

/six-fap/

→ 4., aktualisierte und erweiterte Auflage



Hanspeter Mössenböck

Kompaktkurs C# 5.0

dpunkt.verlag



Hanspeter Mössenböck ist Professor für Informatik an der Universität Linz und Leiter des Instituts für Systemsoftware. Er beschäftigt sich vor allem mit Programmiersprachen, Compilern und Systemsoftware.

Als ehemaliger Mitarbeiter von Prof. Niklaus Wirth an der ETH Zürich war er Mitglied des Oberon-Teams, in dem ein Pascal-Nachfolger samt innovativem Betriebssystem entwickelt wurde. Ferner ist er Autor des Compiler-Generators Coco/R, der heute weltweit als Public-Domain-Software eingesetzt wird. Neben einem Forschungsaufenthalt bei Sun Microsystems in Kalifornien hatte er Gastprofessuren in Oxford und Budapest inne. Er ist Verfasser der Bücher »Sprechen Sie Java?« und »Objektorientierte Programmierung in Oberon-2« sowie Mitverfasser der Bücher »Die .NET-Technologie« und »Ein Compiler-Generator für Mikrocomputer«.

dpunkt.lehrbuch

Bücher und Teachware für die moderne Informatikausbildung

Berater für die dpunkt.lehrbücher sind:

Prof. Dr. Gerti Kappel, E-Mail: gerti@big.tuwien.ac.at

Prof. Dr. Ralf Steinmetz, E-Mail: Ralf.Steinmetz@kom.tu-darmstadt.de

Prof. Dr. Martina Zitterbart, E-Mail: zit@telematik.informatik.uni-karlsruhe.de

Papier
plus⁺
PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei [dpunkt.plus⁺](http://dpunkt.plus+):

www.dpunkt.de/plus

Hanspeter Mössenböck

Kompaktkurs C# 5.0

4., aktualisierte und erweiterte Auflage



dpunkt.verlag

Prof. Dr. Hanspeter Mössenböck
Johannes Kepler Universität Linz
Institut für Systemsoftware
Altenbergerstraße 69 · A-4040 Linz
E-Mail: hanspeter.moessenboeck@jku.at
<http://ssw.jku.at>

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: FrameMaker-Dateien vom Autor
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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

ISBN
Buch 978-3-86490-227-7
PDF 978-3-86491-604-5
ePub 978-3-86491-605-2

4., aktualisierte und erweiterte Auflage 2015
Copyright © 2015 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Geleitwort

C# und dessen Entwicklung sind untrennbar mit der darunterliegenden Laufzeitumgebung – dem .NET-Framework – verbunden. Denn obwohl es unter .NET eine ganze Reihe von Programmiersprachen gibt, nimmt C# als die Implementierungssprache von .NET eine Sonderstellung ein. Das .NET-Framework verfolgt das Ziel, die Entwicklung diverser Anwendungen auf unterschiedlichen Plattformen wie Windows, Windows Phone, iOS oder Android zu vereinfachen und den Entwickler dabei von Routineaufgaben, wie etwa dem Memory Management, zu entlasten.

Heute findet man .NET – und damit auch C# – nicht nur auf klassischen Desktops und Servern, sondern auch auf mobilen Geräten, auf Mikrocontrollern, auf Sensoren und ähnlichen Geräten, die im Umfeld von IoT (Internet of Things) eine zentrale Rolle spielen.

Um dieser Diversität gerecht zu werden, müssen das .NET-Framework und dessen Entwicklung ständig an neue Anforderungen angepasst werden. Beispiele dafür sind moderne CPUs, die Entwickler vor die Aufgabe stellen, sich mit parallelen Anwendungen zu beschäftigen, oder SIMD-Technologien für die Beschleunigung spezieller Berechnungen. Aber auch Spezifika bestimmter Geräteklassen wie kleine Geräte mit limitierten Ressourcen versus große Cloud-Anwendungen werden zunehmend berücksichtigt. Um diese Weiterentwicklung auf eine breite Basis zu stellen, werden zahlreiche .NET-Technologien – z.B. der C#-Compiler – mittlerweile im Rahmen der .NET Foundation als Open-Source-Projekte entwickelt.

Was hat das alles aber mit C# zu tun? Nun, C# wurde so entworfen, dass die Eigenschaften von .NET in dieser Sprache optimal genutzt werden und Teile von .NET selbst, wie etwa ein Großteil der Base Class Library, in dieser Sprache möglichst einfach implementiert werden können. So ermöglichen zum Beispiel partielle Klassen die effiziente Erweiterung generierter Klassen, Language Integrated Queries (LINQ) erlauben die einfachere Verarbeitung und Parallelisierung von Daten, und asynchrone Methoden ermöglichen eine bessere Nutzung verfügbarer Ressourcen bzw. die Erstellung benutzerfreundlicherer Anwendungen.

Die Mächtigkeit von C# sowie seine enge Verflechtung mit .NET sind jedoch für manchen Einsteiger ein wenig verwirrend. Genau hier setzt das vorliegende Buch an. Der Autor gibt darin – basierend auf seiner langjährigen Erfahrung mit Programmiersprachen – einen kompakten Überblick über C# für Praktiker. Die

Querverweise zu Java sowie zahlreiche Beispiele und Übungsaufgaben mit Musterlösungen ermöglichen ein rasches Einarbeiten in die Materie.

Dem Entwickler wird aber auch ein Blick hinter die Kulissen der Sprache gewährt, damit er auch dann versteht, was abläuft, wenn ihn etwaige Wizards bei der Softwareentwicklung mit C# unterstützen.

Andreas Schabus

Technology Advisor

Microsoft Österreich GmbH

Vorwort zur 4. Auflage

C# ist heute eine der meistverwendeten Programmiersprachen – sowohl in der Industrie als auch an Schulen und Universitäten. Aber nicht nur die Anwendergemeinschaft ist gewachsen, sondern auch der Sprachumfang. Die dritte Auflage dieses Buchs beschrieb C# 3.0 und enthielt auch bereits eine Vorschau auf C# 4.0. Mittlerweile stehen wir bei C# 5.0, was eine vierte Auflage dieses Buchs notwendig machte. In C# 5.0 kamen vor allem Sprachelemente und Bibliotheksklassen für die parallele und asynchrone Programmierung hinzu.

Die Taktraten moderner Rechner sind an einer physikalischen Grenze angekommen und können nicht mehr erhöht werden. Die einzige Möglichkeit, Rechner weiterhin schneller zu machen, ist ab nun die Erhöhung der Anzahl ihrer Prozessoren. Mittlerweile sind 4 bis 8 Prozessoren pro Chip bereits Standard und bald wird es Arbeitsplatzrechner mit 16 und mehr Prozessoren geben.

Um all diese Prozessoren auszunutzen, muss man Programme parallelisieren. Leider geht das nicht automatisch, sondern muss vom Programmierer sorgfältig geplant und umgesetzt werden. Die parallele Programmierung ist nicht trivial, und die dabei eingesetzten Sprachmittel waren anfangs rudimentär und kompliziert. Mittlerweile bieten moderne Sprachen wie C# 5.0 jedoch elegante Anweisungsarten und Klassen, die die parallele Programmierung wesentlich erleichtern.

Zum einen gibt es in C# nun Klassen wie `Task` oder `Parallel`, mit denen man Codestücke parallel zueinander laufen lassen kann – idealerweise auf unterschiedlichen Prozessoren. Zum anderen wurde eine neue Art von Methoden eingeführt (sogenannte *asynchrone Methoden*), bei denen der Rufer nicht mehr warten muss, bis die aufgerufene Methode zu Ende gelaufen ist, sondern gleich nach dem Aufruf mit anderen Arbeiten weitermachen kann. Der Rufer und die aufgerufene Methode laufen also (weitgehend) parallel, was vor allem den Vorteil hat, dass der Rufer nicht durch lange Wartezeiten blockiert ist. Falls die Methode Ergebnisse liefert, kann sich der Rufer diese zu einem späteren Zeitpunkt abholen.

Der C#-Compiler verwendet viel Aufwand dafür, diese Parallelität zu verstecken und dem Programmierer den Eindruck eines sequentiellen Programms zu geben, das im Hintergrund parallelisiert wird. Auf diese Weise wird die parallele und asynchrone Programmierung einfacher als je zuvor. Die neue Auflage des Buchs beschreibt die dazu notwendigen Sprachmittel samt Übungsaufgaben in einem eigenen Kapitel.

Unter <http://dotnet.jku.at> findet man wie bisher Musterlösungen zu den Übungsaufgaben am Ende jedes Kapitels. Die Webseite enthält außerdem Powerpoint-Folien einer Lehrveranstaltung über C# samt den neuen Sprachmerkmalen von C# 5.0.

Trotz des gewachsenen Sprachumfangs von C# ist auch die vierte Auflage dieses Buchs wieder kompakt, ohne auf Vollständigkeit oder Genauigkeit zu verzichten. Das Buch ermöglicht Studenten und Praktikern, sich rasch in C# einzuarbeiten und produktiv zu werden. Ich hoffe, dass es viele neue Freunde findet, und bin wie immer für Anregungen und Kritik offen.

Hanspeter Mössenböck
September 2014

Inhalt

1	C# und das .NET-Framework	1
1.1	Ähnlichkeiten zwischen C# und Java	1
1.2	Unterschiede zwischen C# und Java	3
1.3	Das .NET-Framework	4
1.4	Übungsaufgaben	10
2	Erste Schritte	11
2.1	Hello World	11
2.2	Gliederung von Programmen	12
2.3	Symbole	13
2.4	Übungsaufgaben	16
3	Typen	17
3.1	Einfache Typen	18
3.2	Enumerationen	19
3.3	Arrays	20
3.4	Strings	23
3.5	Structs	24
3.6	Klassen	25
3.7	object	26
3.8	Boxing und Unboxing	27
3.9	Übungsaufgaben	28
4	Ausdrücke	31
4.1	Arithmetische Ausdrücke	31
4.2	Vergleichsausdrücke	32
4.3	Boolesche Ausdrücke	33
4.4	Bit-Ausdrücke	33
4.5	Shift-Ausdrücke	33
4.6	Überlaufprüfung	34

4.7	typeof	34
4.8	sizeof	35
4.9	Übungsaufgaben	35
5	Deklarationen	37
5.1	Deklarationen in Namensräumen	38
5.2	Deklarationen in Klassen, Structs und Interfaces	39
5.3	Deklarationen in Enumerationstypen	40
5.4	Deklarationen in Blöcken	40
5.5	Übungsaufgaben	42
6	Anweisungen	43
6.1	Leeranweisung	43
6.2	Zuweisung	43
6.3	Methodenaufruf	44
6.4	if-Anweisung	44
6.5	switch-Anweisung	45
6.6	while-Anweisung	46
6.7	do-while-Anweisung	46
6.8	for-Anweisung	46
6.9	foreach-Anweisung	47
6.10	break- und continue-Anweisungen	48
6.11	goto-Anweisung	48
6.12	return-Anweisung	49
6.13	Übungsaufgaben	50
7	Ein-/Ausgabe	51
7.1	Ausgabe auf den Bildschirm	51
7.2	Formatierte Ausgabe	51
7.3	Ausgabe auf eine Datei	54
7.4	Eingabe von der Tastatur	54
7.5	Eingabe von einer Datei	55
7.6	Lesen der Kommandozeilenparameter	56
7.7	Übungsaufgaben	56
8	Klassen und Structs	57
8.1	Sichtbarkeitsattribute	58
8.2	Felder	60
8.3	Methoden	61

8.4	Konstruktoren	68
8.5	Destruktoren	70
8.6	Properties	71
8.7	Indexer	74
8.8	Überladene Operatoren	75
8.9	Geschachtelte Typen	78
8.10	Partielle Typen	79
8.11	Partielle Methoden	80
8.12	Statische Klassen	81
8.13	Unterschiede zu Java	82
8.14	Übungsaufgaben	82
9	Vererbung	85
9.1	Deklaration von Unterklassen	85
9.2	Kompatibilität zwischen Klassen	87
9.3	Überschreiben und Verdecken von Elementen	88
9.4	Dynamische Bindung	91
9.5	Konstruktoren in Ober- und Unterklasse	94
9.6	Abstrakte Klassen	95
9.7	Versiegelte Klassen	96
9.8	Die Klasse Object	97
9.9	Übungsaufgaben	99
10	Interfaces	101
10.1	Deklaration und Verwendung von Interfaces	101
10.2	Operationen auf Interfaces	103
10.3	Erweiterung von Interfaces	104
10.4	Namenskonflikte	105
10.5	Interface IDisposable	106
10.6	Übungsaufgaben	107
11	Delegates und Events	109
11.1	Einfache Delegates	109
11.2	Multicast-Delegates	110
11.3	Erzeugen von Delegate-Werten	110
11.4	Ereignisse (Events)	112
11.5	Anonyme Methoden	113
11.6	Übungsaufgaben	115

12 Ausnahmen	117
12.1 try-Anweisung	117
12.2 Ausnahmeklassen	118
12.3 Auslösen von Ausnahmen	120
12.4 Ausnahmen in aufgerufenen Methoden	122
12.5 Ausnahmen in Multicast-Delegates	122
12.6 Übungsaufgaben	123
13 Namensräume und Assemblies	125
13.1 Namensräume	125
13.2 Assemblies	128
13.2.1 Assemblies und Module	128
13.2.2 Versionierung von Assemblies	129
13.2.3 Assemblies versus Namensräume	132
13.3 Übungsaufgaben	133
14 Generische Bausteine	135
14.1 Generische Typen	136
14.2 Constraints	137
14.3 Vererbung bei generischen Typen	138
14.4 Generische Methoden	140
14.5 Generische Delegates	141
14.6 Nullwerte	143
14.7 Ko- und Kontravarianz bei generischen Typen	143
14.8 Was geschieht hinter den Kulissen?	147
14.9 Unterschiede zu Java	147
14.10 Übungsaufgaben	148
15 Threads	151
15.1 Die Klasse Thread	151
15.2 Zustände eines Threads	154
15.3 Abbrechen eines Threads	155
15.4 Thread-Synchronisation	156
15.5 Übungsaufgaben	161
16 Iteratoren	163
16.1 Allgemeine Iteratoren	163
16.2 Spezifische Iteratoren	165
16.3 Übungsaufgaben	168

17 Attribute	169
17.1 Schreibweise von Attributen	169
17.2 Parameter von Attributen	170
17.3 Attribute für spezifische Programmelemente	171
17.4 Attribut Serializable	172
17.5 Attribut Conditional	174
17.6 Attribut DllImport	175
17.7 Deklaration eigener Attribute	176
17.8 Übungsaufgaben	177
18 Dokumentationskommentare	179
18.1 XML-Elemente	179
18.2 Erzeugte XML-Datei	181
18.3 Übungsaufgaben	182
19 Auszug aus der .NET-Klassenbibliothek	183
19.1 Hilfsklassen	184
19.2 Collections	187
19.3 Ein-/Ausgabe	196
19.4 Reflection	202
19.5 Übungsaufgaben	206
20 LINQ	209
20.1 Motivation	209
20.2 Lambda-Ausdrücke	210
20.3 Erweiterungsmethoden	214
20.4 Objektinitialisierer	216
20.5 Anonyme Typen	218
20.6 Query-Ausdrücke	220
20.7 LINQ und XML	227
20.8 Übungsaufgaben	229
21 Asynchrone Methoden und Parallelität	231
21.1 Asynchronität	231
21.2 Tasks	232
21.3 Asynchrone Methoden	234
21.4 Explizite Parallelität	240
21.5 Übungsaufgaben	242

22 Interoperabilität mit COM	243
22.1 COM-Objekte von .NET aus ansprechen	244
22.2 .NET-Assemblies von COM aus ansprechen	247
22.3 Übungsaufgaben	249
23 Dynamisch getypte Variablen	251
23.1 Typ dynamic	251
23.2 Operationen auf dynamic-Variablen	253
24 Fallstudien	255
24.1 Anwendungen mit grafischer Benutzeroberfläche	255
24.2 Ein Web-Service für Börsenkurse	264
24.3 Dynamische Webseiten mit ASP.NET	269
24.4 Übungsaufgaben	275
A Anhang	277
A.1 Compileroptionen	277
A.2 Werkzeuge unter .NET	280
A.2.1 ildasm	280
A.2.2 Globaler Assembly-Cache	281
A.3 Grammatik von C#	284
A.4 Unicode und ASCII	291
Literatur	293
Index	295

1 C# und das .NET-Framework

C# (sprich: *see sharp*) ist eine von Microsoft entwickelte Programmiersprache für die .NET-Plattform ([HTWG10]). Obwohl man .NET-Programme in ganz verschiedenen Sprachen schreiben kann (unter anderem in C++, Visual Basic, Java, Cobol oder Eiffel), hat Microsoft mit C# eine neue »Haussprache« geschaffen, um damit die Mächtigkeit von .NET voll auszureizen. C# ist eine objektorientierte Sprache, die sich äußerlich stark an Java anlehnt, aber in ihrer Mächtigkeit deutlich darüber hinausgeht. Sie besitzt all jene Eigenschaften, die man benötigt, um Programme nach dem neuesten Stand der Softwaretechnik zu entwickeln.

C# ist keine revolutionäre Sprache. Sie ist vielmehr eine Kombination aus Java, C++ und Visual Basic, wobei man versucht hat, von jeder Sprache die bewährten Eigenschaften zu übernehmen und die komplexen Eigenschaften zu vermeiden. C# wurde von einem kleinen Team unter der Leitung von *Anders Hejlsberg* entworfen. Hejlsberg ist ein erfahrener Sprachdesigner. Er war bei Borland Chefentwickler von Delphi und ist dafür bekannt, seine Sprachen auf die Bedürfnisse von Praktikern zuzuschneiden.

In diesem Kapitel geben wir einen Überblick über die wichtigsten Eigenschaften von C#. Aufgrund der Ähnlichkeiten zu Java stellen wir dabei die Merkmale von C# denen von Java gegenüber, wobei wir davon ausgehen, dass der Leser bereits programmieren kann und eine Sprache wie Java oder C++ beherrscht. Da man als C#-Entwickler nicht umhinkommt, auch die Grundkonzepte von .NET zu kennen, gehen wir am Ende dieses Kapitels auch kurz auf .NET ein.

1.1 Ähnlichkeiten zwischen C# und Java

Auf den ersten Blick sehen C#-Programme wie Java-Programme aus. Jeder Java-Programmierer sollte daher in der Lage sein, C#-Programme zu lesen. Neben der fast identischen Syntax wurden folgende Konzepte aus Java übernommen:

■ *Objektorientierung*

C# ist wie Java eine objektorientierte Sprache mit einfacher Vererbung. Klassen können nur von einer einzigen Klasse erben, aber mehrere Schnittstellen (Interfaces) implementieren.

■ *Typsicherheit*

C# ist eine typsichere Sprache. Viele Programmierfehler, die durch inkompatible Datentypen in Anweisungen und Ausdrücken entstehen, werden bereits vom Compiler abgefangen. Zeigerarithmetik oder ungeprüfte Typumwandlungen wie in C++ sind in Anwendungsprogrammen verboten. Zur Laufzeit wird sichergestellt, dass Array-Indizes im erlaubten Bereich liegen, dass Objekte nicht durch uninitialisierte Zeiger referenziert werden und dass Typumwandlungen zu einem definierten Ergebnis führen.

■ *Garbage Collection*

Dynamisch erzeugte Objekte werden vom Programmierer nie selbst freigegeben, sondern von einem Garbage Collector automatisch eingesammelt, sobald sie nicht mehr referenziert werden. Das beseitigt viele unangenehme Fehler, die z.B. in C++-Programmen auftreten können.

■ *Namensräume*

Was in Java Pakete sind, nennt man in C# Namensräume. Ein Namensraum ist eine Sammlung von Deklarationen und ermöglicht es, gleiche Namen in unterschiedlichem Kontext zu verwenden.

■ *Threads*

C# unterstützt leichtgewichtige parallele Prozesse in Form von Threads. Es gibt wie in Java Mechanismen zur Synchronisation und Kommunikation zwischen Prozessen.

■ *Generizität*

Sowohl Java als auch C# kennen generische Typen und Methoden. Damit kann man Bausteine herstellen, die mit anderen Typen parametrisierbar sind (z.B. Listen mit beliebigem Elementtyp).

■ *Reflection*

Wie in Java kann man auch in C# zur Laufzeit auf Typinformationen eines Programms zugreifen, Klassen dynamisch zu einem Programm hinzuladen, ja sogar Objektprogramme zur Laufzeit zusammenstellen.

■ *Attribute*

Der Programmierer kann beliebige Informationen an Klassen, Methoden oder Felder hängen und sie zur Laufzeit mittels Reflection abfragen. In Java heißt dieser Mechanismus *Annotationen*.

■ *Bibliotheken*

Viele Typen der C#-Bibliothek sind denen der Java-Bibliothek nachempfunden. So gibt es vertraute Typen wie `Object`, `String`, `ICollection` oder `Stream`, meist sogar mit den gleichen Methoden wie in Java.

Auch aus C++ wurden einige Dinge übernommen, zum Beispiel das Überladen von Operatoren, die Zeigerarithmetik in systemnahen Klassen (die als *unsafe* gekennzeichnet sein müssen) sowie einige syntaktische Details z.B. im Zusammenhang mit Vererbung. Aus Visual Basic stammt beispielsweise die `foreach`-Schleife.

1.2 Unterschiede zwischen C# und Java

Neben diesen Ähnlichkeiten weist C# aber wie alle .NET-Sprachen auch einige Merkmale auf, die in Java fehlen:

- *Referenzparameter*
Parameter können nicht nur durch *call by value* übergeben werden, wie das in Java üblich ist, sondern auch durch *call by reference*. Dadurch sind nicht nur Eingangs-, sondern auch Ausgangs- und Übergangsparameter realisierbar.
- *Objekte am Keller*
Während in Java alle Objekte am Heap liegen, kann man in C# Objekte auch am Methodenaufrufl Keller anlegen. Diese Objekte sind leichtgewichtig und belasten den Garbage Collector nicht.
- *Blockmatrizen*
Für numerische Anwendungen ist das Java-Speichermodell mehrdimensionaler Arrays zu ineffizient. C# lässt dem Programmierer die Wahl, mehrdimensionale Arrays entweder wie in Java anzulegen oder als kompakte Blockmatrizen, wie das in C, Fortran oder Pascal üblich ist.
- *Einheitliches Typsystem*
Im Gegensatz zu Java sind in C# alle Datentypen (auch int oder char) vom Typ object abgeleitet und erben die dort deklarierten Methoden.
- *goto-Anweisung*
Die viel geschmähte goto-Anweisung wurde in C# wieder eingeführt, allerdings mit Einschränkungen, so dass man mit ihr kaum Missbrauch treiben kann.
- *Versionierung*
Bibliotheken werden bei der Übersetzung mit einer Versionsnummer versehen. So kann eine Bibliothek gleichzeitig in verschiedenen Versionen vorhanden sein. Jede Applikation verwendet immer diejenige Version der Bibliothek, mit der sie übersetzt und getestet wurde.

Schließlich hat C# noch eine ganze Reihe von Eigenschaften, die zwar die Mächtigkeit der Sprache nicht erhöhen, aber bequem zu benutzen sind. Sie fallen unter die Kategorie »*syntactic sugar*«, d.h., man kann mit ihnen Dinge tun, die man auch in anderen Sprachen realisieren könnte, nur dass es in C# eben einfacher und eleganter geht. Dazu gehören:

- *Properties und Events*
Diese Eigenschaften dienen der Komponententechnologie. Properties sind spezielle Felder eines Objekts. Greift man auf sie zu, werden automatisch get- und set-Methoden aufgerufen. Mit Events kann man Ereignisse definieren, die von Komponenten ausgelöst und von anderen behandelt werden.

- *Indexer*
Ein Index-Operator wie bei Array-Zugriffen kann durch get- und set-Methoden selbst definiert werden.
- *Delegates*
Delegates sind im Wesentlichen das, was man in Pascal *Prozedurvariablen* und in C *Function Pointers* nennt. Sie sind allerdings etwas mächtiger. Zum Beispiel kann man mehrere Prozeduren in einer einzigen Delegate-Variablen speichern.
- *foreach-Schleife*
Damit kann man bequem über Arrays, Listen oder Mengen iterieren.
- *Iteratoren*
Man kann spezielle Iterator-Methoden schreiben, die eine Folge von Werten liefern, welche dann mit foreach durchlaufen werden kann.
- *Lambda-Ausdrücke*
Lambda-Ausdrücke sind parametrisierte Codestücke, die man an Variablen zuweisen und später aufrufen kann. Sie sind eine Kurzform für namenlose Methoden.
- *Query-Ausdrücke*
Sie erlauben SQL-ähnliche Abfragen auf Hauptspeicherdaten wie Arrays oder Listen.

1.3 Das .NET-Framework

Wer in C# programmiert, kommt früher oder später nicht umhin, sich auch in die Grundlagen des .NET-Frameworks einzuarbeiten, für das C# entwickelt wurde. Das .NET-Framework ist eine Schicht, die auf Windows (und später vielleicht auch einmal auf anderen Betriebssystemen) aufsetzt (siehe Abb. 1–1) und vor allem zwei Dinge hinzufügt:

- Eine **Laufzeitumgebung** (die *Common Language Runtime*), die automatische Speicherbereinigung (*garbage collection*), Sicherheitsmechanismen, Versionierung und vor allem Interoperabilität zwischen verschiedenen Programmiersprachen bietet.
- Eine **objektorientierte Klassenbibliothek** mit umfangreichen Funktionen für grafische Benutzeroberflächen (*Windows Forms*), Web-Oberflächen (*ASP.NET*), Datenbankanschluss (*ADO.NET*), Web-Services, Collection-Klassen, Threads, Reflection und vieles mehr. Sie ersetzt in vielen Fällen das bisherige Windows-API und geht weit über dieses hinaus.

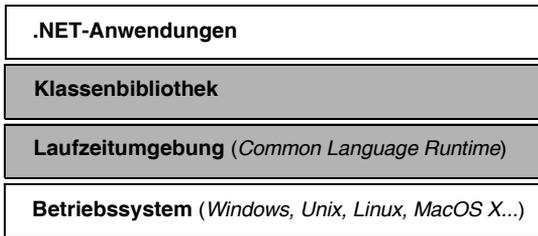


Abb. 1–1 Grobarchitektur des .NET-Frameworks

Obwohl .NET von Microsoft entwickelt wurde, basiert es auf offenen Standards. Der ECMA-Standard 335 definiert zum Beispiel die Common Language Runtime und Teile der Klassenbibliothek, der ECMA-Standard 334 beschreibt die Sprache C#, und auch in Web-Services werden allgemeine Standards wie SOAP, WSDL oder UDDI verwendet. Im Rahmen eines Open-Source-Projekts ([Mono]) wurde das .NET-Framework auf Linux portiert, und Microsoft selbst stellt sogar große Teile des Quellcodes der CLR unter dem Namen SSCLI (*Shared Source Common Language Infrastructure*) zur Verfügung ([SNS03]).

Dieser Abschnitt gibt einen Überblick über die wichtigsten Teile des .NET-Frameworks. Eine ausführlichere Beschreibung findet man zum Beispiel in [BBMW03], [NEGWS12]. oder in [SDKDoc]. Teile der Klassenbibliothek werden in Kapitel 19 beschrieben.

Common Language Runtime

Die *Common Language Runtime* (CLR) ist die Laufzeitumgebung, unter der .NET-Programme ausgeführt werden und die unter anderem Garbage Collection, Sicherheit und Interoperabilität unterstützt.

Ähnlich wie die Java-Umgebung basiert die CLR auf einer *virtuellen Maschine* mit einem eigenen Befehlssatz (CIL – *Common Intermediate Language*), in den die Programme aller .NET-Sprachen übersetzt werden. Unmittelbar vor der Ausführung (*just in time*) werden CIL-Programme dann in den Code der Zielmaschine (z.B. in Intel-Code) umgewandelt (siehe Abb. 1–2). Der CIL-Code garantiert die Interoperabilität zwischen den verschiedenen Sprachen und die Portabilität des Codes, die JIT-Compilation (*just in time compilation*) stellt sicher, dass die Programme trotzdem effizient ausgeführt werden.

Damit verschiedene Sprachen zusammenarbeiten können, genügt es aber nicht, sie in CIL-Code zu übersetzen. Es muss auch gewährleistet sein, dass sie die gleiche Art von Datentypen benutzen. Die CLR definiert daher auch ein gemeinsames Typsystem – das *Common Type System* (CTS), das festlegt, wie Klassen, Interfaces und andere Typen auszusehen haben. Das CTS erlaubt nicht nur, dass eine Klasse, die zum Beispiel in C# implementiert wurde, von einem Visual-Basic-Programm benutzt werden kann; es ist sogar möglich, diese C#-Klasse in Visual Basic

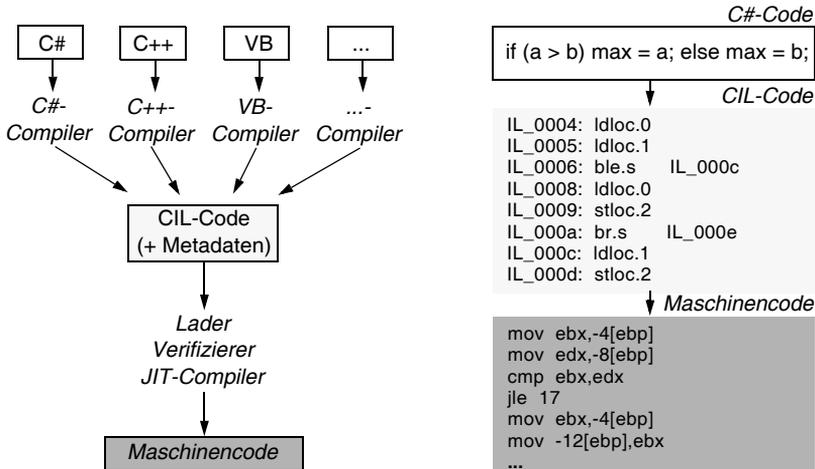


Abb. 1–2 Quellcode, CIL-Code und Maschinencode

durch eine Unterklasse zu erweitern oder eine Ausnahme (*exception*), die in C# ausgelöst wurde, von einem Programm in einer anderen Sprache behandeln zu lassen.

Die CLR stellt Mechanismen zur Verfügung, die .NET-Programme sicherer und robuster machen. Dazu gehört zum Beispiel der *Garbage Collector*, der dafür zuständig ist, den Speicherplatz von Objekten freizugeben, sobald diese nicht mehr benutzt werden. In älteren Sprachen wie C oder C++ ist der Programmierer für die Freigabe von Objekten selbst verantwortlich. Dabei kann es vorkommen, dass er ein Objekt freigibt, das noch von anderen Objekten benutzt wird. Diese Objekte greifen dann »ins Leere« und zerstören fremde Speicherbereiche. Umgekehrt kann es vorkommen, dass ein Programmierer vergisst, Objekte freizugeben, obwohl sie nicht mehr referenziert werden. Diese bleiben dann als Speicherleichen (*memory leaks*) zurück und verschwenden Platz. Solche Fehler sind schwer zu finden, können aber dank Garbage Collector unter .NET nicht vorkommen.

Wenn ein CIL-Programm geladen und in Maschinencode übersetzt wird, prüft die CLR mittels eines *Verifizierers*, dass die Typregeln des CTS nicht verletzt werden. Es ist zum Beispiel verboten, eine Zahl als Adresse zu interpretieren und damit auf fremde Speicherbereiche zuzugreifen.

Assemblies

.NET unterstützt komponentenorientierte Softwareentwicklung. Die Komponenten heißen *Assemblies* und sind die kleinsten Programmbausteine, die separat ausgeliefert werden können. Ein Assembly ist eine Sammlung von Klassen und anderen Ressourcen (z.B. Bildern) und wird entweder als ausführbare EXE-Datei oder als Bibliotheksbaustein in Form einer DLL-Datei (*dynamic link library*) gespeichert (siehe Abb. 1–3). In manchen Fällen kann ein Assembly auch aus mehreren Dateien bestehen.

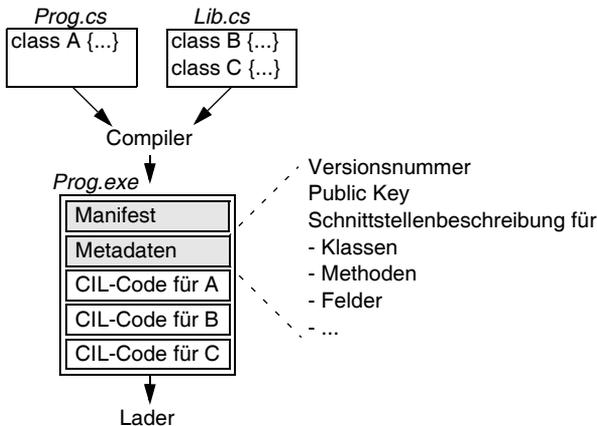


Abb. 1-3 Vom Compiler erzeugtes Assembly *Prog.exe*

Jedes Assembly enthält neben Code auch *Metadaten*, also die Schnittstellenbeschreibung seiner Klassen, Felder, Methoden und sonstigen Programmelemente. Zusätzlich enthält es ein *Manifest*, das man sich als Inhaltsverzeichnis vorstellen kann. Assemblies sind also selbstbeschreibend und können mittels *Reflection* vom Lader, Compiler und anderen Werkzeugen analysiert und benutzt werden.

Assemblies dienen auch der Versionierung, d.h., sie haben eine mehrstufige Versionsnummer, die für alle in ihnen enthaltenen Klassen gilt. Wenn eine Klasse übersetzt wird, werden in ihrem Objektcode die Versionsnummern der aus anderen Assemblies benutzten Klassen vermerkt. Der Lader lädt dann jene Assemblies, die der erwarteten Versionsnummer entsprechen. Unter .NET können also mehrere gleichnamige DLLs mit unterschiedlichen Versionsnummern nebeneinander existieren (*side by side execution*), ohne sich in die Quere zu kommen. Das bedeutet das Ende der »DLL Hell« unter Windows, bei der durch die Installation neuer Software alte DLLs durch gleichnamige neue überschrieben werden konnten und dadurch existierende Software plötzlich nicht mehr funktionierte.

Assemblies müssen auch nicht mehr in die Windows-Registry eingetragen werden. Man kopiert sie einfach ins Applikationsverzeichnis oder in den so genannten *Global Assembly Cache* und kann sie ebenso einfach wieder entfernen.

Assemblies sind gewissermaßen die Nachfolger von COM-Komponenten. Anders als unter COM (*Component Object Model*) braucht man Assemblies aber nicht mehr durch eine IDL (*Interface Definition Language*) zu beschreiben, da sie ja die vollständigen Metadaten enthalten, die der Compiler aus ihrem Quellcode gewonnen hat. Das Common Type System stellt sicher, dass Software, die in unterschiedlichen Sprachen geschrieben wurde, die gleiche Art von Metadaten benutzt und somit binärkompatibel ist. Investitionen in die COM-Technologie sind aber nicht verloren. Es ist möglich, COM-Komponenten von .NET-Klassen aus zu verwenden und umgekehrt (siehe Kapitel 22).

ADO.NET

ADO.NET umfasst alle Klassen der .NET-Bibliothek, die für den Zugriff auf Datenbanken und andere Datenquellen (z.B. XML-Dateien) zuständig sind. Es gab bereits eine Vorgängertechnologie namens ADO (*ActiveX Data Objects*), die jedoch mit ADO.NET nur den Namen gemeinsam hat. ADO.NET ist objektorientiert und somit strukturierter und einfacher zu benutzen.

ADO.NET unterstützt das relationale Datenmodell mit Transaktionen und Sperrmechanismen. Dabei ist es unabhängig von verschiedenen Anbietern und Datenbankarchitekturen. Implementierungen konkreter Datenbankverbindungen an MS SQL Server, OLE DB (*Object Linking and Embedding Database*) und ODBC (*Open Database Connectivity*) werden durch gemeinsame Interfaces abstrahiert.

Der Zugriff auf Datenquellen kann verbindungsorientiert oder verbindungslos erfolgen. Im ersten Fall wird eine ständige Verbindung zur Datenquelle aufrechterhalten, im zweiten Fall wird ein Schnappschuss eines Teils der Datenbank in ein DataSet-Objekt geholt und dann lokal weiterverarbeitet. In beiden Fällen greift man auf die Daten in der Regel mittels SQL (*Structured Query Language*) zu.

ASP.NET

ASP.NET ist jener Teil der .NET-Technologie, der die Programmierung dynamischer Webseiten abdeckt. Mit der Vorgängertechnologie ASP (*Active Server Pages*) hat auch ASP.NET nur den Namen gemeinsam. Das Programmiermodell hat sich grundlegend geändert.

Mit ASP.NET werden Webseiten am Server dynamisch aus aktuellen Daten zusammengestellt und in Form von reinem HTML an Klienten geschickt, wo sie von jedem Web-Browser angezeigt werden können. Im Gegensatz zu ASP wird in ASP.NET ein objektorientiertes Programmiermodell verwendet. Sowohl die Webseite als auch die in ihr vorkommenden GUI-Elemente sind Objekte, die man über einen Namen ansprechen und auf deren Felder und Methoden man in Programmen zugreifen kann. All das geschieht in einer kompilierten Sprache wie C# oder Visual Basic .NET und nicht wie in ASP in einer interpretierten Sprache wie JavaScript oder VBScript. Daher hat man auch Zugriff auf die gesamte Klassenbibliothek von .NET.

Die Verarbeitung von Benutzereingaben folgt einem ereignisgesteuerten Modell. Wenn der Benutzer ein Textfeld ausfüllt, einen Button anklickt oder einen Eintrag aus einer Liste wählt, wird ein Ereignis ausgelöst, das dann durch serverseitigen Code behandelt werden kann. Obwohl der Server – wie am Internet üblich – zustandslos ist, wird der Zustand einer Webseite zwischen den einzelnen Benutzeraktionen aufbewahrt, und zwar in der Seite selbst. Das stellt eine wesentliche Erleichterung gegenüber älteren Programmiermodellen dar, bei denen der Programmierer für die Zustandsverwaltung selbst verantwortlich war.

ASP.NET bietet eine reichhaltige Bibliothek von GUI-Elementen, die weit über das hinausgeht, was unter HTML verfügbar ist, obwohl alle GUI-Elemente letzt-

endlich auf HTML abgebildet werden. Der Programmierer hat sogar die Möglichkeit, eigene GUI-Elemente zu implementieren und somit die Benutzeroberfläche von Webseiten seinen speziellen Bedürfnissen anzupassen. Besonders einfach ist die Darstellung von Datenbankabfrageergebnissen in Form von Listen und Tabellen, was von ASP.NET weitgehend automatisiert wird. Eine weitere Neuheit von ASP.NET sind Validatoren, mit denen Benutzereingaben auf ihre Gültigkeit überprüft werden können.

Mit der Entwicklungsumgebung Visual Studio .NET kann man Webseiten interaktiv erstellen, wie man das bei Benutzeroberflächen von Desktop-Anwendungen gewohnt ist. GUI-Elemente können mit der Maus in einem Fenster positioniert werden. Über Menüs und Property-Fenster kann man Attribute setzen und Methoden spezifizieren, die als Reaktion auf Benutzereingaben aufgerufen werden sollen. All das verwischt die Unterschiede zwischen der Programmierung lokaler Desktop-Anwendungen und Internet-Anwendungen und erleichtert zum Beispiel das Erstellen von Web-Shops und tagesaktuellen Informationsseiten (z.B. Börseninformationen). ASP.NET wird in Abschnitt 24.3 näher erklärt.

Web-Services

Web-Services werden von Microsoft als einer der Kernpunkte der .NET-Technologie bezeichnet, obwohl es sie auch außerhalb von .NET gibt. Es handelt sich um Prozedurfernaufrufe (*remote procedure calls*), die als Protokolle meist HTTP und SOAP (eine Anwendung von XML) benutzen.

Das Internet hat sich als äußerst leistungsfähig und geeignet erwiesen, um auf weltweit verstreute Informationen und Dienste zuzugreifen. Bisher erfolgte dieser Zugriff jedoch meist über Web-Browser. Web-Services sollen nun eine neue Art des Zusammenspiels zwischen verteilten Applikationen ermöglichen, bei denen die Kommunikation ohne Web-Browser abläuft. Normale Desktop-Anwendungen können sich Informationen wie aktuelle Wechselkurse oder Buchungsdaten über ein oder mehrere Web-Services holen, die als Prozeduren auf anderen Rechnern laufen und über das Internet angesprochen werden.

Die Aufrufe und Parameter werden dabei in der Regel mittels SOAP [SOAP] codiert, eines auf XML basierenden Standards, der von den meisten großen Firmen unterstützt wird. Der Programmierer merkt jedoch von all dem nichts. Er ruft einen Web-Service wie eine normale Methode auf, und .NET sorgt dafür, dass der Aufruf nach SOAP umgewandelt, über das Internet verschickt und auf dem Zielrechner wieder decodiert wird. Am Zielrechner wird die gewünschte Methode aufgerufen, die ihre Ergebnisse wieder transparent über SOAP an den Rufer zurückschickt. Der Rufer und die gerufene Methode können dabei in ganz verschiedenen Sprachen geschrieben sein und auf unterschiedlichen Betriebssystemen laufen.

Damit .NET die SOAP-Codierung und Decodierung korrekt durchführen kann, werden Web-Services samt ihren Parametern mittels WSDL (*Web Services Description Language* [WSDL]) beschrieben. Auch das erledigt .NET automatisch. Web-Services werden in Abschnitt 24.2 dieses Buchs näher erklärt.

1.4 Übungsaufgaben¹

1. **Eignung von C# für große Softwareprojekte**
Inwiefern helfen die Eigenschaften von C# bei der Entwicklung großer Softwareprojekte?
2. **Merkmale von .NET**
Was sind die Hauptmerkmale des .NET-Frameworks? Welche dieser Merkmale ähneln der Java-Umgebung und welche sind neu?
3. **Sicherheit**
Begründen Sie, warum C# eine sichere Sprache ist. Welche Arten von Programmierfehlern oder gefährlichen Situationen werden vom C#-Compiler oder der CLR abgefangen?
4. **Interoperabilität**
Warum können unter .NET Programme, die in unterschiedlichen Sprachen geschrieben wurden, nahtlos zusammenarbeiten?
5. **Assemblies**
Warum sind .NET-Assemblies einfacher zu installieren und zu deinstallieren als COM-Objekte?
6. **Web-Recherche**
Besuchen Sie die Webseiten [MS], [MSDN] und [CodeGal], um sich einen Überblick über das .NET-Framework und C# zu verschaffen.
7. **Mono**
Besuchen Sie die Webseite [Mono], um mehr über die Portierung von .NET auf Linux zu erfahren.

1. Musterlösungen zu den Übungsaufgaben in diesem Buch findet man unter [JKU].

2 Erste Schritte

Dieses Kapitel beschreibt die Grundstruktur von C#-Programmen sowie ihre Übersetzung und Ausführung mit dem .NET *Software Development Kit* (SDK). Es zeigt auch, aus welchen Symbolen C#-Programme zusammengesetzt sind.

2.1 Hello World

Wir beginnen mit dem bekannten Hello-World-Programm, das einfach den Text "Hello World" auf dem Bildschirm ausgibt. In C# sieht es folgendermaßen aus:

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

Das Programm besteht aus einer Klasse `Hello` und einer Methode `Main` (Achtung: Groß- und Kleinschreibung ist in C# signifikant). Jedes Programm hat genau eine `Main`-Methode, die aufgerufen wird, wenn man es startet. Die Ausgabeanweisung heißt hier `Console.WriteLine("...")`, wobei `WriteLine` eine Methode der Klasse `Console` ist, die aus dem Namensraum `System` stammt. Um `Console` bekannt zu machen, muss man `System` in der ersten Zeile mittels `using` importieren. C#-Programme werden in Dateien mit der Endung `.cs` gespeichert.

Die einfachste Arbeitsumgebung für .NET ist das *Software Development Kit* (SDK) von Microsoft, das man sich kostenlos von [MS] besorgen kann. Es ist kommandozeilenorientiert und bietet neben einem Compiler (`csc`) noch einige andere Werkzeuge (z.B. `sn`, `ildasm`), die in Anhang A.2 beschrieben werden. Wenn wir unser Hello-World-Programm in eine Datei `Hello.cs` abspeichern, können wir es durch Eingabe von

```
csc Hello.cs
```

im Konsolenfenster übersetzen und mittels

```
Hello
```

aufzurufen. Die Ausgabe erscheint wieder im Konsolenfenster.

Der Dateiname (z.B. Hello.cs) muss übrigens unter .NET nicht wie in Java mit dem Klassennamen (z.B. Hello) übereinstimmen, obwohl es aus Lesbarkeitsgründen empfehlenswert ist. Eine Datei kann auch mehrere Klassen enthalten. In diesem Fall sollte sie nach der Hauptklasse benannt sein.

2.2 Gliederung von Programmen

Der Quelltext eines C#-Programms kann auf mehrere Dateien verteilt sein. Jede Datei kann aus einem oder mehreren Namensräumen bestehen, von denen jeder eine oder mehrere Klassen oder andere Typen enthalten kann. Abb. 2–1 zeigt diese Struktur.

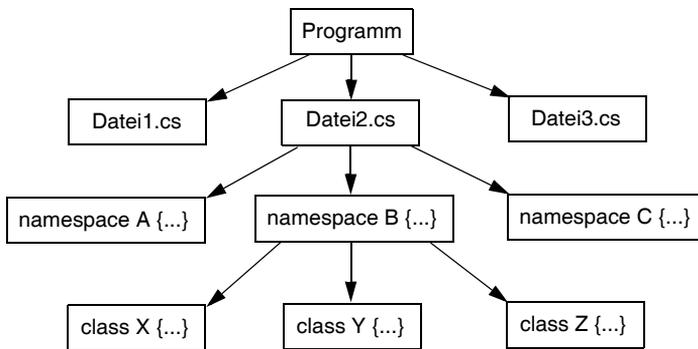


Abb. 2–1 Gliederung von Programmen

Unser Hello-World-Programm besteht nur aus einer einzigen Datei und einer einzigen Klasse. Namensraum wurde keiner angegeben, was bedeutet, dass die Klasse Hello zu einem namenlosen Standardnamensraum gehört, den .NET für uns bereitstellt. Namensräume werden in Kapitel 5 und 13 behandelt, Klassen in Kapitel 8.

Programme aus mehreren Dateien

Wenn ein Programm aus mehreren Dateien besteht, können wir diese entweder gemeinsam oder getrennt übersetzen. Im ersten Fall entsteht eine einzige ausführbare Datei, im zweiten Fall eine ausführbare Datei und eine DLL (*dynamic link library*).

Nehmen wir an, eine Klasse Counter in der Datei Counter.cs wird von einer Klasse Prog in der Datei Prog.cs benutzt:

```

public class Counter { // Dieser Code steht in der Datei Counter.cs
    private int val = 0;
    public void Add(int x) { val = val + x; }
    public int Val() { return val; }
}

using System; // Dieser Code steht in der Datei Prog.cs
public class Prog {
    public static void Main() {
        Counter c = new Counter();
        c.Add(3); c.Add(5);
        Console.WriteLine("val = " + c.Val());
    }
}

```

Wir können diese beiden Dateien nun gemeinsam übersetzen:

```
csc Prog.cs Counter.cs
```

wodurch eine ausführbare Datei Prog.exe entsteht, die beide Klassen enthält.

Alternativ dazu könnten wir aus Counter aber auch eine Bibliothek (DLL) machen, indem wir schreiben:

```
csc /target:library Counter.cs
```

Der Compiler erzeugt auf diese Weise eine Datei Counter.dll, die wir dann bei der Übersetzung von Prog.cs folgendermaßen referenzieren müssen:

```
csc /reference:Counter.dll Prog.cs
```

Aus dieser Übersetzung entsteht zwar auch eine Datei Prog.exe; sie enthält aber nur die Klasse Prog. Die Klasse Counter steht nach wie vor in der Datei Counter.dll und wird beim Aufruf von Prog dynamisch dazugeladen. Die verschiedenen Formen des Compileraufrufs werden in Anhang A.1 genauer beschrieben.

2.3 Symbole

C#-Programme bestehen aus Namen, Schlüsselwörtern, Zahlen, Zeichen, Zeichenketten, Operatoren und Kommentaren. Wir beschreiben diese Symbole hier informell. In Anhang A.3 kann man ihre genaue Syntax nachlesen.

Namen. Ein Name besteht aus Buchstaben, Ziffern und dem Zeichen "_". Das erste Zeichen muss ein Buchstabe oder ein "_" sein. Groß- und Kleinbuchstaben haben unterschiedliche Bedeutung (d.h. red ist ungleich Red). Da C# den Unicode-Zeichensatz benutzt (siehe Anhang A.4), können Namen auch griechische, arabische oder chinesische Zeichen enthalten. Man muss sie allerdings auf unseren Tastaturen als Nummerncodes eingeben. Der Code `\u03C0` bedeutet z.B. π , die Zeichenfolge `b\u0061c` den Namen back.

Schlüsselwörter. C# kennt 77 Schlüsselwörter, Java nur 50. Das deutet schon darauf hin, dass C# komplexer ist als Java. Schlüsselwörter sind reserviert, d.h., sie dürfen nicht als Namen verwendet werden. Will man ein Schlüsselwort dennoch als Name verwenden, so muss man das Zeichen @ davorsetzen (z.B. @if).

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

Namenskonventionen. Bei der Namenswahl und bei der Groß-/Kleinschreibung sollte man sich an die Regeln halten, die auch in der Klassenbibliothek von C# benutzt werden:

- Namen beginnen mit großen Anfangsbuchstaben (z.B. Length, WriteLine), außer bei lokalen Variablen und Parametern (z.B. i, len) oder bei Feldern einer Klasse, die von außen nicht sichtbar sind.
- In zusammengesetzten Wörtern beginnt jedes Wort mit einem Großbuchstaben (z.B. WriteLine). Die Trennung von Wörtern durch "_" wird in C# selten verwendet.
- Methoden ohne Rückgabewert sollten mit einem Verb beginnen (z.B. DrawLine). Alles andere sollte in der Regel mit einem Substantiv beginnen (z.B. Size, IndexOf, Collection). Felder oder Methoden mit booleschem Typ können auch mit einem Adjektiv beginnen, wenn sie eine Eigenschaft ausdrücken (z.B. Empty).
- Da Schlüsselwörter und Namen aus der .NET-Bibliothek englisch sind, sollte man auch seine eigenen Programmobjekte englisch benennen.

Zeichen und Zeichenketten. Zeichenkonstanten werden zwischen einfache Hochkommas eingeschlossen (z.B. 'x'), Zeichenkettenkonstanten zwischen doppelte Hochkommas (z.B. "John"). In beiden dürfen beliebige Zeichen vorkommen, außer das schließende Hochkomma, ein Zeilenende oder das Zeichen \, das als *Escape*-Zeichen verwendet wird. Folgende Escape-Sequenzen dienen zur Darstellung von Sonderzeichen in Zeichen- und Zeichenkettenkonstanten:

\	'
\"	"
\\	\
\0	0x0000 (das Zeichen mit dem Wert 0)
\a	0x0007 (alert)
\b	0x0008 (backspace)

```

\f      0x000c (form feed)
\n      0x000a (new line)
\r      0x000d (carriage return)
\t      0x0009 (horizontal tab)
\v      0x000b (vertical tab)

```

Um zum Beispiel den Text

```
file "C:\sample.txt"
```

als Zeichenkette darzustellen, muss man schreiben:

```
"file \"C:\\sample.txt\""
```

Daneben können wie in Namen auch Unicode-Konstanten (z.B. `\u0061`) verwendet werden (siehe Anhang A.4).

Wenn vor einer Zeichenkette das Zeichen `@` steht, dürfen darin Zeilenumbrüche vorkommen, `\` wird nicht als Escape-Zeichen interpretiert und das Hochkomma muss verdoppelt werden. Das obige Beispiel könnte man also auch so schreiben:

```
@"file ""C:\sample.txt"""
```

Ganze Zahlen. Ganze Zahlen können in Dezimalschreibweise (z.B. 123) oder in Hexadezimalschreibweise (z.B. `0x00a6`) vorkommen. Der Typ der Zahl ist der kleinste Typ aus `int`, `uint`, `long` oder `ulong`, zu dem der Zahlenwert passt. Durch Anhängen der Endung `u` oder `U` (z.B. `123u`) erzwingt man den kleinsten passenden vorzeichenlosen Typ (`uint` oder `ulong`), durch Anhängen von `l` oder `L` (z.B. `0x00a6l`) den kleinsten passenden Typ aus der Menge `long` und `ulong`.

Gleitkommazahlen. Gleitkommazahlen bestehen aus einem ganzzahligen Teil, einem Kommateil und einem Exponenten (`3.14E0` bedeutet z.B. $3.14 \cdot 10^0$). Jeder dieser Teile kann fehlen, aber zumindest einer davon muss vorkommen. Die Zahlen `3.14`, `314E-2` und `.314E1` sind also gültige Schreibweisen desselben Werts. Der Typ einer Gleitkommakonstante ist `double`, durch die Endung `f` oder `F` (z.B. `1f`) erzwingt man den Typ `float`, durch `m` oder `M` (z.B. `12.3m`) den Typ `decimal`.

Kommentare. Es gibt zwei Arten von Kommentaren: *Zeilenendekommentare* beginnen mit `//` und erstrecken sich bis zum Zeilenende, z.B.:

```
// ein Kommentar
```

Klammerkommentare beginnen mit `/*` und enden mit `*/`. Sie können sich auch über mehrere Zeilen erstrecken, dürfen aber nicht geschachtelt werden, z.B.:

```
/* ein Kommentar,
   der zwei Zeilen einnimmt */
```

Zeilenendekommentare werden für kurze Erläuterungen verwendet, Klammerkommentare meist zum Auskommentieren von Code.

2.4 Übungsaufgaben

1. Übersetzen und Ausführen eines Programms

Besorgen Sie sich das .NET-Framework (z.B. von [MS]) und installieren Sie es auf Ihrem Rechner. Öffnen Sie einen beliebigen Texteditor (z.B. *Notepad*). Tippen Sie das Hello-World-Programm aus Abschnitt 2.1 ab und speichern Sie es in einer Datei *Hello.cs*. Übersetzen Sie das Programm und führen Sie es aus.

2. Arbeiten mit einer Entwicklungsumgebung

Anstatt mit einem gewöhnlichen Texteditor zu arbeiten, können Sie auch eine Entwicklungsumgebung wie *Visual Studio .NET* verwenden, die jedoch im Gegensatz zum .NET-Framework nicht kostenlos ist. Eine kostenlose Entwicklungsumgebung ist zum Beispiel *SharpDevelop*, die von [SharpDev] geladen werden kann. Implementieren und übersetzen Sie das Hello-World-Programm mit dieser Entwicklungsumgebung.

3. Online-Dokumentation

Studieren Sie die Online-Dokumentation zu .NET, die Sie nach Installation des .NET-Frameworks unter *Start > Programs > Microsoft .NET Framework SDK > Documentation* finden. Suchen Sie zum Beispiel im Karteireiter *Contents* die Sprachspezifikation von C# (*Reference > Compiler and Language Reference > C#*). Um die Dokumentation einer bestimmten Klasse (z.B. *Console*) zu erhalten, wählen Sie den Karteireiter *Index* und geben im Suchfeld den Namen der Klasse ein.

4. Programme in mehreren Dateien

Tippen Sie die Klassen *Counter* und *Prog* aus Abschnitt 2.2 ab. Erstellen Sie daraus zwei getrennte Dateien *Counter.cs* und *Prog.cs* und übersetzen Sie sie wie in Abschnitt 2.2 beschrieben.

5. Symbole

Welche der folgenden Namen, Zahlen und Zeichenketten sind gemäß den Regeln von C# korrekt?

<i>Namen</i>	<i>Zahlen</i>	<i>Zeichenketten</i>
Length	0027	"one\ntwo\nthree"
base	0x3a	"\u00dcberschrift"
route66	520L	'Hello "John"'
_top	0xF3U	"Hello \"John\""
input-file	3E05	@ "Hello ""John"""
3pieces	.001	"quote /* " */ "