
set	113, 184	String.Empty	52	Typprüfung	152, 165
SetValue	174	String-Literale, reguläre	51, 52	Typsicherheit bei Auflistungen	182
short	49	String-Literale, Verbatim-	51, 52	Typumwandlung	152, 154, 165
Shortcut	28	struct	191		
ShowDialog	214	Struktogramm	74		
Signierung	223	Strukturen	190, 194	Überladen	105, 148
Size	33	Strukturierte Fehlerbehandlung	199	Überladen von Operatoren	155
SnapLine (s. Ausrichtungslinien)	33	Subklasse	132	Überlauf	63
Snippets	27, 28	Substring	105	Überlaufprüfung	63
Snippets einfügen	28	Superklasse	132	Überschreiben	146
Sort	174	switch	76	Überwachung	207, 208
SpecialFolder	213	switch-Anweisung	76	Überwachungsfenster	207
Speicherverwaltung	193	Symbole	5	uint	49
Sprunganweisung	84	Synchroner Dateizugriff	214	ulong	49
Sprungmarke	84	Syntax	40	UML	137
Stack	109	Syntaxfehler	197	UML-Klassendiagramme	137
Stack, Stapel	187, 193	System.Collections.-		Unäre Operatoren	64
Standardereignis	23	Generic	183	Unboxing	193
Standardkonstruktor	134	System.IO	212	unchecked	63
Standardprojektordner	14	SystemInformation	211	Unerreichbarer Code	101
Standardprojektordner ändern	15			Unicode	50, 214
Stapel, Stack	187, 193			Unified Modeling Language (UML)	137
Stapelspeicher	109			Unregelmäßige Arrays	173
Startformular festlegen	37	TabIndex	33	ushort	49
Startklasse, Aufbau	43	TabStop	33	using	120
Startobjekt	90	Tastatureinstellungen ermitteln	211	UTF-8-Format	214
Startprojekt	11	Testzertifikat	223		
static	121, 123	Text, Eigenschaft	21, 33		
Statische Klassen	123	TextAlign	33		
Steuerelemente	23, 24	TextChanged	24		
Steuerelemente ausrichten	33, 35	Textdateien	214		
Steuerelemente benennen	21	this	93, 183		
Steuerelemente beschriften	20	throw	201		
Steuerelemente einfügen	20	Tilde (~), Destruktor	120		
Steuerelemente gruppieren	31	ToDateTime	59		
Steuerelemente in den Hinter-/Vordergrund setzen	36	TODO	45		
Steuerelemente löschen	22	ToLower	104, 105		
Steuerelemente markieren	31	Toolbox	19		
Steuerelemente positionieren	34, 35	Tools und Features abrufen	136		
Steuerelemente sperren/entsperren	36	ToString	57, 104, 135		
Steuerelemente umbenennen	32	ToUpper	104, 105		
		Trim	104		
		TrimEnd	104		

Verknüpfung	28	Werttypen	193	X
VersionString	211	Wertzuweisungen	55	
Verweistypen	193	while	79	XAML
Verzweigte Arrays	173	while-Anweisung	79	XCOPY-Deployment
Verzweigungen	70	Wiederholungen	70	XML-Dokumentation
Verzweigungen, mehrseitige	76	Wiederholungen, bedingte	70	
virtual	146, 150	Wiederholungen, fußgesteuerte	70	Y
Visible	33	Wiederholungen, kopfgesteuerte	70	yield
Visual C#	4, 6	Wiederholungen,		
Visual Studio	6	zählergesteuerte	71, 82	
Visual Studio-Installer	136	Windows Forms	19	Z
void	93	Windows Forms App (.NET Core)	13	
Vorzeichenoperatoren	64	Windows Forms App (.NET Framework)	13	Zahlen, große
W				
Warnung	197	Windows Forms-Designer	15, 20, 24	Zahlen, komplexe
Warteschlange, Queue	185	Windows-Anwendung	11, 13, 19	Zeichen
Wellenlinie, grüne	197	Windows-Anwendung erzeugen	20	Zeichenkettenmethoden
Wellenlinie, rote	197	Windowsanwendung, Aufbau	43	Zertifikat
Werkzeugkasten	19	Windows-Anwendungen		Zugriffsmodifizierer
Werte ändern	206	weitergeben	219	Zuletzt verwendete Projektvorlagen
Werte anzeigen	206	Windows-Zwischenablage	211	92, 170
Werte überwachen	206	WorkingArea	211	Zuweisung
Wertemengen	189	Wort vervollständigen	26	Zuweisungskompatibilität
		Write	215	Zuweisungsoperatoren
		WriteLine	215	55, 67
				Zwischenablage
				211
				Zwischensprache
				8

Impressum

Matchcode: VCSPNET_2022

Autor: Ralph Steyer

Produziert im HERDT-Digitaldruck

1. Ausgabe, Juni 2022

HERDT-Verlag für Bildungsmedien GmbH
Am Kuemmerling 19
55294 Bodenheim
Internet: www.herdt.com
E-Mail: info@herdt.com

© HERDT-Verlag für Bildungsmedien GmbH, Bodenheim

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlags reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit großer Sorgfalt erstellt und geprüft. Trotzdem können Fehler nicht vollkommen ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Wenn nicht explizit an anderer Stelle des Werkes aufgeführt, liegen die Copyrights an allen Screenshots beim HERDT-Verlag. Sollte es trotz intensiver Recherche nicht gelungen sein, alle weiteren Rechteinhaber der verwendeten Quellen und Abbildungen zu finden, bitten wir um kurze Nachricht an die Redaktion.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Die in diesem Buch und in den abgebildeten bzw. zum Download angebotenen Dateien genannten Personen und Organisationen, Adress- und Telekommunikationsangaben, Bankverbindungen etc. sind frei erfunden. Eventuelle Übereinstimmungen oder Ähnlichkeiten sind unbeabsichtigt und rein zufällig.

Die Bildungsmedien des HERDT-Verlags enthalten Verweise auf Webseiten Dritter. Diese Webseiten unterliegen der Haftung der jeweiligen Betreiber, wir haben keinerlei Einfluss auf die Gestaltung und die Inhalte dieser Webseiten. Bei der Bucherstellung haben wir die fremden Inhalte daraufhin überprüft, ob etwaige Rechtsverstöße bestehen. Zu diesem Zeitpunkt waren keine Rechtsverstöße ersichtlich. Wir werden bei Kenntnis von Rechtsverstößen jedoch umgehend die entsprechenden Internetadressen aus dem Buch entfernen.

Die in den Bildungsmedien des HERDT-Verlags vorhandenen Internetadressen, Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen waren zum Zeitpunkt der Erstellung der jeweiligen Produkte aktuell und gültig. Sollten Sie die Webseiten nicht mehr unter den angegebenen Adressen finden, sind diese eventuell inzwischen komplett aus dem Internet genommen worden oder unter einer neuen Adresse zu finden. Sollten im vorliegenden Produkt vorhandene Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen nicht mehr der beschriebenen Software entsprechen, hat der Hersteller der jeweiligen Software nach Drucklegung Änderungen vorgenommen oder vorhandene Funktionen geändert oder entfernt.