Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Golo Roden

Auf der Fährte von C#

Einführung und Referenz

Springer

Golo Roden
Carl-Kistner-Str. 17
79115 Freiburg im Breisgau

ISBN 978-3-540-27888-7 e-ISBN 978-3-540-27889-4

DOI 10.1007/978-3-540-27889-4

ISSN 1439-5428

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.

Einbandgestaltung: KünkelLopka, Heidelberg
Satz und Herstellung: le-tex publishing services oHG, Leipzig

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.com

In Liebe
für Susanna

Vorwort

Über dieses Buch

Warum *Auf der Fährte von C#*? Warum noch ein Buch zu dieser Programmiersprache?

Zunächst – dieses Buch ist grundlegend anders als andere verfügbare Literatur zu C#, und die Unterschiede liefern gleichzeitig auch die Begründung, warum man gerade dieses Buch lesen sollte. Doch worin bestehen die Unterschiede zwischen *Auf der Fährte von C#* und anderen Büchern?

Die Antwort auf diese Frage resultiert aus der Tatsache, dass andere verfügbare Texte zu C# nicht nur einen, sondern sogar den wesentlichen Aspekt der Anwendungsentwicklung missachten, nämlich den Aspekt, warum Anwendungen eigentlich geschrieben werden - nicht um der Anwendung, sondern um der Verarbeitung von Daten willen.

Andere Bücher, in denen Anwendungsentwicklung behandelt wird, gehen allerdings nicht datenzentrisch vor, sondern betonen statt dessen den Code, der im Grunde genommen nur Mittel zum Zweck ist. Der Aufbau von Datenstrukturen und das Denken in diesen werden – wenn überhaupt – lediglich flüchtig vermittelt oder gänzlich vernachlässigt.

Dieses zunächst unerwartete Vorgehen kann historisch begründet werden, denn den klassischen Programmiersprachen wie unter anderem C fehlen sprachliche Fähigkeiten, um damit datenzentrische Anwendungen entwickeln zu können. Erst moderne und durchgängig objektorientierte Sprachen wie beispielsweise Java und vor allem C# enthalten dieses Potenzial, doch anscheinend hat dieser Paradigmenwechsel die Literatur noch nicht erreicht.

Auf der Fährte von C# hingegen ist datenzentrisch aufgebaut, woraus ein untypischer Aufbau resultiert. Das zentrale Element datenzentrischer Anwendungen sind Typen, die in anderen Büchern in der Regel erst im weiteren Verlauf beschrieben werden. In diesem Buch machen sie nicht nur den Anfang, sondern bilden den grundlegenden Kern, auf dem alle weiteren Kapitel fußen.

Das Ziel des Ganzen ist, dass von Anfang an eine objekt- und datenorientierte Denkweise aufgebaut wird, da diese beiden Aspekte die entscheidende Basis für moderne und professionelle Anwendungsentwicklung darstellen.

Einen guten Programmierstil zu lehren und die Fähigkeit zu vermitteln, einen im mathematischen Sinne eleganten und dabei verständlichen, kommentierten und dokumentierten Code zu entwickeln, ist ein weiteres Ziel dieses Buches – ein Thema, das von den meisten anderen Autoren häufig ebenfalls vernachlässigt wird. So soll vermieden werden, dass sich bei Anfängern schlechte Angewohnheiten einschleichen, die im weiteren Verlauf mühsam wieder korrigiert werden müssen.

Auf diese Weise dauert es eventuell ein wenig länger bis zur ersten eigenen Anwendung, dafür verfügt man dann allerdings auch über fundiertes, begründetes Wissen und beherrscht die Thematik.

Zielgruppe

Ein Aspekt ist bei all dem besonders wichtig. Dieses Buch ist nämlich derart geschrieben, dass es so wohl von Anfängern wie auch von Fortgeschrittenen verwendet werden kann.

Entwickler, die noch keine oder nur sehr wenig Erfahrung in C# oder einer anderen Programmiersprache haben, können *Auf der Fährte von C#* als Lehrbuch nutzen, da großer Wert auf Verständlichkeit und ausführliche, detaillierte Erklärungen gelegt wird.

Zugleich kann es gerade wegen seines hohen Detailgrades fortgeschrittenen Entwicklern auf Dauer als verlässliche Referenz dienen, so dass man dieses Buch im Gegensatz zu vielen anderen nicht nach einer Weile ad acta legt, sondern beständig Nutzen aus ihm ziehen kann.

Auf Grund dieser Dualität ist *Auf der Fährte von C#* abstrakter und anspruchsvoller als andere Texte, was für Anfänger partiell durchaus eine Herausforderung darstellen kann – dafür ist das Resultat allerdings auch ertragreicher.

Struktur

Im ersten Teil des Buches folgen nach einer kurzen Einführung in die Themen .NET, C# und allgemeine Konzepte der Anwendungsentwicklung als eigentlicher Anfang – wie bereits erwähnt – die Typen, auf denen anschließend die einzelnen Datenstrukturen aufbauen.

Danach werden Variablen, Operatoren und Anweisungen beschrieben, womit dann bereits einfache Anwendungen entwickelt werden können. Weitergehende Themen wie Nebenläufigkeit, Fehlerbehandlung und die Speicherverwaltung von .NET runden den ersten Teil schließlich ab.

Dabei werden zusammen mit den Konzepten und der Sprache C# auch zugleich der Umgang mit den Werkzeugen von .NET, erprobte Praktiken und guter Programmierstil vermittelt.

Den zweiten Teil bildet eine alphabetisch geordnete Referenz aller Schlüsselwörter von C#. Für jeden einzelnen Eintrag stehen neben allgemeinen und detaillierten Informationen ein repräsentatives Codebeispiel und Verweise auf weiterführende Themen zur Verfügung.

Die verwendeten Fachbegriffe entsprechen nach Möglichkeit den deutschsprachigen Ausdrücken. Für den Fall der Fälle befindet sich im Anhang eine Liste der gängigen englischsprachigen Entsprechungen.

Kontakt

Ergänzt wird dieses Buch durch die Webseite guide to C# (http://www.guidetocsharp.de), auf der sich das komplette Buch als E-Book sowie Aktualisierungen und Errata finden. Zudem kann diese Version auch überall und immer dann genutzt werden, wenn man das Buch nicht mit sich führt.

Bei Fragen, Anregungen, Lob oder Kritik erreichen Sie den Autor über seine Webseite goloroden.de (http://www.goloroden.de).

Danken möchte ich meiner Frau Susanna und meinen Eltern Magda und Wilfried für ihre beflügelnde und hilfreiche Unterstützung und für ihre kreativen Ideen. Außerdem möchte ich den Springer-Verlag, Microsoft und myCSharp.de dankend erwähnen, ohne die es *Auf der Fährte von C#* in dieser Form nicht gäbe.

Zu guter letzt – ich widme dieses Buch meiner Frau Susanna, der einen großen Liebe meines Lebens.

Norddorf auf Amrum, im März 2007

Kapitelübersicht

Inhalt

In *Auf der Fährte von C#* lernen Sie die Entwicklung von Anwendungen in C#. Die einzelnen Kapitel bauen aufeinander auf, sind aber derart gestaltet, dass sie auch als Referenz genutzt werden können. Nutzen Sie diese Kapitelübersicht, um sich einen Überblick über den Aufbau von *Auf der Fährte von C#* zu verschaffen.

In Kapitel 1 *Einführung in .NET* werden die Begriffe .NET und C# detailliert erläutert. .NET wird zu anderen Plattformen zur Softwareentwicklung abgegrenzt, zudem werden die einzelnen Komponenten und Konzepte von .NET erläutert. Schließlich wird die speziell für .NET entwickelte Sprache C# vorgestellt.

In Kapitel 2 *Anwendungsarchitektur* wird der Aufbau moderner Anwendungen beschrieben, die so wohl objekt- wie auch komponentenorientiert entwickelt werden. Die in diesem Kapitel vorgestellten Begriffe bilden den Rahmen zur Einordnung der weiteren Themen.

In Kapitel 3 *Objektorientierung* werden die grundlegenden Konzepte der objektorientierten Programmierung beschrieben. Außerdem werden das prozedurale und das objektorientierte Paradigma gegenübergestellt und hinsichtlich ihrer Herkunft und ihrer Leistungsfähigkeit verglichen.

In Kapitel 4 *Typen* werden Werte- und Verweistypen gegenübergestellt und ihre jeweiligen Besonderheiten beschrieben. Eine Einführung in nullbare Wertetypen und eine Übersicht über die vordefinierten Typen runden das Kapitel ab.

In Kapitel 5 *Namensräume* werden Namensräume als Konzept vorgestellt, das zur Organisation von Typen dient. Neben Aliasnamen für Namensräume wird außerdem beschrieben, wie eigene Namensräume definiert werden können.

In Kapitel 6 *Klassen und Strukturen* wird beschrieben, wie Klassen und Strukturen erstellt werden. Insbesondere werden in diesem Kapitel Felder, Eigenschaften und Methoden eingeführt. Außerdem werden Konstanten und readonly-Variablen vorgestellt.

In Kapitel 7 *Vererbung* werden die objektorientierten Konzepte Vererbung und Polymorphie beschrieben. Außerdem werden Versionierung von Methoden mit Hil-

fe von Vererbung und die Auswirkungen von Vererbung auf Typmitglieder dargestellt.

In Kapitel 8 *Schnittstellen* werden Schnittstellen als grundlegendes Konzept bei der Anwendungsentwicklung vorgestellt. Außerdem wird aufgezeigt, auf welche Arten Schnittstellen implementiert werden können.

In Kapitel 9 *Delegaten* werden Delegaten vorgestellt, um eine oder mehrere Methoden zu kapseln. Neben der Bindung an benannte Methoden werden insbesondere auch anonyme Methoden beschrieben.

In Kapitel 10 *Ereignisse* wird beschrieben, wie Ereignisse implementiert werden und welche Aspekte es dabei zu beachten gilt. Besonderes Augenmerk wird dabei auf den Zusammenhang zwischen Ereignissen und Delegaten gelegt.

In Kapitel 11 *Generika* wird die Möglichkeit vorgestellt, Typen generisch zu implementieren oder generische Typen als Typparameter zu verwenden. Außerdem wird beschrieben, welche Typbedingungen bei generischen Typen zum Einsatz kommen können, um die Ausführung typsicher zu machen.

In Kapitel 12 *Nullbare Wertetypen* werden nullbare Wertetypen als ein Hybrid aus Werte- und Verweistyp vorgestellt. Mit nullbaren Wertetypen ist es möglich, die Vorteile des Literals null auch für Wertetypen zu nutzen.

In Kapitel 13 *Enumerationen* werden Enumerationen als einfache Variante eines wertebasierten Aufzählungstyps vorgestellt. Neben der Implementierung wird auch die interne technische Umsetzung beschrieben.

In Kapitel 14 *Variablen* wird das Deklarieren und Definieren von Variablen beschrieben. Neben der Zuweisung an sich werden auch die Besonderheiten der Zuweisung an nullbare Wertetypen und an Verweistypen erläutert. Außerdem wird die Instanziierung von Objekten erläutert.

In Kapitel 15 *Arrays* wird das Konzept der Arrays vorgestellt, um Mengen von gleich typisierten Daten zu speichern. Dabei werden so wohl ein- und mehrdimensionale wie auch verschachtelte Arrays beschrieben. Außerdem wird aufgezeigt, wie mit Arrays Indexer für Klassen umgesetzt werden können.

In Kapitel 16 *Operatoren* werden die diversen Operatoren vorgestellt – arithmetische, relationale, logische, bitweise und Zeichenkettenoperatoren. Außerdem wird auf verkürzende Schreibweisen und Operatorüberladung eingegangen.

In Kapitel 17 *Ausdrücke* werden implizites und explizites Konvertieren sowie die Implementierung eigener Konvertierungsoperatoren vorgestellt, um Typen ineinander umzuwandeln.

In Kapitel 18 *Anweisungen* werden die verschiedenen Arten von Anweisungen wie Bedingungen und Schleifen vorgestellt. Außerdem wird die Schnittstelle IEnumerator als Basis für sämtliche Aufzählungstypen beschrieben und deren Umsetzung mit der foreach-Anweisung.

In Kapitel 19 *Linq* wird die in C# seit der Version 3.0 enthaltene Abfragesprache behandelt, mit der Aufzählungstypen durchsucht, sortiert und gruppiert werden können.

In Kapitel 20 *Ausnahmen* werden Ausnahmen und die Möglichkeiten zur Ausnahmebehandlung vorgestellt. Außerdem werden Leistung und Ressourcenbedarf von Ausnahmen beschrieben.

In Kapitel 21 *Attribute* wird beschrieben, wie Attribute entwickelt werden. Außerdem werden Attributparameter und -ziele vorgestellt.

In Kapitel 22 *Speicherverwaltung* werden Destruktoren und die Speicherbereinigung beschrieben. Außerdem werden Interning von Zeichenketten und Konzepte zur verspäteten Initialisierung vorgestellt.

Inhaltsverzeichnis

Kapitel 1
Einführung in .NET

1.1 Was ist .NET?

.NET ist eine Plattform von Microsoft zur Entwicklung und Ausführung von Anwendungen.

Da sich am Ende des vergangenen Jahrtausends zahlreiche Plattformen mit ihren jeweiligen Konzepten und Standards verbreitet hatten, wurde die Entwicklung von Anwendungen für Windows zunehmend komplexer und damit anspruchsvoller.

Nur die Windows-eigene Basis namens Win32 zu kennen, war bei weitem nicht mehr ausreichend, man musste sich zusätzlich mit COM, DCOM, Windows DNA, DirectX, ASP, ISAPI, VBA, WSH und zahlreichen anderen Technologien beschäftigen, um zeitgemäße Anwendungen entwickeln zu können.

Die Idee hinter .NET war, eine einheitliche und konsistent objektorientierte Plattform zu schaffen, die alle bestehenden Konzepte vereint. Insofern beerbt .NET gewissermaßen die genannten Plattformen, auch wenn diese – allein schon aus Gründen der Abwärtskompatibilität – zumindest vorerst weiterhin zur Verfügung stehen werden.

Um trotz dieser Revolution einen sanften Übergang zu ermöglichen, ist es möglich, .NET wie COM anzusprechen, und COM im Gegenzug aus .NET heraus zu nutzen. Daher kann .NET durchaus als eine Revolution mit evolutionärem Rahmenwerk bezeichnet werden.

1.2 Plattformunabhängigkeit

Die Grundlage von .NET bildet die Common Language Infrastructure – im folgenden als CLI abgekürzt –, eine Spezifikation, welche die plattform- und sprachunabhängige Entwicklung und Ausführung von Anwendungen beschreibt.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Die CLI wurde im August 2000 von Microsoft, Intel und Hewlett Packard bei der ECMA zur Standardisierung eingereicht und ein Jahr später, im Dezember 2001, unter dem Namen ECMA-335 als Standard verabschiedet. Da es sich bei der CLI um einen frei verfügbaren Standard handelt, kann potenziell von jedem Interessierten eine eigene Implementierung entwickelt werden.

.NET ist dabei die bekannteste und am weitesten verbreitete Implementierung der CLI, wobei es außer .NET an sich noch weitere Varianten gibt, die ebenfalls von Microsoft stammen: Das .NET Compact Framework zur Nutzung auf mobilen Geräten, das .NET Micro Framework für eingebettete Geräte und schließlich Rotor unter FreeBSD als Studie einer CLI-Implementierung auf einer anderen Plattform als Windows.

Die erste Version von .NET erschien am 13. Februar 2002 in Verbindung mit der dazugehörigen integrierten Entwicklungsumgebung – Visual Studio .NET. Bereits ein Jahr später folgten .NET 1.1 und Visual Studio .NET 2003, wobei diese Aktualisierungen neben einigen wenigen technischen Änderungen im wesentlichen Sicherheitsaktualisierungen enthielten.

Zudem war .NET 1.1 die erste Version von .NET, die nicht nur als zusätzliche Komponente zu Windows ausgeliefert wurde – diese Version ist standardmäßig in Windows Server 2003 enthalten.

Version 2.0 erschien weitere zweieinhalb Jahre später zusammen mit Visual Studio 2005 am 7. November 2005. Neben zahlreichen technischen Änderungen und Verbesserungen ist bemerkenswert, dass der Begriff .NET nicht mehr im Produktnamen von Visual Studio enthalten ist.

Am 6. November 2006 schließlich wurde .NET 3.0 veröffentlicht, das im Gegensatz zu Version 2.0 keine eigenständige Version im klassischen Sinne darstellt, sondern im Kern nach wie vor auf .NET 2.0 basiert und dieses lediglich um vier Komponenten erweitert: Die Windows Presentation Foundation, die Windows Communication Foundation, die Windows Workflow Foundation und Windows Card Space.

Wie bereits .NET 1.1 wird auch .NET 3.0 nicht nur als zusätzliche Komponente zu Windows ausgeliefert, sondern ist bereits in Windows Vista integriert.

Am 27. Februar 2008 wurde .NET 3.5 zusammen mit Visual Studio 2008 veröffentlicht. Wie bereits .NET 3.0 setzt auch .NET 3.5 auf der Version 2.0 von .NET auf und erweitert dieses um weitere Komponenten. Besonders hervorzuheben sind dabei die neue Version 3.0 von C# und eine in C# enthaltene integrierte Abfragesprache namens Linq.

Neben diesen Implementierungen der CLI durch Microsoft gibt es einige weitere Implementierungen, von denen vor allem Mono von Novell nennenswert ist. Mono war ursprünglich von Miguel de Icaza und dessen Firma Ximian entwickelt worden, die allerdings am 4. August 2003 von Novell übernommen wurde.

Am 30. Juni 2004 schließlich wurde Mono in Version 1.0 veröffentlicht und stellt seitdem eine interessante Alternative zu .NET dar, da es zum einen nicht nur Windows, sondern auch zahlreiche andere Betriebssysteme wie unter anderem Linux und Mac OS X unterstützt, und zum anderen als Opensource verfügbar ist.

1.3 Sprachunabhängigkeit

Die CLI beschreibt jedoch nicht nur die plattform-, sondern auch die sprachunabhängige Entwicklung und Ausführung von Anwendungen. Sprachunabhängig bedeutet dabei, dass es keine Rolle spielt, in welcher Programmiersprache eine Anwendung geschrieben wurde.

Für die Entwicklung unter Win32 gab es essenzielle Unterschiede zwischen den verschiedenen Programmiersprachen. Konnte beispielsweise COM aus Visual C++ uneingeschränkt genutzt werden, war dies in Visual Basic nur eingeschränkt möglich. Außerdem unterschieden sich die verschiedenen Sprachen in ihren jeweiligen Konventionen, so dass die erzeugten Anwendungen auch diesbezüglich nicht immer vollständig kompatibel zueinander waren.

Zudem war es nicht möglich, in verschiedenen Sprachen geschriebene Komponenten beliebig miteinander zu kombinieren. Das Resultat dieser Inkompatibilitäten war, dass Komponenten häufig in mehreren programmiersprachenspezifischen Varianten zur Verfügung standen.

In .NET ist die Kombination von Komponenten und Anwendungen hingegen uneingeschränkt möglich, was im wesentlichen der Verdienst der beiden wichtigsten Komponenten von .NET ist: Der Common Language Runtime – im folgenden als CLR abgekürzt – und der Framework Class Library – im folgenden als FCL abgekürzt.

Die CLR dient zur Ausführung von Anwendungen, wobei diese Anwendungen in Sprachen entwickelt worden sein müssen, die .NET als Zielplattform adressieren. All diesen Sprachen ist gemein, dass sie einen Unterstandard der CLI implementieren, nämlich das Common Language Subset – im folgenden als CLS abgekürzt. Das CLS beschreibt Eigenschaften, die zur CLR, und damit zu .NET, kompatible Sprachen aufweisen müssen.

Außerdem werden die erzeugten Anwendungen nicht – wie in klassischen Programmiersprachen – in Maschinensprache umgewandelt, die direkt vom Prozessor ausgeführt werden kann. Statt dessen wird eine für .NET spezifische Zwischensprache eingesetzt, die sogenannte Microsoft Intermediate Language – im folgenden als MSIL abgekürzt.

Erst zur Ausführungszeit werden die MSIL-Anweisungen in Maschinensprache umgesetzt, die dann auf die jeweils ausführende Hardwareplattform optimiert werden kann. Dies geschieht durch einen Compiler, der „just in time“ arbeitet, also erst auf Anforderung und nur das jeweils Notwendige übersetzt, und daher auch als JIT-Compiler bezeichnet wird.

Da die Übersetzung während der Ausführung stattfindet, dauert der erste Aufruf einer .NET-Anwendung naturgemäß ein wenig länger als bei Anwendungen, die in klassischen Programmiersprachen geschrieben wurden. Die Optimierung des JIT-Compilers gleicht dies aber aus, so dass JIT-übersetzte Anwendungen in der Regel schneller ausgeführt werden können.

Die FCL – Framework Class Library – schließlich stellt eine Klassenbibliothek zur Verfügung, die einige tausend Klassen für häufig auftretende Aufgaben enthält und aus allen .NET-spezifischen Sprachen heraus genutzt werden kann. Da die FCL

von der CLR bereitgestellt wird, bleiben Anwendungen für .NET trotz einem potenziell hohen Funktionsumfang verhältnismäßig kompakt, da diese die FCL nicht enthalten müssen.

1.4 Verwalteter Code

Außer der Anpassung an die ausführende Hardwareplattform hat die Verwendung einer Zwischensprache noch einen weiteren Grund. Die CLR kann nämlich die auszuführenden MSIL-Anweisungen vor der Übersetzung in Maschinensprache analysieren und potenziell eingreifen. Da der Code zur Ausführungszeit kontrolliert wird, bezeichnet man ihn als verwalteten Code.

Auf diese Weise kann die CLR unter anderem sicherstellen, dass Anwendungen nur auf Speicher zugreifen, auf den sie an dieser Stelle zugreifen dürfen. Zudem kann überprüft werden, auf welche Ressourcen eine Anwendung versucht zuzugreifen, wobei dies bei fehlender Berechtigung verhindert werden kann.

So kann Anwendungen, die aus nicht vertrauenswürdigen Quellen wie dem Internet stammen, zwar die prinzipielle Ausführung erlaubt, der Zugriff beispielsweise auf das Dateisystem aber verweigert werden. Diese Einschränkung des Zugriffs auf Ressourcen zur Laufzeit wird als Code Access Security – im folgenden als CAS abgekürzt – bezeichnet.

Schließlich führt die CLR von Zeit zu Zeit eine Speicherbereinigung durch, entfernt Code und Daten, die nicht mehr benötigt werden, und gibt damit wieder Speicher zur weiteren Verwendung frei. Diese Speicherbereinigung wird als Garbage Collection bezeichnet und im folgenden als GC abgekürzt.

Zusammengefasst gilt bei .NET also, dass Sicherheit höher priorisiert wird als eine möglichst schnelle Ausführung von Anwendungen um jeden Preis.

1.5 Erweiterungen

Zu diesem Grundgerüst von .NET, das aus der Common Language Runtime und der Framework Class Library besteht, gibt es einige Erweiterungen, die erwähnenswert sind. Diese gliedern sich im wesentlichen in vier Kategorien: Windowsanwendungen, Webanwendungen, Kommunikation und Datenverwaltung.

Für Windowsanwendungen sind zunächst GDI+ und Windows Forms zu nennen. Während GDI+ den objektorientierten und verwalteten Nachfolger der Grafikbibliothek GDI darstellt, lassen sich mit Windows Forms grafische Oberflächen mit den gängigen Steuerelementen erstellen.

Im Rahmen von .NET 3.0 wurde die Windows Presentation Foundation – im folgenden als WPF abgekürzt – eingeführt, die statt eines pixelorientierten Ansatzes einen vektororientierten Ansatz verfolgt und zudem über die auf XML basierende Sprache XAML genutzt werden kann.

Webanwendungen und Web Services werden in .NET mit Hilfe von ASP.NET umgesetzt, das den objektorientierten und verwalteten Nachfolger von klassischem ASP darstellt. Mit dem in Windows Vista enthaltenen Webserver IIS 7.0 erfährt ASP.NET zudem eine direkte Integration in den Webserver.

Zur Kommunikation mit anderen Anwendungen gibt es in .NET zahlreiche Möglichkeiten – von Remoting über Web Services bis hin zur Microsoft Message Queue. Mit .NET 3.0 wurde als weitere Komponente die Windows Communication Foundation – im folgenden als WCF abgekürzt – eingeführt, die alle bisherigen Konzepte kapselt und mit einer einzigen einheitlichen Schnittstelle versieht.

Zugriff auf Datenbanken und sonstige Ressourcen geschieht in .NET mit Hilfe von ADO.NET, das wiederum den Nachfolger von DAO und ADO darstellt. Mit der Version 3.0 der Sprache C# wurden zudem Funktionen zur Datenabfrage unter dem Namen Linq direkt in die Programmiersprache integriert.

Schließlich gibt es noch zwei Komponenten, die sich nicht in die genannten vier Kategorien eingliedern lassen, sondern eigenständig für sich stehen. Zum einen ist dies die Windows Workflow Foundation – im folgenden als WF abgekürzt –, mit der sich Workflows in .NET gestalten lassen, zum anderen Windows Card Space – im folgenden als WCS abgekürzt –, das zur Verwaltung digitaler Identitäten dient.

1.6 Was ist C#?

C#, das als „ßiescharp [ˌsiːˈʃɑɹp]“ ausgesprochen wird, ist eine Programmiersprache für .NET, die von Microsoft in Zusammenarbeit mit dem Erfinder von Delphi, Anders Heijlsberg, speziell für diese Plattform entwickelt wurde und daher auch als Lingua Franca für .NET bezeichnet wird. Wie die CLI wurde auch C# von der ECMA standardisiert, wobei die Sprache den Namen ECMA-334 trägt.

Der Name von C# lehnt sich in seiner Schreibweise an den in der Musik um einen Halbton erhöhten Notenwert C namens Cis an, der ebenfalls als C# geschrieben wird, und bezeichnet daher eine höhere Variante der Programmiersprache C. Außerdem kann C# so wohl als symbolische Anspielung an die Sprache C++ wie auch als Wortspiel „see sharp“ gesehen werden.

Ein Ziel bei der Entwicklung von C# war nicht nur, eine sich perfekt in .NET einfügende Sprache zu schaffen, sondern diese Sprache modern, durchgängig objekt- wie auch komponentenorientiert, und vor allem verständlich zu gestalten.

Unter Win32 waren Visual C++ und Visual Basic die gängigen Sprachen zur Entwicklung von Anwendungen für Windows, allerdings weisen beide gravierende Nachteile auf. Mit Visual C++ lassen sich zwar alle Möglichkeiten von Windows ausreizen, und es kann uneingeschränkt auf COM zugegriffen werden, insbesondere für Anfänger ist es allerdings auf Grund einiger ausgefallener Eigenheiten deutlich zu komplex.

Im Gegenzug dazu ermöglicht Visual Basic einen sehr einfachen Einstieg, bietet aber bei weitem nicht die Möglichkeiten von Visual C++, ist zudem bezogen

auf COM nur eine Sprache zweiter Klasse und zeichnet sich vor allem durch die niedrige Ausführungsgeschwindigkeit der erzeugten Anwendungen aus.

Schließlich sind Visual C++ und Visual Basic auch untereinander nicht ohne weiteres kompatibel, es gibt weder ein gemeinsames Typsystem noch ein einheitliches System zur Fehlerbehandlung, so dass Komponenten, die in der einen Sprache entwickelt wurden, nicht notwendigerweise in der anderen weiter genutzt werden können.

Dank dem Common Language Subset verfügen alle Sprachen unter .NET potenziell über die gleichen Fähigkeiten, das heißt, es gibt unter .NET keine Sprachen erster und zweiter Klasse, wie dies unter Win32 der Fall war. Die verschiedenen Sprachen für .NET unterscheiden sich daher zum einen syntaktisch – so hat C# mit Visual Basic .NET nicht viel gemein –, zum anderen legen sie ihre Schwerpunkte auf unterschiedliche Aspekte der Entwicklung und sprechen daher verschiedene Zielgruppen an.

Im Vergleich zu Visual Basic .NET ist C# die abstraktere, kompaktere und mathematisch elegantere Sprache, lässt einigen syntaktischen Zucker außen vor, und ermöglicht daher eher, sich auf die wesentlichen Aspekte der zu entwickelnden Anwendung – die Datenstrukturen und die Algorithmen – zu konzentrieren.

Kapitel 2
Anwendungsarchitektur

2.1 Lösungen und Anwendungen

Warum werden Anwendungen entwickelt? Die Beantwortung dieser Frage setzt voraus, dass eine Definition des Begriffs Anwendung vorliegt, auf die man sich beziehen kann.

In erster Linie werden Anwendungen entwickelt, um Aufgaben einer spezifischen Domäne zu erledigen und die damit einhergehenden Probleme zu lösen. Daher wäre es eigentlich angebracht, an Stelle von Anwendungsentwicklung von der Entwicklung von Lösungen zu sprechen. Dennoch wird im alltäglichen Sprachgebrauch der Begriff der Anwendung häufig synonym mit dem der Lösung verwendet.

Dabei darf allerdings nicht vergessen werden, dass eine moderne Anwendung – im Sinne einer Lösung – sehr wohl aus mehr als nur einer Anwendung – im Sinne einer ausführbaren Anwendung – bestehen kann.

Einerseits kann nämlich bereits die eigentliche Funktionalität auf mehrere Anwendungen verteilt werden, wie es beispielsweise in Umgebungen mit einer Server- und mehreren Client-Anwendungen der Fall ist. Andererseits wird gerade für Webanwendungen häufig eine zusätzliche Windowsanwendung zur Wartung entwickelt.

Außerdem nutzen die meisten modernen Anwendungen eine Reihe unterstützender Anwendungen, wie eine Datenbank, einen Webserver und sonstige Services, die zwar nicht direkt integriert sind, aber dennoch – wie die eigentliche Anwendung – ihren jeweiligen Teil zur übergeordneten Lösung beitragen.

Insofern sind Anwendungen genau genommen autonome Bestandteile einer Lösung und kommunizieren miteinander über definierte Schnittstellen und Kanäle. Dennoch werden im folgenden die Begriffe Anwendung und Lösung synonym verwendet, da die in diesem Buch entwickelten Lösungen zumeist nur aus einer einzelnen Anwendung bestehen.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

2.2 Komponenten und Assemblies

Die hierarchische Struktur einer Lösung mit mehreren untergeordneten Anwendungen besteht vergleichbar auch innerhalb einer einzelnen Anwendung. Waren diese früher häufig monolithisch – also aus einem Guss – aufgebaut, werden heutzutage in der Regel eine Reihe von eigenständigen Komponenten entwickelt, die anschließend miteinander integriert und zu einer Anwendung verschmolzen werden.

Komponenten zeichnen sich vor allem durch zwei Aspekte aus: Zum einen sind sie autarke, in sich abgeschlossene Einheiten, die nicht von anderen Einheiten abhängen. Auf Grund dessen kann die Entwicklung einer komponentenorientierten Anwendung problemlos innerhalb eines Teams aufgeteilt werden. Zum anderen verfügt jede Komponente über eine definierte Schnittstelle, über die sie von außen angeprochen werden kann. Um eine Komponente in eine Anwendung einzubinden, muss nur ihre Schnittstelle, nicht aber ihr innerer Aufbau bekannt sein, so dass eine Komponente als Blackbox fungiert.

Außerdem sind verschiedene Komponenten untereinander austauschbar, sofern sie die gleiche Schnittstelle bereitstellen. Dies erleichtert nicht nur die Wartbarkeit von Anwendungen, da im Nachhinein problemlos einzelne Komponenten an Stelle der ganzen Anwendung ausgetauscht werden können. Zudem wird auch die Testbarkeit verbessert, da bei einer Änderung in einer Komponente nur diese erneut getestet werden muss, nicht aber die vollständige Anwendung.

Auf Grund dieser Eigenschaften spielen Komponenten eine essenzielle Rolle in der modernen, teambasierten Anwendungsentwicklung.

Eine Komponente besteht – wie bereits Lösungen und Anwendungen – wiederum aus kleineren Bestandteilen, den Assemblies. Eine Assembly ist dabei ein Konzept, das von Microsoft in .NET neu eingeführt wurde. Diese wird physisch von einer Datei mit der Endung .dll oder .exe repräsentiert, wobei eine Assembly außer der Dateiendung nicht viel mit einer klassischen Datei mit der selben Endung gemein hat.

Eine Assembly enthält in erster Linie ausführbare Anweisungen in MSIL. Zudem kann sie aber noch Ressourcen enthalten, die zur ihrer Ausführung benötigt werden, wie beispielsweise Bilder oder Klänge. Erzeugt wird eine Assembly aus einer oder mehreren Dateien, die Anweisungen in C# enthalten, und zusätzlich aus den einzubindenden Ressourcendateien.

Außerdem verfügt eine Assembly über Metadaten, die ihren Inhalt näher beschreiben. So werden nicht nur Angaben zu den enthaltenen Anweisungen und Ressourcen gemacht, sondern auch Informationen über die Assembly an sich wie beispielsweise ihr Hersteller und ihre Versionsnummer bereitgestellt.

2.3 Datentypen

Die Anweisungen in einer Assembly, die ursprünglich in C# geschrieben wurden, sind ihrerseits allerdings noch einmal in sogenannten Typen organisiert, und zwar gemeinsam mit den Datenstrukturen, die sie zu ihrer Ausführung benötigen. Das

Entwerfen und Organisieren dieser Typen ist die wesentliche Arbeit, die bei der Entwicklung einer Anwendung in C# geleistet wird.

Zusammenfassend heißt das auf der einen Seite, dass alles, was sich oberhalb einer Assembly befindet, zwar zu der Ausführung einer Anwendung benötigt wird, aber letztlich auch nur die Ausführung sowie die Verteilung der Anwendung betrifft. Insbesondere wird all dies nicht in C# umgesetzt, da dort nur noch ausführbare Dateien gehandhabt werden, und man, wenn überhaupt, nur mit MSIL – nicht aber mit C# – in Berührung kommt.

Auf der anderen Seite wird im Gegenzug alles, was sich unterhalb einer Assembly befindet, in C# umgesetzt. Das heißt, Anwendungen zu entwickeln bedeutet im wesentlichen, Datenstrukturen zu modellieren, sie zu organisieren und darauf basierend die Verfahren aufzubauen, die mit diesen Datenstrukturen arbeiten.

Kapitel 3
Objektorientierung

3.1 Prozedurales Paradigma

In den vergangenen Kapiteln wurde C# als objektorientierte Sprache bezeichnet, wobei dieser Begriff bislang noch nicht näher erläutert wurde.

Um das Konzept der Objektorientierung zu verstehen, muss man wissen, wie Programmiersprachen früher aufgebaut waren. Im vorherigen Kapitel wurde die Entwicklung von Anwendungen im wesentlichen als Modellierung und Organisation von Datenstrukturen und als Aufbau der entsprechenden datenverarbeitenden Verfahren bezeichnet.

Weil Daten ein dermaßen grundlegendes Element für die Entwicklung von Anwendungen darstellen, kann de facto keine Anwendung ohne Daten bestehen. Damit diese aber überhaupt verarbeitet werden können, müssen sie der Anwendung zunächst als Eingabe vorliegen. Zudem wird eine Ausgabemöglichkeit benötigt, um die verarbeiteten Daten an den Benutzer zurückzugeben.

Diese Aspekte jeder Anwendung – das ursprüngliche Einlesen der zu verarbeitenden Daten, die eigentliche Verarbeitung und schließlich die Ausgabe von Ergebnissen – werden in der Informatik als EVA-Prinzip bezeichnet, wobei EVA als Akronym für Eingabe, Verarbeitung und Ausgabe steht.

Allerdings darf dabei nicht vergessen werden, dass verschiedene Arten von Daten bestehen, die dementsprechend auch unterschiedlich modelliert und gehandhabt werden müssen. Zudem muss dieser Diversität in Programmiersprachen Rechnung getragen werden, weshalb es dort sogenannte Typen gibt, die jeweils eine spezielle Art von Daten repräsentieren.

In klassischen Programmiersprachen gibt es nur einfache Typen, um elementare Daten aufzunehmen, wie beispielsweise Typen zur Speicherung einer Ganzzahl, einer Dezimalzahl oder eines Buchstaben. Die Verwendung nur dieser einfachen Typen reicht bereits aus, um alle möglichen Daten abzubilden, allerdings ist dafür teilweise eine aufwändige Umwandlung komplexer Daten in einfache Typen notwendig.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Neben der potenziell aufwändigen Umwandlung birgt dieses Verfahren noch einen weiteren essenziellen Nachteil: Ein komplexer Typ wird nicht als komplexer Typ verarbeitet, sondern als Menge von einfachen Typen. Dass diese einfachen Typen gemeinsam einen komplexen Typ darstellen, kann nicht explizit abgebildet werden, statt dessen ist dieses Wissen nur noch implizit vorhanden.

Ein komplexer Typ, wie beispielsweise eine Person, der auf die einfachen Typen Nachname, Vorname und Alter abgebildet wird, ist demzufolge in der Anwendung nicht mehr als komplexer Typ vorhanden, sondern er besteht nur noch in den eigenständigen einfachen Typen für Nachname, Vorname und Alter. Die Entscheidung, diese einfachen Typen wieder zu einem komplexen Typ zusammenzufügen, obliegt dem Entwickler.

Außerdem werden einzelne Typen in klassischen Programmiersprachen nicht voneinander abgeschirmt, das heißt, jeder Teil der Anwendung kann auf sämtliche Daten unbeschränkt zugreifen. Dass dies nicht nur die gezielte, sondern auch die unbeabsichtigte Veränderung von Daten ermöglicht und damit potenziell eine Quelle für zahlreiche, aber nicht offensichtliche Fehler darstellt, liegt auf der Hand.

Aus all diesen Gründen liegt der Schwerpunkt bei der Entwicklung von Anwendungen in den klassischen Programmiersprachen auf dem Code, der die Typen und damit die Daten verarbeitet, und nicht auf den Typen selbst. Da dabei häufig benötigte Codezeilen in Funktionen – sogenannten Prozeduren – zusammengefasst werden, wird diese Art der Entwicklung als prozedurales Paradigma bezeichnet.

3.2 Objektorientiertes Paradigma

Für die Entwicklung komplexer Anwendungen ist das prozedurale Paradigma allerdings nicht tragfähig und stößt schnell an seine Grenzen. Die wesentlichen Probleme dabei sind neben den ungenügenden Typen und den fehlerhaften und ungültigen Zugriffen zumeist das redundante Zurverfügungstellen von Daten an verschiedenen Stellen der Anwendung sowie häufig eine nicht überschaubare Komplexität.

Betrachtet man die reale Welt, dann fällt auf, dass man in der Regel mit komplexen Typen in Berührung kommt, deren Komplexität deutlich über der einer Zahl oder eines Buchstaben liegt. Die in den klassischen Programmiersprachen gängigen einfachen Typen treten hingegen nur selten eigenständig auf.

Die Objekte der realen Welt haben wie komplexe Typen gewisse Eigenschaften, verfügen zudem aber noch über Aktionen, die sie ausführen können. So verfügt eine Person als Objekt neben den bereits erwähnten Daten wie Nachname, Vorname und Alter auch über Aktionen, wie beispielsweise laufen, lesen und schlafen. Auf diese Art, nämlich die Zweiteilung in Eigenschaften und Aktionen, kann fast jedes Objekt hinreichend beschrieben werden.

Der entscheidende Punkt bei dieser Darstellung ist, dass jedes Objekt für sich über die Information verfügt, welche Aktionen es in einem gegebenen Kontext ausführen kann – es handelt autonom. Außerdem sind die Eigenschaften eines Objek-

tes von außen nicht einsehbar, denn um Daten zu erhalten, muss eine entsprechende Anfrage gestellt werden, die einer Aktion entspricht.

Dieses Prinzip, die internen Daten zu verbergen und nach außen nur über kontrollierte Aktionen zur Verfügung zu stellen, wird als Information Hiding bezeichnet und ist eine weitere wesentliche Voraussetzung für die Autonomie von Objekten. Diese Autonomie wiederum ist die Basis für die Entwicklung komponentenorientierter Anwendungen.

Der komplexe Typ, der einem Objekt zu Grunde liegt, wird dabei als Klasse bezeichnet. Im Gegensatz zum Objekt, das eine konkrete Ausprägung darstellt, repräsentiert die Klasse lediglich den Bauplan für dieses Objekt. Daher kann es zwar zahlreiche Objekte geben, die auf der selben Klasse basieren, allerdings hat jedes Objekt nur genau eine Klasse, die den Aufbau des Objektes beschreibt.

Die Klasse Person definiert also, welche Eigenschaften und Aktionen eine konkrete Person auszeichnen – sie stellt aber keine konkrete Person dar, ebenso wenig wie ein Bauplan eines Hauses ein konkretes Haus darstellt. Erst ein Objekt der Klasse Person repräsentiert eine konkrete Person. Da ein Objekt also eine konkrete Ausprägung darstellt, wird es auch als Instanz einer Klasse und seine Erzeugung als Instanziierung bezeichnet.

Im Unterschied zum prozeduralen Paradigma ist der Code, der die Objekte verarbeitet, in den Objekten selbst enthalten, weshalb auf Klassen und Objekten basierende Anwendungen deutlich weniger codezentrisch aufgebaut sind als klassische Anwendungen. Statt dessen wird sehr viel mehr Augenmerk auf die Modellierung und Organisation der notwendigen Klassen und der Verbindungen zwischen ihnen gelegt.

Auf Grund der Fokussierung auf Objekte wird diese Art, Anwendungen zu entwickeln, als objektorientiertes Paradigma oder auch als objektorientierte Programmierung bezeichnet. Dieses Paradigma ist die Basis jeglicher modernen Anwendungsentwicklung und wird von fast allen modernen Programmiersprachen unterstützt oder – bei konsequent objektorientierten Sprachen wie C# – sogar gefordert.

Kapitel 4
Typen

4.1 Werte- und Verweistypen

Damit eine Anwendung Daten verarbeiten kann, muss Speicher für diese Daten reserviert werden. Wie viel Speicher dafür benötigt wird, ist jedoch abhängig von den Typen der Daten, da nicht jeder Typ gleich viel Speicher belegt. Zudem wird in C# noch zwischen zwei Arten von Typen unterschieden, nämlich Werte- und Verweistypen.

Wertetypen sind – wie ihr Name schon sagt – Typen, die Werte direkt speichern. Das heißt, wird von der Anwendung auf einen Wertetyp zugegriffen, dann werden die Daten direkt aus der entsprechenden Stelle im Speicher gelesen.

Im Gegensatz dazu speichern Verweistypen nur die Adresse der Speicherstelle, an der die eigentlichen Daten abgelegt sind. Greift die Anwendung also auf einen Verweistyp zu, wird zunächst aus der entsprechenden Stelle im Speicher gelesen, wo sich die eigentlichen Daten befinden, woraufhin diese in einem zweiten Schritt dann von dort gelesen werden können.

Diese zunächst aufwändig erscheinende Trennung in Werte- und Verweistypen liegt in der Größe der Daten begründet, die gespeichert werden sollen. Daten, deren Umfang im Voraus bekannt ist, werden in der Regel als Wertetyp abgelegt. Da Wertetypen – vereinfacht gesagt – in einer Tabelle im Speicher verwaltet werden, findet der Zugriff auf diese sehr schnell statt.

Bei Daten, deren Umfang allerdings nicht von vornherein feststeht, oder deren Umfang sich im Lauf der Zeit ändern kann, würde diese Tabelle immer wieder fragmentiert und müsste von Zeit zu Zeit umsortiert werden. Um das zu vermeiden, werden die eigentlichen Daten getrennt von dieser Tabelle an einer freien Adresse im Speicher abgelegt, während in der Tabelle nur ein Verweis auf diese Adresse abgelegt wird.

Da ein Verweis auf eine Speicherstelle unabhängig von deren Adresse immer gleich viel Speicher benötigt, kann die Tabelle problemlos genutzt werden, um diese Verweise aufzunehmen. Dieses Verfahren löst außerdem das Problem, wie umfangreiche Daten innerhalb einer Anwendung weitergereicht werden. Statt sämtliche Da-

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

ten zu kopieren, wird lediglich der Verweis weitergegeben, was zum einen deutlich weniger Speicher verbraucht und zum anderen wesentlich schneller ausgeführt wird.

Allerdings ergibt sich aus der Eigenschaft, in erster Linie nur mit einem Verweis an Stelle der eigentlichen Daten zu arbeiten, ein wesentlicher Unterschied zwischen Werte- und Verweistypen, der beim Umgang mit diesen beachtet werden muss. Wird ein Wertetyp kopiert, um ihn an anderer Stelle in der Anwendung zu verwenden, wird tatsächlich auf einer Kopie gearbeitet. Veränderungen an dieser beeinflussen die ursprünglichen Daten nicht.

Wird statt dessen aber ein Verweistyp kopiert, so wird nur der Verweis kopiert – die eigentlichen Daten liegen nach wie vor nur ein einziges Mal im Speicher. Werden nun die Daten der vermeintlichen Kopie verändert, ändern sich dadurch auch die ursprünglichen Daten, denn beide Verweise zeigen auf die selbe Adresse im Speicher. Um versehentliche Änderungen an Daten zu vermeiden, ist es wichtig, diesen Unterschied zu verinnerlichen.

Aus dieser Unterscheidung in Werte- und Verweistypen ergibt sich die Frage, welchen Wert ein Typ enthält, wenn er zwar bereits im Speicher angelegt wurde, ihm aber noch keine Daten zugewiesen wurden. Die Antwort auf diese Frage hängt davon ab, ob es sich um einen Werte- oder einen Verweistyp handelt.

Während Wertetypen ein Standardwert zugewiesen wird, werden Verweistypen als *null* gekennzeichnet. Das bedeutet, dass sie derzeit nicht auf eine Speicheradresse verweisen. Dabei ist zu beachten, dass *null* ein eigener Wert ist und nicht der Zahl Null entspricht. Außerdem muss im späteren Verlauf beim Zugriff auf einen Verweistyp stets überprüft werden, ob überhaupt Daten vorliegen oder ob der Verweistyp *null* ist.

Schließlich gibt es noch einen Hybriden zwischen Werte- und Verweistypen, nämlich die nullbaren Wertetypen. Ihr Ursprung liegt in der Notwendigkeit, einen Wertetyp kennzeichnen zu können, dessen Wert unbekannt oder undefiniert ist.

Häufig wird dafür der Standardwert verwendet, allerdings besteht gelegentlich Bedarf, zwischen diesem und einem tatsächlich unbekannten oder undefinierten Wert zu unterscheiden, wofür bei nullbaren Wertetypen dann null verwendet werden kann. Intern werden nullbare Wertetypen allerdings als Verweistypen umgesetzt, da nur diese die Nutzung von *null* ermöglichen.

4.2 Vordefinierte Typen

Damit bei der Entwicklung von Anwendungen nicht jeder Typ vom Benutzer entwickelt werden muss, enthält C# eine Reihe vordefinierter Typen für einfache Daten, die automatisch in jeder Anwendung zur Verfügung stehen.

Für Ganzzahlen bietet C# acht verschiedene Typen, die sich in erster Linie durch ihren Wertebereich unterscheiden. Der Wertebereich berechnet sich dabei aus der Anzahl der verfügbaren Bits, wobei nochmals zwischen vorzeichenbehafteten und vorzeichenfreien Typen unterschieden wird. Als Standardwert verwenden diese Typen die Zahl Null.

Theoretisch sollte für eine Aufgabe zwar der am besten passende Typ verwendet werden, in der Praxis werden allerdings fast ausschließlich *int* und *long* eingesetzt, da 32- und 64-Bit-Prozessoren mit diesen Typen besser umgehen können. Außerdem wirkt sich auf Grund der Art, wie .NET Speicher für Typen reserviert, auch der geringere Speicherbedarf der kleineren Typen – wenn überhaupt – nur unwesentlich aus.

Typ	Minimum	Maximum	Größe	Vorzeichen
sbyte	−128	127	8 Bit	Ja
short	−32.768	32.767	16 Bit	Ja
int	−2.147.483.648	2.147.483.647	32 Bit	Ja
long	−9.223.372.036.854.775.808	9.223.372.036.854.775.807	64 Bit	Ja
byte	0	255	8 Bit	Nein
ushort	0	65.535	16 Bit	Nein
uint	0	4.294.967.295	32 Bit	Nein
ulong	0	18.446.744.073.709.551.615	64 Bit	Nein

Für Dezimalzahlen bietet C# drei verschiedene Typen, die sich nicht nur durch ihren Wertebereich, sondern auch durch die Anzahl der verfügbaren Nachkommastellen unterscheiden. Die Typen *float* und *double* entsprechen dabei dem IEEE 754-Standard, der seit 1985 einen weltweit einheitlichen Standard zur Verarbeitung von Dezimalzahlen definiert.

Der Typ *decimal* hingegen verfügt zwar über einen kleineren Wertebereich als *float* und *double*, dafür aber über eine deutlich höhere Genauigkeit, was diesen Typ insbesondere für Finanzberechnungen interessant macht.

Als Besonderheit bieten die Typen *float* und *double* die Möglichkeit, die Werte $+0$ und -0, $+\infty$, $-\infty$ und NaN zu speichern. Die Werte $+0$ und -0 sind vor allem beim Runden interessant. Neben $+\infty$ und $-\infty$ zur Darstellung positiver und negativer Unendlichkeit können *float* und *double* auch den Wert NaN – Not a Number – speichern, um ein mathematisch nicht definiertes Ergebnis abzubilden. Für *decimal* stehen diese besonderen Werte nicht zur Verfügung.

Als Standardwert verwenden diese Typen ebenso wie die ganzzahligen Typen die Zahl Null.

Typ	Minimum	Maximum	Größe	Nachkommastellen
float	$\pm 1{,}5 \times 10^{-45}$	$\pm 3{,}4 \times 10^{38}$	32 Bit	7
double	$\pm 5{,}0 \times 10^{-324}$	$\pm 1{,}7 \times 10^{308}$	64 Bit	15 bis 16
decimal	$\pm 1{,}0 \times 10^{-28}$	$\pm 7{,}9 \times 10^{28}$	128 Bit	28 bis 29

Außer diesen Typen für Ganz- und Dezimalzahlen bietet C# noch den Typ *char* zur Aufnahme eines einzelnen Zeichens, wobei Unicode voll unterstützt wird. Ein einzelnes Zeichen wird in C# dabei durch einfache Anführungszeichen eingeschlossen. Als Standardwert wird das Zeichen mit dem Unicode-Wert Null verwendet.

Typ	Größe
Char	16 Bit

Schließlich unterstützt C# noch den Typ *bool*, der zur Darstellung der logischen Werte *true* und *false* dient. Die Werte *true* und *false* werden – ebenso wie *null* – als Literale bezeichnet. Der Standardwert für diesen Typ ist *false*.

Obwohl ein Bit in der Theorie genügen würde, um *bool* abzubilden, wird in der Praxis ein Byte verwendet, da dies die kleinste Einheit ist, die im Speicher belegt werden kann.

Typ	Größe
bool	8 Bit

Alle bislang vorgestellten Typen sind Wertetypen, deren Speicherbedarf im Voraus bekannt ist. Außer diesen Typen enthält C# noch zwei Verweistypen, nämlich *string* und *object*.

Der Typ *string* dient zur Aufnahme von Text, der aus beliebig vielen Zeichen bestehen kann. Wie *char* ist auch dieser Typ uneingeschränkt Unicode-fähig. Ein Text wird in C# durch doppelte Anführungszeichen eingeschlossen.

Der Speicherbedarf liegt aus Leistungs- und Verwaltungsgründen bei mindestens 20 Byte, wächst aber linear mit der Länge des zu speichernden Textes. Der Verweis an sich belegt – je nach Speicherarchitektur – 32 oder 64 Bit.

Typ	Größe
string	Mindestens 20 Byte

Um in den Typen *char* und *string* Sonderzeichen wie beispielsweise einen Zeilenumbruch speichern zu können, können Zeichen nicht nur in ihrer kanonischen Form angegeben, sondern auch als Unicode-Zeichen oder Escape-Sequenzen maskiert werden. Ein Unicode-Zeichen wird durch einen umgekehrten Schrägstrich eingeleitet, dem ein kleines u und die vierstellige Nummer des Zeichens folgen.

```
'\u0013'
```

Die Escape-Sequenzen beginnen ebenfalls mit einem umgekehrten Schrägstrich, bestehen weiterhin aber nur aus einem einzelnen Zeichen, das die entsprechende Escape-Sequenz identifiziert.

Um die Interpretation der Escape-Sequenzen durch C# zu unterdrücken, kann einem Text außerhalb der doppelten Anführungszeichen ein @ vorangestellt werden. Insbesondere bei der Verwendung von Pfadangaben, die zahlreiche umgekehrte

Sequenz	Bedeutung
`'`	Einfaches Anführungszeichen
`"`	Doppeltes Anführungszeichen
`\`	Umgekehrter Schrägstrich
`0`	Zeichen mit dem Unicode-Wert 0
`a`	Alarmton
`b`	Rückschritt
`f`	Seitenvorschub
`n`	Neue Zeile
`r`	Wagenrücklauf
`t`	Horizontaler Tabulator
`v`	Vertikaler Tabulator

Schrägstriche enthalten, kann dies nützlich sein – diese müssten ansonsten jeweils als Escape-Sequenz angegeben werden.

Der Typ *object* schließlich spielt eine Sonderrolle, da alle anderen Typen von ihm abstammen. Daher kann er für jeden anderen Typ eingesetzt werden, das heißt, *object* kann einen Verweis auf beliebige Daten speichern. Dennoch findet der Zugriff typsicher statt, so dass nach wie vor der ursprüngliche Typ der Daten bekannt ist. Das heißt, dass beispielsweise auf einen Text nicht wie auf eine Zahl zugegriffen werden kann, auch wenn der Text als *object* abgelegt ist.

Zudem kann jeder Typ in *object* umgewandelt und von *object* wieder in den ursprünglichen Typ zurückgewandelt werden, was als Boxing beziehungsweise Unboxing bezeichnet wird.

Neben den 32 oder 64 Bit, die für den Verweis auf die Daten anfallen, und dem Speicherplatz für die Daten an sich, benötigt dieser Typ weitere 64 Bit für interne Verwaltungsinformationen.

Typ	Größe
object	64 Bit

4.3 Benutzerdefinierte Typen

Außer diesen vordefinierten Typen können Typen in C# auch vom Benutzer definiert werden. Zu diesem Zweck gibt es einige Konzepte, auf denen benutzerdefinierte Typen aufgebaut werden, wobei dafür wiederum verschiedene Werte- und Verweistypen zur Auswahl stehen.

An Wertetypen bietet C# Strukturen und Enumerationen, an Verweistypen neben den im vergangenen Kapitel erwähnten Klassen auch Schnittstellen, Arrays, Delegaten und die im Ansatz beschriebenen nullbaren Wertetypen an. Die Definition eigener Typen auf Basis dieser Konzepte wird in den nächsten Kapiteln im Detail beschrieben.

Kapitel 5
Namensräume

5.1 Was sind Namensräume?

Die Typen, die während der Entwicklung einer Anwendung entstehen, werden jeweils mit einem beschreibenden Namen versehen, welche Art von Daten dieser Typ verarbeiten soll. Allerdings kann es durch den Einsatz von Komponenten vorkommen, dass zwei verschiedene Typen unabhängig voneinander den gleichen Namen tragen, wobei der Name innerhalb der jeweiligen Komponente eindeutig ist.

Um diese Typen dennoch unterscheiden zu können, werden sie üblicherweise in sogenannten Namensräumen organisiert. Ein Namensraum stellt dabei einen Container dar, der die in ihm enthaltenen Typen von den Typen anderer Namensräume abschottet. Wird ein Typ in keinen Namensraum eingeordnet, befindet er sich automatisch im globalen Namensraum, der mit global:: bezeichnet wird.

Innerhalb eines Namensraumes reicht der Name eines Typs zu seiner Identifikation aus. Typen, die sich in anderen Namensräumen befinden, müssen jedoch zusätzlich mit ihrem zugehörigen Namensraum angesprochen werden. Namensräume können zudem hierarchisch angeordnet werden, um verschiedene Ebenen zu definieren. Der Name eines Typs einschließlich seines kompletten Namensraumbezeichners wird als vollqualifizierter Name bezeichnet.

Die Verschachtelung von Namensräumen wird in C# mit Hilfe des Punkt-Operators durchgeführt, wobei der Punkt dann die einzelnen hierarchischen Ebenen voneinander trennt.

Die FCL enthält bereits zahlreiche Namensräume, von denen der wichtigste System heißt. In ihm befinden sich alle elementaren Typen, die zur Entwicklung von Anwendungen benötigt werden. Unterhalb von System gibt es spezialisierte Namensräume wie beispielsweise System.Data oder System.Xml zum Zugriff auf Datenbanken und XML-Dokumente.

Generell gilt die Regel, dass der oberste Namensraum einer Komponente dem Namen der Firma entsprechen sollte, welche die Komponente entwickelt. Darunter wird üblicherweise ein Namensraum angeordnet, dessen Name dem der Komponen-

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

te oder der Anwendung entspricht. Weitere Namensräume, mit denen die verwendeten Typen detaillierter organisiert werden können, finden sich schließlich auf der untersten Ebene.

Beispiele für Namensräume, die diesem Schema folgen, sind beispielsweise Microsoft.IE, Microsoft.SqlServer.Server oder Microsoft.WindowsMobile.DirectX.Direct3D. Wichtig bei der Benennung von Namensräumen ist, dass sprechende Namen verwendet werden, aus denen hervorgeht, welche Arten von Daten die enthaltenen Typen abdecken. Außerdem muss das erste Zeichen eines Namens ein Buchstabe oder ein Unterstrich sein, Ziffern oder sonstige Zeichen sind nicht erlaubt.

Bei der Namensvergabe muss zudem beachtet werden, dass die Groß- und Kleinschreibung in C# generell relevant ist. Daher bezeichnen die Namen System und system verschiedene Namensräume. Für die Schreibweise zusammengesetzter Wörter wird in C# je nach Kontext entweder Pascal Case oder Camel Case verwendet. In Pascal Case wird der Anfangsbuchstabe jedes Wortes groß geschrieben, in Camel Case bildet das erste Wort hierzu eine Ausnahme, da dessen Anfangsbuchstabe klein geschrieben wird.

Bei Namensräumen wird immer Pascal Case als Schreibweise verwendet, weswegen es beispielsweise System.SqlServer und nicht system.sqlServer heißt.

Außerdem gelten in .NET für sämtliche Namen die Richtlinien, dass Abkürzungen nicht und Akronyme nur dann verwendet werden, wenn sie allgemein gebräuchlich sind. Akronyme, die aus höchstens zwei Zeichen bestehen, werden vollständig groß geschrieben, ab einer Länge von drei Zeichen gilt wieder, dass je nach Kontext Pascal Case oder Camel Case verwendet wird.

Auf Grund dieser Richtlinien heißt es System.IO an Stelle von System.Io und System.Xml an Stelle von System.XML.

Schließlich ist noch zu beachten, dass Bezeichner nicht identisch mit Schlüsselwörtern der Sprache C# sein dürfen – als Schlüsselwörter werden dabei alle Wörter bezeichnet, die in C# bereits als Anweisung oder als sonstiger Ausdruck enthalten sind. Sofern der Name eines Bezeichners zwingend einem Schlüsselwort entsprechen muss, wird dem Bezeichner ein @ vorangestellt, wodurch der Bezeichner und das Schlüsselwort dann unterschieden werden können. Diese Möglichkeit sollte allerdings nur in Ausnahmefällen in Betracht gezogen und in der Regel vermieden werden.

5.2 Vordefinierte Namensräume

Verwendet man Typen aus einem anderen Namensraum in einer eigenen Komponente, so muss jedes Mal der vollqualifizierte Name des Typs angegeben werden. Da die Lesbarkeit des Codes bei tief verschachtelten Namensräumen dadurch beeinträchtigt werden kann, ist es möglich, Namensräume einzubinden, so dass deren Typen so verwendet werden können, als befänden sie sich im aktuellen Namensraum.

Dazu dient in C# die *using*-Direktive, die in der Regel zu Beginn einer Datei angegeben wird, wobei sich ein Namensraum durchaus über mehr als eine Datei erstrecken kann. Da einige Namensräume wie unter anderem System von fast jeder Komponente benötigt werden, fügen die meisten Entwicklungsumgebungen in eine neue Datei automatisch die entsprechenden Zeilen ein.

C#

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

Die Semikola am jeweiligen Zeilenende kennzeichnen in C# das Ende einer Anweisung. Da jede Anweisung durch ein Semikolon abgeschlossen werden muss, kann sie auch auf mehrere Zeilen verteilt werden, was bei langen Zeilen eventuell die Lesbarkeit verbessern kann. Zudem werden zusätzliche Leerstellen und Leerzeilen ignoriert.

Die Namensräume alphabetisch zu sortieren und zwischen Namensräumen, deren oberste Ebene sich unterscheidet, eine Leerzeile einzufügen, erleichtert das Auffinden eines bestimmten eingebundenen Namensraumes und wird im allgemeinen als guter Stil angesehen.

C#

```
1 using Microsoft.IE;
2 using Microsoft.SqlServer.Server;
3
4 using System;
5 using System.Collections.Generic;
6 using System.Text;
```

Statt einen Namensraum einzubinden, kann alternativ ein Alias definiert werden, so dass der Namensraum zumindest über einen kürzeren Namen angesprochen werden kann.

C#

```
1 using D3D =
2     Microsoft.WindowsMobile.DirectX.Direct3D;
```

Alle Typen, die im Namensraum Microsoft.WindowsMobile.DirectX.Direct3D enthalten sind, können nun vollqualifiziert über den Alias D3D angesprochen werden. Ebenso kann ein Alias für einen Typen definiert werden, indem statt eines Namensraumes der Name eines Typs angegeben wird. Generell kann die Verwendung von Aliasen manchmal nützlich sein, allerdings wird dieses Konstrukt in der Praxis eher selten genutzt.

5.3 Benutzerdefinierte Namensräume

Außer der Einbindung von bestehenden Namensräumen können auch eigene Namensräume definiert werden, um eigene Typen zusammenzufassen und zu organisieren. Dazu dient in C# die namespace-Anweisung. Als Parameter erfordert sie den Namen eines Namensraumes, zudem folgt ihr ein Namensraumrumpf, der von geschweiften Klammern eingeschlossen wird.

Anweisungen, denen ein durch geschweifte Klammern eingeschlossener Block folgt, werden in C# nicht durch ein Semikolon abgeschlossen und stellen daher eine Ausnahme von der Regel dar.

C#

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

Um verschachtelte Namensräume zu erstellen, kann eine weitere *namespace*-Anweisung in den Rumpf eingebettet werden. Die im Rumpf eingebetteten Zeilen einzurücken, erhöht die Lesbarkeit, da Blockanfang und -ende sofort ersichtlich sind, zudem gilt dies ebenfalls als guter Stil.

C#

```
1 using System;
2
3 namespace GoloRoden
4 {
5     namespace GuideToCSharp
6     {
7     }
8 }
```

Statt dessen kann auch direkt in der äußeren Anweisung der vollqualifizierte Name des inneren Namensraumes angegeben werden.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5 }
```

Dieser Code kann nun mit Hilfe des C#-Compilers in MSIL übersetzt werden. Dabei besteht prinzipiell die Möglichkeit, eine Komponente zur Verwendung in einer Anwendung mit der Dateiendung .dll oder eine eigenständige Anwendung mit der Dateiendung .exe zu erzeugen.

Für eine eigenständig lauffähige Anwendung müssen allerdings einige Bedingungen erfüllt werden, denen der vorliegende Code nicht gerecht wird, weshalb derzeit nur die Möglichkeit besteht, eine Komponente zu erzeugen.

In .NET erfolgt das Kompilieren mit Hilfe des Compilers csc.exe, in Mono trägt der Compiler den Namen mcs.exe.

Um in .NET eine Komponente zu erzeugen, muss der Compiler mit dem /target-Parameter und dem Wert library aufgerufen werden, wobei /target optional als /t abgekürzt werden kann. Zudem muss als weiterer Parameter die zu kompilierende Datei angegeben werden.

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

Die Parameter des Compilers von Mono sind kompatibel, so dass der Aufruf fast identisch mit dem des Compilers von .NET ist.

```
mcs /target:library Component.cs
```

Das Ergebnis ist in beiden Fällen eine Assembly mit der Dateiendung .dll, die als Komponente in einer Anwendung eingesetzt werden kann. Sofern ein anderer Name für die Assembly vergeben werden soll, kann dazu so wohl in .NET wie auch in Mono der Parameter /out verwendet werden.

```
csc /target:library /out:File.dll Component.cs
```

beziehungsweise

```
mcs /target:library /out:File.dll Component.cs
```

Kapitel 6
Klassen und Strukturen

6.1 Was sind Klassen?

Da C# eine objektorientierte Sprache ist, wird am häufigsten das Konzept der Klasse zur Erstellung eigener Typen verwendet. Klassen sind, wie bereits erwähnt, Baupläne für Objekte, mit denen Daten modelliert werden können.

Eine Klasse wird in C# mit dem Schlüsselwort *class* erzeugt, dem der Name der Klasse folgt. Klassen besitzen ebenso wie Namensräume einen durch geschweifte Klammern eingeschlossenen Rumpf, weshalb ihre Definition nicht durch ein Semikolon abgeschlossen wird.

Wie bei Namensräumen, so gibt es auch bei Klassen Richtlinien, wie deren Namen gebildet werden. Ein Klassenname besteht aus einem oder mehreren Substantiven, die den Zweck der Klasse beschreiben, wobei in der Regel der Singular verwendet wird. Für die Schreibweise gilt Pascal Case.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     class ComplexNumber
6     {
7     }
8 }
```

Dieser Code erzeugt eine Klasse zur Darstellung komplexer Zahlen. Komplexe Zahlen zeichnen sich in der Mathematik dadurch aus, dass mit ihnen die Wurzel von -1 berechnet werden kann, was unter Verwendung lediglich reeller Zahlen nicht möglich ist. Die Wurzel aus -1 wird dabei mit der imaginären Einheit i bezeichnet, wobei

$i^2 = -1$

DOI: 10.1007/978-3-540-27889-4, © Springer 2008

gilt. Komplexe Zahlen werden in der Regel in der Form

```
a + b × i
```

dargestellt, wobei a als der Real- und b als der Imaginärteil bezeichnet werden. Bevor die Klasse ausgebaut wird, um komplexe Zahlen darstellen und verarbeiten zu können, sollte sie zunächst kommentiert werden.

Prinzipiell gibt es in C# drei Arten von Kommentaren. Die einfachste Variante stellen einzeilige Kommentare dar, die durch einen doppelten Schrägstrich eingeleitet werden, und an einer beliebigen Stelle einer Zeile beginnen können, wobei für einen Kommentar in der Regel eine neue Zeile verwendet wird, um die Übersichtlichkeit zu bewahren.

Einzeilige Kommentare werden im wesentlichen für interne Kommentare des Entwicklers verwendet und kennzeichnen häufig Zeilen im Code, an denen die Arbeit noch nicht abgeschlossen ist.

Außerdem werden einzeilige Kommentare oft verwendet, um die Arbeitsweise von Code zu erläutern, so dass dies auch nach Wochen oder Monaten noch nachvollzogen werden kann, ohne dass eine mühsame Analyse und Einarbeitung erforderlich wäre.

Generell ist es beim Einfügen von Kommentaren ratsam, diese mit einem Datum zu versehen. Vor allem in Teams wird dies zudem häufig durch ein Namenskürzel ergänzt, was Nachfragen erleichtert. Daher wird es im allgemeinen als guter Stil angesehen, wenn Code derart kommentiert wird.

In welcher Sprache kommentiert wird, ist prinzipiell beliebig, allerdings wird oft auf Englisch zurückgegriffen, unter anderem, um in mehrsprachigen Teams über eine einheitliche Kommunikationssprache zu verfügen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     class ComplexNumber
 6     {
 7         // TODO gr: Add code here.
 8         //          2007-04-08
 9     }
10 }
```

Bei umfangreicheren Kommentaren kann es lästig sein, jede Zeile einzeln durch einen doppelten Schrägstrich einleiten zu müssen. Daher gibt es die zweite Variante von Kommentaren, sogenannte Blockkommentare, die durch einen Schrägstrich gefolgt von einem Stern eingeleitet werden und erst dann enden, wenn sie durch einen Stern gefolgt von einem Schrägstrich wieder geschlossen werden.

Wie viele Zeilen sich innerhalb eines solchen Blockkommentars befinden, ist dabei beliebig. Das Anwendungsgebiet von Blockkommentaren ist dabei aber das gleiche wie das von einzeiligen Kommentaren. Generell gilt für Kommentare, dass

sie keine Anweisungen darstellen und daher nicht mit einem Semikolon abgeschlossen werden müssen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     class ComplexNumber
 6     {
 7         /* TODO gr: Add code here.
 8                     2007-04-08 */
 9     }
10 }
```

Beiden Typen von Kommentaren ist gemein, dass sie nur für den internen Gebrauch gedacht sind. Gerade bei Komponenten, die auch von anderen Entwicklern genutzt werden, ist jedoch eine Trennung in private und öffentliche Kommentare sinnvoll. Die privaten Kommentare werden dabei weiterhin dafür genutzt, den Code mit internen Anmerkungen zu versehen, die öffentlichen Kommentare dienen hingegen als Dokumentation.

Zur Erstellung dieser Dokumentation dient die dritte Variante, die durch drei Schrägstriche eingeleitet wird und durch XML formatiert werden kann, weshalb diese Kommentare gelegentlich auch als XML-Kommentare bezeichnet werden. In diesen Kommentaren werden im Gegensatz zu den anderen Typen weder Datum noch Namenskürzel angegeben.

Außerdem können XML-Kommentare nicht an beliebiger Stelle im Code auftreten, sondern müssen direkt vor dem zu kommentierenden Element stehen. Die Beschreibung einer Klasse wird durch die XML-Elemente <summary> und </summary> eingeschlossen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     class ComplexNumber
 9     {
10         // TODO gr: Add code here.
11         //          2007-04-08
12     }
13 }
```

Für jede Klasse muss entschieden werden, ob sie nur innerhalb der Assembly genutzt werden können soll, welche die Klasse enthält, oder ob jede Komponente

der Anwendung Zugriff erhalten soll. Die Antwort auf die Frage, ob es sinnvoll ist, die Sichtbarkeit einzuschränken, hängt vom Zweck der Klasse ab.

Handelt es sich um eine unterstützende Klasse, die nur innerhalb der Assembly benötigt wird, empfiehlt es sich, die Sichtbarkeit einzuschränken. Ist die Klasse jedoch eine tragende Datenstruktur, die der gesamten Anwendung zur Verfügung stehen soll, wird sie uneingeschränkt zur Verfügung gestellt.

Um die Sichtbarkeit einer Klasse auf die sie enthaltende Assembly zu beschränken, wird ihre Definition mit dem Zugriffsmodifizierer *internal* versehen. Der öffentliche Zugriff wird erreicht, indem statt dessen der Zugriffsmodifizierer *public* angegeben wird. Wird auf die Angabe eines solchen Zugriffsmodifizierers verzichtet, ist eine Klasse implizit *internal*.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         // TODO gr: Add code here.
11         //          2007-04-08
12     }
13 }
```

Aus Gründen der Übersichtlichkeit wird generell für jede Klasse eine eigene Datei verwendet, deren Name dem der enthaltenen Klasse entspricht. Prinzipiell ist es zwar möglich, innerhalb einer Datei mehrere Klassen zu definieren, dies ist jedoch unüblich und gilt als schlechter Stil.

Zwei Ausnahmen von dieser Regel kommen in C# vor: Partielle Klassen und verschachtelte Klassen. Partielle Klassen, die seit Version 2.0 von C# verfügbar sind, ermöglichen es durch Angabe des zusätzlichen Schlüsselwortes *partial* bei der Definition der Klasse, eine Klasse auf mehrere Dateien zu verteilen.

Dies wird beispielsweise von Visual Studio genutzt, um vom Benutzer geschriebenen Code und von Visual Studio generierten Code, der sich auf die gleiche Klasse bezieht, voneinander zu trennen, so dass der Benutzer nicht versehentlich generierten Code überschreibt oder verändert, und umgekehrt. Außer in Fällen, in denen generierter und benutzergeschriebener Code gemischt werden, sollte vom Einsatz partieller Klassen abgesehen werden.

Verschachtelte Klassen hingegen ermöglichen, innerhalb einer Klasse eine weitere Klasse zu definieren, genau so, wie auch innerhalb eines Namensraumes ein weiterer Namensraum angelegt werden kann. Im Gegensatz zu Namensräumen ist dies bei Klassen in der Praxis jedoch unüblich, zudem gibt es so gut wie keine Anwendungsfälle, in denen ein solches Verfahren notwendig wäre, weshalb darauf nicht näher eingegangen wird.

6.2 Felder

Damit ein Objekt Daten speichern kann, müssen in der zugehörigen Klasse Felder für die einzelnen Daten definiert werden. Felder sollten nur für solche Daten definiert werden, die nicht funktional abhängig von anderen Daten sind – das heißt, lassen sich Daten aus anderen vorhandenen Daten ermitteln, werden sie nicht abgespeichert.

Um eine komplexe Zahl mit der Klasse ComplexNumber abbilden zu können, werden zwei Felder benötigt, nämlich eines für den Real- und eines für den Imaginärteil. Die Frage, von welchem Typ diese Felder sind, ist einfach zu beantworten: Da so wohl Real- wie auch Imaginärteil nach Definition reelle Zahlen sind, werden beide mit Hilfe eines Typs für Dezimalzahlen dargestellt.

Die Bennenung von Feldern erfolgt ähnlich wie die von Klassen, da auch hier der Name aus einem oder mehreren Substantiven gebildet wird und als Gesamtbegriff im Singular steht. Allerdings wird für Felder Camel Case eingesetzt, zudem wird den Namen häufig ein Unterstrich vorangestellt.

Da die Definition eines Feldes in C# eine Anweisung darstellt, wird sie mit einem Semikolon abgeschlossen. Zudem werden auch Felder mit Hilfe von XML-Kommentaren dokumentiert, wobei auch hier wieder das <summary>-Tag zum Einsatz kommt.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         float _realPart;
14
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         float _imaginaryPart;
19
20         // TODO gr: Add code here.
21         //          2007-04-08
22     }
23 }
```

Ebenso wie für Klassen, so muss auch für Felder die Sichtbarkeit entschieden werden. Außer *internal* und *public*, welche die gleiche Bedeutung wie bei Klassen

haben, steht für Felder zusätzlich noch das Schlüsselwort *private* zur Verfügung. Wird ein Feld als *private* gekennzeichnet, kann nur aus der Klasse auf das Feld zugegriffen werden, die das Feld enthält.

Im Sinne eines durchgängig objektorientierten Aufbaus einer Anwendung ist es allerdings erforderlich, quasi jedes Feld als *private* zu markieren, um den direkten Zugriff von außen zu verhindern.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         private float _realPart;
14
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         private float _imaginaryPart;
19
20         // TODO gr: Add code here.
21         //          2007-04-08
22     }
23 }
```

Felder können mit Standardwerten versehen werden, indem ihnen bei der Definition der gewünschte Wert zugewiesen wird, wobei dies in der Praxis eher selten angewandt wird, weswegen im folgenden in der Regel darauf verzichtet wird.

Genau genommen wird bei Feldern zwischen Deklaration und Definition unterschieden – während das Feld bei der Deklaration nur der Klasse hinzugefügt wird, wird ihm bei der Definition zusätzlich noch ein Wert zugewiesen. In der Regel wird diese Unterscheidung allerdings nur selten genutzt und generell von Definition gesprochen.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     /// <summary>
6     /// Represents a complex number.
7     /// </summary>
8     public class ComplexNumber
```

```
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         private float _realPart = 0;
14
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         private float _imaginaryPart = 0;
19
20         // TODO gr: Add code here.
21         //          2007-04-08
22     }
23 }
```

Die einzige Ausnahme zu dieser Regel besteht in der Definition eines Feldes mit konstantem Wert, wobei dessen Typ das Schlüsselwort *const* vorangestellt wird. Der Wert eines solchen konstanten Feldes kann im weiteren Verlauf der Anwendung dann nicht mehr geändert werden. Konstanten werden beispielsweise genutzt, um mathematisch feststehende Werte wie die Zahl Pi oder die Eulersche Zahl zu definieren.

C#

```
1 private const double _pi = 3.1415926;
```

6.3 Eigenschaften

Da der Zugriff auf Felder, die als *private* gekennzeichnet wurden, nur noch innerhalb der Klasse möglich ist, stellt sich die Frage, wie Daten eines Objektes überhaupt gelesen oder geschrieben werden können, ohne die Sichtbarkeit des entsprechenden Feldes wieder auf *internal* oder *public* ändern zu müssen.

Die Lösung stellen Eigenschaften dar, die zwar nach außen sichtbar sind, aber auf die Felder einer Klasse zugreifen können. Der Unterschied zwischen dem Zugriff auf ein Feld mit Hilfe einer Eigenschaft und dem direkten Zugriff liegt darin, dass die Eigenschaft zusätzliche Prüfungen ausführen kann.

Eine Eigenschaft trägt üblicherweise den gleichen Namen wie das Feld, für das die Eigenschaft zuständig ist. Der einzige Unterschied liegt darin, dass Pascal Case an Stelle von Camel Case verwendet wird und der einleitende Unterstrich entfällt. Zudem ist eine Eigenschaft in der Regel *internal* oder *public*, da sie ansonsten von außen nicht sichtbar wäre – dennoch können Eigenschaften theoretisch auch als *private* gekennzeichnet werden.

Da eine Eigenschaft den Zugriff auf ein Feld gestattet, muss sie über den gleichen Typ verfügen. Der Zugriff an sich erfolgt über zwei Schlüsselwörter, *get* und *set*, die für das Auslesen und Schreiben der entsprechenden Daten zuständig sind.

Die einfachste Variante einer Eigenschaft besteht darin, mit *get* lediglich ein Feld zurückzugeben, ohne weitere Prüfungen auszuführen. Dies geschieht mit Hilfe der Anweisung *return*, der das zurückzugebende Feld folgt. Ebenso wird mit *set* nur der zu setzende Wert in das entsprechende Feld geschrieben. Der zu setzende Wert befindet sich dabei in einem Parameter namens *value* und kann mit Hilfe des Zuweisungsoperators geschrieben werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         private float _realPart;
14
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         private float _imaginaryPart;
19
20         public float RealPart
21         {
22             get
23             {
24                 return _realPart;
25             }
26
27             set
28             {
29                 _realPart = value;
30             }
31         }
32
33         public float ImginaryPart
34         {
35             get
36             {
37                 return _imaginaryPart;
38             }
39
40             set
41             {
42                 _imaginaryPart = value;
43             }
44         }
```

```
45
46          // TODO gr: Add code here.
47          //          2007-04-08
48      }
49  }
```

Zudem ist es mit Eigenschaften möglich, ein Feld nur für den Lese- oder nur für den Schreibzugriff freizugeben, indem nur entweder *get* oder *set* definiert wird. Außerdem kann seit C# 2.0 entweder *get* oder *set* ein stärker einschränkender Zugriffsmodifizierer zugewiesen werden, um beispielsweise den schreibenden Zugriff auf die Klassenebene zu beschränken, den lesenden Zugriff aber auf Anwendungsebene zu gestatten.

Dazu wird entweder *get* oder *set* ein entsprechender Zugriffsmodifizierer wie *internal* oder *private* vorangestellt. Hierbei muss allerdings beachtet werden, dass dies nur erlaubt ist, wenn eine Eigenschaft so wohl über *get* wie auch *set* verfügt, und selbst dann darf ein weiterer Zugriffsmodifizierer nur bei einem der beiden Schlüsselwörter angegeben werden. Das andere behält den Zugriffsmodifizierer, der für die Eigenschaft an sich definiert ist.

Zudem muss der Zugriffsmodifizierer, der *get* oder *set* vorangestellt wird, restriktiver sein als der Zugriffsmodifizierer der gesamten Eigenschaft. Wenn also beispielsweise der schreibende Zugriff auf den Realteil einer komplexen Zahl auf die Klasse beschränkt werden soll, muss dem *set* ein *private* vorangestellt werden.

C#

```
 1 public float RealPart
 2 {
 3     get
 4     {
 5         return _realPart;
 6     }
 7
 8     private set
 9     {
10         _realPart = value;
11     }
12 }
```

Auch Eigenschaften werden mit Hilfe von XML-Kommentaren dokumentiert, wobei ein Kommentar mit einem <summary>-Tag zum Einsatz kommt. Zusätzlich enthält der Kommentar für eine Eigenschaft aber noch eine Beschreibung der Daten, die von der Eigenschaft ausgelesen beziehungsweise gesetzt werden. Dies geschieht mit Hilfe des <value>-Tags.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
```

```
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         private float _realPart;
14 
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         private float _imaginaryPart;
19 
20         /// <summary>
21         /// Gets or sets the real part.
22         /// </summary>
23         /// <value>The real part.</value>
24         public float RealPart
25         {
26             get
27             {
28                 return _realPart;
29             }
30 
31             set
32             {
33                 _realPart = value;
34             }
35         }
36 
37         /// <summary>
38         /// Gets or sets the imaginary part.
39         /// </summary>
40         /// <value>The imaginary part.</value>
41         public float ImginaryPart
42         {
43             get
44             {
45                 return _imaginaryPart;
46             }
47 
48             set
49             {
50                 _imaginaryPart = value;
51             }
52         }
53 
54         // TODO gr: Add code here.
55         //          2007-04-08
56     }
57 }
```

Als funktional abhängige Eigenschaft bietet sich der Betrag einer komplexen Zahl an, der aus dem Real- und dem Imaginärteil ermittelt werden kann, indem beide quadriert und addiert werden und aus dem Ergebnis die Wurzel gezogen wird.

$$|z| = \sqrt{a^2 + b^2}$$

Da mathematische Operatoren noch nicht behandelt wurden, wird die entsprechende Eigenschaft an dieser Stelle nur als Platzhalter eingefügt, wobei als Ergebnis vorerst immer 0 zurückgegeben wird. Zusätzlich wird die Eigenschaft mit einem Kommentar versehen, der darauf hinweist, dass die Arbeit an diesem Codeabschnitt noch nicht abgeschlossen ist. Da lediglich das Auslesen des Betrages Sinn ergibt, wird für diese Eigenschaft nur *get* definiert, so dass ein schreibender Zugriff nicht möglich ist.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Contains the real part.
12         /// </summary>
13         private float _realPart;
14
15         /// <summary>
16         /// Contains the imaginary part.
17         /// </summary>
18         private float _imaginaryPart;
19
20         /// <summary>
21         /// Gets or sets the real part.
22         /// </summary>
23         /// <value>The real part.</value>
24         public float RealPart
25         {
26             get
27             {
28                 return _realPart;
29             }
30
31             set
32             {
33                 _realPart = value;
34             }
35         }
36
```

```
37          /// <summary>
38          /// Gets or sets the imaginary part.
39          /// </summary>
40          /// <value>The imaginary part.</value>
41          public float ImginaryPart
42          {
43              get
44              {
45                  return _imaginaryPart;
46              }
47
48              set
49              {
50                  _imaginaryPart = value;
51              }
52          }
53
54          /// <summary>
55          /// Gets the absolute value.
56          /// </summary>
57          /// <value>The absolute value.</value>
58          public float AbsoluteValue
59          {
60              get
61              {
62                  // TODO gr: Calculate absolute value.
63                  //          2007-04-08
64                  return 0;
65              }
66          }
67
68          // TODO gr: Add code here.
69          //          2007-04-08
70      }
71  }
```

Obwohl es möglich ist, innerhalb von *get* und *set* weitere Anweisungen unterzubringen, enthalten die meisten Eigenschaften lediglich die Minimalvariante zum Lesen und Schreiben eines Feldes. Seit der Version 3.0 von C# gibt es für solche Standardeigenschaften eine verkürzte Schreibweise, so dass an Stelle von

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents a foo class.
7      /// </summary>
8      public class Foo
9      {
10         /// <summary>
```

```
11          /// Contains a bar value.
12          /// </summary>
13          private object _bar;
14
15          /// <summary>
16          /// Gets or sets the bar value.
17          /// </summary>
18          /// <value>The bar value.</value>
19          public object Bar
20          {
21              get
22              {
23                  return this._bar;
24              }
25
26              set
27              {
28                  this._bar = value;
29              }
30          }
31      }
32  }
```

auch die kürzere Variante

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents a foo class.
 7      /// </summary>
 8      public class Foo
 9      {
10          /// <summary>
11          /// Gets or sets the bar value.
12          /// </summary>
13          /// <value>The bar value.</value>
14          public object Bar
15          {
16              get;
17              set;
18          }
19      }
20  }
```

geschrieben werden kann. Semantisch sind beide Varianten identisch, allerdings stellt sich bei der verkürzten Schreibweise die Frage, auf welches Feld mit Hilfe der Eigenschaft zugegriffen wird. Die Antwort auf diese Frage lautet, dass C# intern ein Feld anlegt, dessen Name dem Entwickler nicht bekannt ist, weshalb auf dieses Feld ausschließlich über die Eigenschaft zugegriffen werden kann.

6.4 Methoden

Während Eigenschaften zwar geeignet sind, auf Felder lesend und schreibend zuzugreifen, sind ihre Möglichkeiten, andere Aufgaben auszuführen, eher gering. Außerdem beziehen sich Eigenschaften immer nur auf jeweils ein Feld, allerdings kann es vorkommen, dass mehrere Werte verarbeitet werden müssen. Für diese Fälle, die über einen reinen Datenzugriff hinausgehen, gibt es Methoden.

Eine Methode ist ein benannter Codeabschnitt, der über seinen Namen aufgerufen und ausgeführt werden kann. Dabei können einer Methode mit Hilfe von Parametern Daten übergeben werden, außerdem kann eine Methode über einen Rückgabewert verfügen. Parameter dienen also der Eingabe von Daten, der Rückgabewert hingegen der Ausgabe von Daten, wobei beide allerdings optional sind.

Eine einfache Methode verfügt weder über Parameter noch über einen Rückgabewert und bezieht alle Daten, die zu ihrer Ausführung benötigt werden, aus der Klasse, welche die Methode enthält.

Prinzipiell werden Parameter in einer kommagetrennten Liste an die Methode übergeben, die in runden Klammern hinter dem Methodennamen angegeben wird. Werden keine Parameter verwendet, so wird nur ein leeres Paar runder Klammern an den Methodennamen angehängt. Der Rückgabewert wird hingegen vor dem Methodennamen notiert, indem der Typ des Rückgabewertes angegeben wird. Wird kein Rückgabewert verwendet, wird dies mit dem Schlüsselwort *void* gekennzeichnet. Als Methode ohne Parameter und Rückgabewert wird daher Conjugate eingeführt, welche die Konjugation einer komplexen Zahl berechnet. Die Konjugation ergibt sich, indem das Vorzeichen des Imaginärteils umgekehrt wird, so dass die Konjugation der komplexen Zahl

$$a + b \times i$$

als

$$a - b \times i$$

dargestellt wird. Da mathematische Operatoren an dieser Stelle noch nicht behandelt wurden, wird die Methode nur als Platzhalter eingefügt, wobei sie vorerst über keine Funktionalität verfügt.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
```

```
11          /// Gets or sets the real part.
12          /// </summary>
13          /// <value>The real part.</value>
14          public float RealPart
15          {
16              get;
17              set;
18          }
19
20          /// <summary>
21          /// Gets or sets the imaginary part.
22          /// </summary>
23          /// <value>The imaginary part.</value>
24          public float ImginaryPart
25          {
26              get;
27              set;
28          }
29
30          /// <summary>
31          /// Gets the absolute value.
32          /// </summary>
33          /// <value>The absolute value.</value>
34          public float AbsoluteValue
35          {
36              get
37              {
38                  // TODO gr: Calculate the absolute value
39                  //          and return it to the caller.
40                  //          2007-04-08
41              }
42          }
43
44          void Conjugate()
45          {
46              // TODO gr: Calculate the conjugation.
47              //          2007-04-09
48          }
49
50          // TODO gr: Add code here.
51          //          2007-04-08
52      }
53  }
```

Für die Namensgebung einer Methode gilt, dass der Name mit einem Verb beginnt, dem Substantive folgen können, wobei für die Schreibweise Pascal Case verwendet wird. Auch für eine Methode kann die Sichtbarkeit definiert werden, wobei die gleichen Zugriffsmodifizierer wie bei Feldern zur Verfügung stehen.

In der Regel werden Methoden, die Hilfsaufgaben übernehmen, als private gekennzeichnet. Methoden, welche die Schnittstelle einer Klasse nach außen darstellen, werden mit dem gleichen Zugriffsmodifizierer wie die Klasse gekennzeichnet, also mit *internal* oder *public*, je nachdem, ob die Methode nur in der Assembly oder

der gesamten Anwendung benötigt wird. Wird kein Zugriffsmodifizierer angegeben, ist eine Methode implizit *private*.

Methoden werden, da sie die Semantik einer Klasse definieren, ebenfalls mit XML-Kommentaren versehen, wobei wiederum das <summary>-Tag zum Einsatz kommt. Da eine Methode keine Anweisung ist, wird ihre Definition nicht mit einem Semikolon abgeschlossen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Gets or sets the real part.
12         /// </summary>
13         /// <value>The real part.</value>
14         public float RealPart
15         {
16             get;
17             set;
18         }
19
20         /// <summary>
21         /// Gets or sets the imaginary part.
22         /// </summary>
23         /// <value>The imaginary part.</value>
24         public float ImginaryPart
25         {
26             get;
27             set;
28         }
29
30         /// <summary>
31         /// Gets the absolute value.
32         /// </summary>
33         /// <value>The absolute value.</value>
34         public float AbsoluteValue
35         {
36             get
37             {
38                 // TODO gr: Calculate the absolute value
39                 //          and return it to the caller.
40                 //          2007-04-08
41             }
42         }
43
44         /// <summary>
```

```
45          /// Calculates the conjugation.
46          /// </summary>
47          public void Conjugate()
48          {
49              // TODO gr: Calculate the conjugation.
50              //          2007-04-09
51          }
52
53          // TODO gr: Add code here.
54          //          2007-04-08
55      }
56  }
```

Wenn ein Rückgabewert für eine Methode benötigt wird, so kann er dadurch definiert werden, dass sein Typ in der Definition der Methode an Stelle von *void* angegeben wird. Eine Methode, die überprüft, ob so wohl Real- wie auch Imaginärteil dem Zahlenwert Null entsprechen – und damit die gesamte komplexe Zahl der komplexen Null entspricht – gibt entweder *true* oder *false* zurück, womit sich als Typ des Rückgabewertes *bool* ergibt.

Methoden, deren Rückgabewert *bool* ist, folgen bei der Benennung einer weiteren Richtlinie: Als Verb wird in der Regel is eingesetzt, so dass sich für den Test auf Null der Name IsZero ergibt. Der Grund für diese Richtlinie ist, dass der Name einer solchen Methode als logische Aussage gelesen werden kann.

Außerdem enthalten Methoden, die über einen Rückgabewert verfügen, als letzte Anweisung ein *return*, so dass in dieser Hinsicht eine gewisse Ähnlichkeit zu *get* von Eigenschaften besteht.

Sofern eine Methode über einen Rückgabewert verfügt, wird dieser gesondert von <summary> in dem XML-Kommentar der Methode aufgeführt und durch das XML-Tag <returns> gekennzeichnet. Um innerhalb der Dokumentation Schlüsselwörter als solche hervorzuheben, können sie durch das XML-Tag <c> markiert werden.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents a complex number.
 7      /// </summary>
 8      public class ComplexNumber
 9      {
10          /// <summary>
11          /// Gets or sets the real part.
12          /// </summary>
13          /// <value>The real part.</value>
14          public float RealPart
15          {
16              get;
```

```
17              set;
18          }
19
20          /// <summary>
21          /// Gets or sets the imaginary part.
22          /// </summary>
23          /// <value>The imaginary part.</value>
24          public float ImginaryPart
25          {
26              get;
27              set;
28          }
29
30          /// <summary>
31          /// Gets the absolute value.
32          /// </summary>
33          /// <value>The absolute value.</value>
34          public float AbsoluteValue
35          {
36              get
37              {
38                  // TODO gr: Calculate the absolute value
39                  //          and return it to the caller.
40                  //          2007-04-08
41              }
42          }
43
44          /// <summary>
45          /// Calculates the conjugation.
46          /// </summary>
47          public void Conjugate()
48          {
49              // TODO gr: Calculate the conjugation.
50              //          2007-04-09
51          }
52
53          /// <summary>
54          /// Checks whether the complex number is zero.
55          /// </summary>
56          /// <returns><c>true</c> if the real and the
57          /// imaginary part are zero; <c>false</c>
58          /// otherwise.<returns>
59          public bool IsZero()
60          {
61              // TODO gr: Check whether the real and the
62              //          imaginary part are zero and return
63              //          the result to the caller.
64              //          2007-04-09
65          }
66
67          // TODO gr: Add code here.
68          //          2007-04-08
69      }
70  }
```

Schließlich können Methoden auch Parameter enthalten, mit deren Hilfe Daten an eine Methode bei ihrem Aufruf übergeben werden können. Wie bereits erwähnt, werden Parameter in einer kommaseparierten Liste innerhalb der runden Klammern definiert. Im Gegensatz zum Rückgabewert reicht es allerdings nicht aus, hierbei nur die Typen der Parameter anzugeben, da sie sonst innerhalb der Methode nicht unterscheidbar wären.

Daher erhält jeder Parameter einen Namen, wobei dafür die Richtlinien der Namensgebung von Feldern gelten, mit der Ausnahme, dass für die Schreibweise von Parametern Camel Case verwendet wird. Als Beispiel bieten sich die Addition und die Multiplikation mit einer weiteren komplexen Zahl und die Potenz mit einer reellen Zahl an. Alle drei Methoden verfügen über keinen Rückgabewert, da das Ergebnis direkt in der komplexen Zahl gespeichert wird.

Während den ersten beiden Methoden ein Objekt der Klasse ComplexNumber übergeben wird, erwartet die Potenz eine Dezimalzahl als Parameter. Jeder Parameter wird, wie bereits der Rückgabewert, durch einen entsprechenden XML-Kommentar beschrieben, der durch das <param>-Tag gekennzeichnet wird. Innerhalb des öffnenden Tags befindet sich das Attribut name, dem der Name des beschriebenen Parameters zugewiesen wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         /// <summary>
11         /// Gets or sets the real part.
12         /// </summary>
13         /// <value>The real part.</value>
14         public float RealPart
15         {
16             get;
17             set;
18         }
19
20         /// <summary>
21         /// Gets or sets the imaginary part.
22         /// </summary>
23         /// <value>The imaginary part.</value>
24         public float ImginaryPart
25         {
26             get;
27             set;
28         }
29
30         /// <summary>
```

```
31          /// Gets the absolute value.
32          /// </summary>
33          /// <value>The absolute value.</value>
34          public float AbsoluteValue
35          {
36              get
37              {
38                  // TODO gr: Calculate the absolute value
39                  //          and return it to the caller.
40                  //          2007-04-08
41              }
42          }
43
44          /// <summary>
45          /// Calculates the conjugation.
46          /// </summary>
47          public void Conjugate()
48          {
49              // TODO gr: Calculate the conjugation.
50              //          2007-04-09
51          }
52
53          /// <summary>
54          /// Checks whether the complex number is zero.
55          /// </summary>
56          /// <returns><c>true</c> if the real and the
57          /// imaginary part are zero; <c>false</c>
58          /// otherwise.<returns>
59          public bool IsZero()
60          {
61              // TODO gr: Check whether the real and the
62              //          imaginary part are zero and return
63              //          the result to the caller.
64              //          2007-04-09
65          }
66
67          /// <summary>
68          /// Adds the specified summand to the current complex
69          /// number.
70          /// </summary>
71          /// <param name="summand">The complex number that is
72          /// used as summand.</param>
73          public void Add(ComplexNumber summand)
74          {
75              // TODO gr: Add the summand to the current
76              //          complex number.
77              //          2007-04-09
78          }
79
80          /// <summary>
81          /// Multiplies the current complex number with the
82          /// specified factor.
83          /// </summary>
84          /// <param name="factor">The complex number that is
```

```
 85          /// used as factor.</param>
 86          public void Multiply(ComplexNumber factor)
 87          {
 88              // TODO gr: Multiply the factor with the current
 89              //          complex number.
 90              //          2007-04-09
 91          }
 92
 93          /// <summary>
 94          /// Raises the current complex number to the power of
 95          /// the specified real number.
 96          /// </summary>
 97          /// <param name="exponent">The real number that is
 98          /// used as exponent.</param>
 99          public void Pow(float exponent)
100          {
101              // TODO gr: Raise the current complex number to a
102              //          power.
103              //          2007-04-09
104          }
105
106          // TODO gr: Add code here.
107          //          2007-04-08
108      }
109  }
```

Obwohl die Definitionen der Methoden Add, Multiply und Pow prinzipiell gleich aussehen, unterscheiden sie sich in einem wesentlichen Aspekt: Die Parameter von Add und Multiply sind vom Typ ComplexNumber – einem Verweistyp –, während der Parameter der Methode Pow vom Typ *float* ist – einem Wertetyp. Das heißt, dass die Methoden Add und Multiply nur einen Verweis auf ihren jeweiligen Parameter erhalten, die Methode Pow dagegen eine Kopie des Wertes des Parameters.

Verändert eine der Methoden also ihren Parameter, so hat das verschiedene Auswirkungen. Die Methoden Add und Multiply würden nicht nur den Wert ändern, auf den sie zugreifen, sondern auch den Wert der Methode, aus der sie aufgerufen werden, was eventuell nicht gewünscht ist. Die Methode Pow hingegen kann ihren Wert nach Belieben ändern, da sie eine eigene Kopie erhalten und daher keinen Zugriff auf die Daten der aufrufenden Methode hat.

Diese beiden Möglichkeiten, einen Parameter als Verweis oder als echte Kopie der Daten zu übergeben, werden by reference und by value genannt. Wertetypen werden standardmäßig by value übergeben, Verweistypen by reference. Allerdings kann auch ein Wertetyp by reference übergeben werden, so dass die aufrufende und die aufgerufene Methode auf die gleichen Daten zugreifen.

Dies geschieht, indem das Schlüsselwort *ref* dem Parameter vorangestellt wird, was allerdings in der Praxis nur sehr selten benötigt wird. Andersherum kann ein Verweistyp auch by value übergeben werden, allerdings muss dazu händisch eine Kopie des Objektes angelegt werden, was unter Umständen sehr aufwändig ist.

Gelegentlich kommt es vor, dass mehr als ein Rückgabewert benötigt wird. In der Regel sollte man für diesen Fall eine eigene Datenstruktur entwickeln, welche

alle notwendigen Daten aufnehmen kann. Alternativ können Parameter aber auch als zusätzliche Rückgabewerte definiert werden, indem ihnen das Schlüsselwort *out* vorangestellt wird. Parameter, die als Ausgabeparameter gekennzeichnet werden, werden implizit by reference übergeben.

Um eine Methode aufzurufen, wird zunächst das Objekt, an dem sie aufgerufen werden soll, genannt. Darauf folgt der Operator . und der Name der Methode, gefolgt von runden Klammern. Schließlich wird dieser Aufruf mit einem Semikolon abgeschlossen. Sofern Parameter an die Methode übergeben werden sollen, werden deren Werte innerhalb der runden Klammern kommasepariert angegeben. Sofern eine Methode innerhalb des eigenen Objektes aufgerufen werden soll, entfällt die Angabe des Objektnamens.

C#

```
1 // Conjugate a complex number.
2 complexNumber.Conjugate();
3
4 // Raise it to the power of 2.
5 complexNumber.Pow(2);
6
7 // Raise to the power of 2 from within the current instance.
8 Pow(2);
```

Allen Feldern, Eigenschaften und Methoden, die bislang vorgestellt wurden, ist gemein, dass sie objektgebunden sind. Das heißt, sie beziehen sich immer auf ein Objekt, auch wenn sie innerhalb einer Klasse definiert wurden. In der Regel entspricht dies dem gewünschten Verhalten, gelegentlich sollen Felder, Eigenschaften oder Methoden aber klassengebunden sein.

Auf klassengebundene Felder, Eigenschaften und Methoden kann direkt über die Klasse zugegriffen werden, ohne ein bestimmtes Objekt ansprechen zu müssen. Zudem können diese Elemente verwendet werden, ohne dass überhaupt ein Objekt der entsprechenden Klasse erzeugt wurde. Außerdem existiert ein klassengebundenes Element nur ein einziges Mal, unabhängig davon, wie viele Objekte erzeugt wurden – alle Objekte der Klasse teilen sich die einzige Instanz der klassengebundenen Elemente.

Klassengebundene Felder können beispielsweise dazu genutzt werden, um klassenweit gültige Status- oder Konfigurationsdaten allen Objekten der Klasse zur Verfügung zu stellen, ohne dass für jedes Objekt eine eigene Verwaltung dieser Daten bestehen muss. Elemente, die klassengebunden sind, werden in C# als statisch bezeichnet.

Um ein Element als statisch zu kennzeichnen, wird hinter dessen Zugriffsmodifizierer das Schlüsselwort *static* angegeben. Wenn eine Klasse nur statische Elemente enthält, kann neben den einzelnen Elementen auch die gesamte Klasse als statisch markiert werden, indem hinter ihrem Zugriffsmodifizierer das Schlüsselwort *static* angegeben wird. Da ein Objekt einer statischen Klasse auf Grund der fehlenden eigenen Felder, Eigenschaften und Methoden sinnlos wäre, kann von einer statischen Klasse kein Objekt erzeugt werden.

Der Aufruf einer statischen Methode erfolgt genauso wie der einer objektgebundenen Methode, mit der Ausnahme, dass nicht das Objekt vorangestellt wird, an dem die Methode aufgerufen werden soll. Statt dessen wird die Klasse angegeben, welche die entsprechende Methode enthält. Sofern eine Methode innerhalb der eigenen Klasse aufgerufen werden soll, entfällt die Angabe des Klassennamens.

C#

```
1 // Call a static method on class Foo.
2 Foo.Bar();
3
4 // Call a static method with parameters.
5 Foo.Bar("Hello world.");
6
7 // Call a static method from within the current class.
8 Bar();
```

Eine besondere Rolle in diesem Zusammenhang spielt die statische Methode Main, bei der die Ausführung einer Anwendung startet, weshalb in der gesamten Anwendung nur eine einzige Methode diesen Namens existieren darf. Die Klasse, in der die Methode Main enthalten ist, spielt dabei zunächst keine Rolle, da von ihr kein Objekt erzeugt wird – was wiederum begründet, warum Main eine statische Methode sein muss.

Als Rückgabewert für Main können die Typen *void* und *int* angegeben werden, je nachdem, ob ein Rückgabewert benötigt wird. Falls nicht, wird *void* verwendet, bei Angabe von *int* kann mit Hilfe der Anweisung *return* ein Wert zurückgegeben werden, der vom Betriebssystem ausgewertet werden kann. Insbesondere bei Konsolenanwendungen ist dieser Rückgabewert ein häufig genutztes Verfahren, um Fehler in der Anwendung an das Betriebssystem zu melden.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents the application class.
7      /// </summary>
8      public static class Program
9      {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15         }
16     }
17 }
```

Die Klasse Program verfügt mit der Existenz der statischen Methode Main über alle Voraussetzungen, um in eine ausführbare Assembly mit der Dateiendung .exe übersetzt zu werden. Das Kompilieren erfolgt genauso wie bei einer Assembly, die als Komponente übersetzt wird, außer dass der Parameter an Stelle von /target:library nun /target:exe lautet.

Der Aufruf von

```
csc /target:exe Program.cs
```

unter .NET und von

```
mcs /target:exe Program.cs
```

unter Mono erzeugen also eine entsprechende Assembly, die ausgeführt werden kann. Unter anderen Betriebssystemen als Windows kann es notwendig sein, die Assembly explizit über die Runtime von Mono zu starten.

```
mono Program.exe
```

Da die Klasse Program die Klasse ComplexNumber nutzen können soll, muss sie entsprechenden Zugriff erhalten. Dies geschieht entweder, indem die Klasse ComplexNumber als eigene Komponente übersetzt und anschließend eingebunden wird, oder indem beide Klassen in der gleichen Assembly bereitgestellt werden. Das zweite ist an dieser Stelle deutlich einfacher, weshalb die Anwendung mit

```
csc /target:exe Program.cs ComplexNumber.cs
```

unter .NET und mit

```
mcs /target:exe Program.cs ComplexNumber.cs
```

unter Mono erneut übersetzt wird. Da die Klasse ComplexNumber inzwischen ein wenig länger geworden ist, bietet es sich an, den Code zu gliedern. Dazu gibt es in C# die Direktive #region, die den Beginn einer Region markiert, die durch eine weitere Direktive – #endregion – abgeschlossen wird. Regionen werden beispielsweise von Visual Studio dazu genutzt, Abschnitte zusammenfassen und zuklappen zu können.

Des weiteren kann eine Region benannt werden, indem hinter der Direktive #region eine Beschreibung angegeben wird. Außerdem können Regionen ineinander verschachtelt werden, um untergeordnete Regionen zu erstellen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         #region Properties
```

```
11          /// <summary>
12          /// Gets or sets the real part.
13          /// </summary>
14          /// <value>The real part.</value>
15          public float RealPart
16          {
17              get;
18              set;
19          }
20
21          /// <summary>
22          /// Gets or sets the imaginary part.
23          /// </summary>
24          /// <value>The imaginary part.</value>
25          public float ImginaryPart
26          {
27              get;
28              set;
29          }
30
31          /// <summary>
32          /// Gets the absolute value.
33          /// </summary>
34          /// <value>The absolute value.</value>
35          public float AbsoluteValue
36          {
37              get
38              {
39                  // TODO gr: Calculate the absolute value
40                  //          and return it to the caller.
41                  //          2007-04-08
42              }
43          }
44          #endregion
45
46          #region Methods
47          /// <summary>
48          /// Calculates the conjugation.
49          /// </summary>
50          public void Conjugate()
51          {
52              // TODO gr: Calculate the conjugation.
53              //          2007-04-09
54          }
55
56          /// <summary>
57          /// Checks whether the complex number is zero.
58          /// </summary>
59          /// <returns><c>true</c> if the real and the
60          /// imaginary part are zero; <c>false</c>
61          /// otherwise.<returns>
62          public bool IsZero()
63          {
64              // TODO gr: Check whether the real and the
```

```
65              //          imaginary part are zero and return
66              //          the result to the caller.
67              //          2007-04-09
68          }
69
70          /// <summary>
71          /// Adds the specified summand to the current complex
72          /// number.
73          /// </summary>
74          /// <param name="summand">The complex number that is
75          /// used as summand.</param>
76          public void Add(ComplexNumber summand)
77          {
78              // TODO gr: Add the summand to the current
79              //          complex number.
80              //          2007-04-09
81          }
82
83          /// <summary>
84          /// Multiplies the current complex number with the
85          /// specified factor.
86          /// </summary>
87          /// <param name="factor">The complex number that is
88          /// used as factor.</param>
89          public void Multiply(ComplexNumber factor)
90          {
91              // TODO gr: Multiply the factor with the current
92              //          complex number.
93              //          2007-04-09
94          }
95
96          /// <summary>
97          /// Raises the current complex number to the power of
98          /// the specified real number.
99          /// </summary>
100         /// <param name="exponent">The real number that is
101         /// used as exponent.</param>
102         public void Pow(float exponent)
103         {
104             // TODO gr: Raise the current complex number to
105             //          a power.
106             //          2007-04-09
107         }
108         #endregion
109
110         // TODO gr: Add code here.
111         //          2007-04-08
112     }
113 }
```

Die Methoden zur Addition und Multiplikation von komplexen Zahlen haben einen Nachteil, denn sie ermöglichen nur die Addition und Multiplikation von zwei komplexen Zahl. Um allerdings die Summe oder das Produkt aus einer komplexen

und einer reellen Zahl zu berechnen, muss die reelle Zahl erst in eine komplexe Zahl abgebildet werden, bei welcher der Realteil der reellen Zahl entspricht, der Imaginärteil hingegen Null ist.

Zur Lösung dieses Problems können die Methoden Add und Multiply mehrfach definiert werden, sofern sich die einzelnen Definitionen in ihrer Signatur unterscheiden. Als Signatur wird dabei der Name einer Methode einschließlich der Typen ihrer Parameter bezeichnet. Der Rückgabewert spielt für die Signatur allerdings keine Rolle, weshalb zwar zwei Methoden mit dem gleichen Rückgabewert und unterschiedlichen Parametern definiert werden können, allerdings nicht umgekehrt.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents a complex number.
    /// </summary>
    public class ComplexNumber
    {
        #region Properties
        /// <summary>
        /// Gets or sets the real part.
        /// </summary>
        /// <value>The real part.</value>
        public float RealPart
        {
            get;
            set;
        }

        /// <summary>
        /// Gets or sets the imaginary part.
        /// </summary>
        /// <value>The imaginary part.</value>
        public float ImginaryPart
        {
            get;
            set;
        }

        /// <summary>
        /// Gets the absolute value.
        /// </summary>
        /// <value>The absolute value.</value>
        public float AbsoluteValue
        {
            get
            {
                // TODO gr: Calculate the absolute value
                //          and return it to the caller.
                //          2007-04-08
```

```
42              }
43          }
44          #endregion
45
46          #region Methods
47          /// <summary>
48          /// Calculates the conjugation.
49          /// </summary>
50          public void Conjugate()
51          {
52              // TODO gr: Calculate the conjugation.
53              //          2007-04-09
54          }
55
56          /// <summary>
57          /// Checks whether the complex number is zero.
58          /// </summary>
59          /// <returns><c>true</c> if the real and the
60          /// imaginary part are zero; <c>false</c>
61          /// otherwise.<returns>
62          public bool IsZero()
63          {
64              // TODO gr: Check whether the real and the
65              //          imaginary part are zero.
66              //          2007-04-09
67
68              // Return the result to the caller.
69              return false;
70          }
71
72          /// <summary>
73          /// Adds the specified summand to the current complex
74          /// number.
75          /// </summary>
76          /// <param name="summand">The complex number that is
77          /// used as summand.</param>
78          public void Add(ComplexNumber summand)
79          {
80              // TODO gr: Add the summand to the current
81              //          complex number.
82              //          2007-04-09
83          }
84
85          /// <summary>
86          /// Adds the specified summand to the current complex
87          /// number.
88          /// </summary>
89          /// <param name="summand">The real number that is
90          /// used as summand.</param>
91          public void Add(float summand)
92          {
93              // TODO gr: Add the summand to the current
94              //          complex number.
95              //          2007-04-10
```

```
 96          }
 97
 98          /// <summary>
 99          /// Multiplies the current complex number with the
100          /// specified factor.
101          /// </summary>
102          /// <param name="factor">The complex number that is
103          /// used as factor.</param>
104          public void Multiply(ComplexNumber factor)
105          {
106              // TODO gr: Multiply the factor with the current
107              //          complex number.
108              //          2007-04-09
109          }
110
111          /// <summary>
112          /// Multiplies the current complex number with the
113          /// specified factor.
114          /// </summary>
115          /// <param name="factor">The real number that is
116          /// used as factor.</param>
117          public void Multiply(float factor)
118          {
119              // TODO gr: Multiply the factor with the current
120              //          complex number.
121              //          2007-04-10
122          }
123
124          /// <summary>
125          /// Raises the current complex number to the power of
126          /// the specified real number.
127          /// </summary>
128          /// <param name="exponent">The real number that is
129          /// used as exponent.</param>
130          public void Pow(float exponent)
131          {
132              // TODO gr: Raise the current complex number to a
133              //          power.
134              //          2007-04-09
135          }
136          #endregion
137
138          // TODO gr: Add code here.
139          //          2007-04-08
140      }
141  }
```

Seit der Version 3.0 von C# gibt es neben partiellen Klassen auch sogenannte partielle Methoden, die ebenfalls mit Hilfe des Schlüsselwortes *partial* definiert werden. Eine partielle Methode ermöglicht es, das Vorhandensein einer Methode in einer partiellen Klasse zu definieren, ohne die Methode an sich bereitstellen zu müssen. Die partielle Methode kann dann in einem anderen Bestandteil der Klas-

se implementiert werden, geschieht dies nicht, wird der Aufruf der entsprechenden Methode entfernt.

Das Einsatzgebiet von partiellen Methoden ähnelt dem von partiellen Klassen: Während es mit partiellen Klassen möglich ist, generierten Code von benutzerdefiniertem Code zu trennen, was beispielsweise von den Designern in Visual Studio genutzt wird, ermöglichen partielle Methoden dem Designer, eine Methode zu definieren und bereits zu verwenden, deren Inhalt allerdings vom Entwickler noch implementiert werden muss.

Partielle Methoden müssen zwingend mit dem Zugriffsmodifizierer *private* gekennzeichnet werden und können nur *void* als Rückgabetyp haben. Zudem können partielle Methoden nur innerhalb einer partiellen Klasse definiert werden, da sie sonst nicht vom Entwickler ergänzt werden könnten. Zu guter letzt können partielle Methoden so wohl klassen- wie auch instanzbezogen sein und über Parameter verfügen.

6.5 Konstruktoren

Nachdem die Klasse ComplexNumber nun sämtliche benötigten Felder, Eigenschaften und Methoden enthält, fehlt zu der Vollendung ihres Rahmens noch eine Methode, die zur Laufzeit der Anwendung ein Objekt dieser Klasse erzeugt und dieses Objekt mit geeigneten Standardwerten initialisiert. Eine solche Methode wird in der objektorientierten Programmierung als Konstruktor bezeichnet.

Prinzipiell trägt ein Konstruktor immer den Namen der Klasse und gleicht abgesehen von einer Ausnahme einer normalen Methode: Ein Konstruktor verfügt im Gegensatz zu allen anderen Methoden nicht über einen Rückgabewert, so dass dessen Angabe schlichtweg entfällt. Parameter hingegen können auch bei Konstruktoren angegeben werden, um beispielsweise Standardwerte für das zu erstellende Objekt vorzugeben.

Wird für eine Klasse kein Konstruktor definiert, verfügt sie implizit über einen parameterlosen Konstruktor, der lediglich dazu dient, ein Objekt dieser Klasse zu erzeugen. Ebenso wie normale Methoden können Konstruktoren überladen und mit einem Zugriffsmodifizierer versehen werden, wobei dieser angibt, von wo aus die Klasse instanziiert werden kann.

In der Regel wird als Zugriffsmodifizierer der der Klasse verwendet, allerdings gibt es Fälle, in denen die Instanziierung verhindert werden soll. Um ein solches Verhalten zu erreichen, kann *private* als Zugriffsmodifizierer für den Konstruktor angegeben werden, wodurch eine Instanziierung nur noch aus der Klasse selbst erfolgen kann.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
```

```
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         #region Properties
11         /// <summary>
12         /// Gets or sets the real part.
13         /// </summary>
14         /// <value>The real part.</value>
15         public float RealPart
16         {
17             get;
18             set;
19         }
20 
21         /// <summary>
22         /// Gets or sets the imaginary part.
23         /// </summary>
24         /// <value>The imaginary part.</value>
25         public float ImginaryPart
26         {
27             get;
28             set;
29         }
30 
31         /// <summary>
32         /// Gets the absolute value.
33         /// </summary>
34         /// <value>The absolute value.</value>
35         public float AbsoluteValue
36         {
37             get
38             {
39                 // TODO gr: Calculate the absolute value
40                 //          and return it to the caller.
41                 //          2007-04-08
42             }
43         }
44         #endregion
45 
46         #region Methods
47         /// <summary>
48         /// Calculates the conjugation.
49         /// </summary>
50         public void Conjugate()
51         {
52             // TODO gr: Calculate the conjugation.
53             //          2007-04-09
54         }
55 
56         /// <summary>
57         /// Checks whether the complex number is zero.
```

```
58          /// </summary>
59          /// <returns><c>true</c> if the real and the
60          /// imaginary part are zero; <c>false</c>
61          /// otherwise.<returns>
62          public bool IsZero()
63          {
64              // TODO gr: Check whether the real and the
65              //          imaginary part are zero and return
66              //          the result to the caller.
67              //          2007-04-09
68          }
69
70          /// <summary>
71          /// Adds the specified summand to the current complex
72          /// number.
73          /// </summary>
74          /// <param name="summand">The complex number that is
75          /// used as summand.</param>
76          public void Add(ComplexNumber summand)
77          {
78              // TODO gr: Add the summand to the current
79              //          complex number.
80              //          2007-04-09
81          }
82
83          /// <summary>
84          /// Adds the specified summand to the current complex
85          /// number.
86          /// </summary>
87          /// <param name="summand">The real number that is
88          /// used as summand.</param>
89          public void Add(float summand)
90          {
91              // TODO gr: Add the summand to the current
92              //          complex number.
93              //          2007-04-10
94          }
95
96          /// <summary>
97          /// Multiplies the current complex number with the
98          /// specified factor.
99          /// </summary>
100         /// <param name="factor">The complex number that is
101         /// used as factor.</param>
102         public void Multiply(ComplexNumber factor)
103         {
104             // TODO gr: Multiply the factor with the current
105             //          complex number.
106             //          2007-04-09
107         }
108
109         /// <summary>
110         /// Multiplies the current complex number with the
111         /// specified factor.
```

```
112         /// </summary>
113         /// <param name="factor">The real number that is
114         /// used as factor.</param>
115         public void Multiply(float factor)
116         {
117             // TODO gr: Multiply the factor with the current
118             //          complex number.
119             //          2007-04-10
120         }
121
122         /// <summary>
123         /// Raises the current complex number to the power of
124         /// the specified real number.
125         /// </summary>
126         /// <param name="exponent">The real number that is
127         /// used as exponent.</param>
128         public void Pow(float exponent)
129         {
130             // TODO gr: Raise the current complex number to a
131             //          power.
132             //          2007-04-09
133         }
134         #endregion
135
136         #region Constructors
137         /// <summary>
138         /// Initializes a new instance of the ComplexNumber
139         /// type using default values.
140         /// </summary>
141         public ComplexNumber()
142         {
143             // TODO gr: Set default values for the real and
144             //          imaginary part.
145             //          2007-04-25
146         }
147
148         /// <summary>
149         /// Initializes a new instance of the ComplexNumber
150         /// type using the specified real value.
151         /// </summary>
152         /// <param name="realPart">The real part.</param>
153         public ComplexNumber(float realPart)
154         {
155             // TODO gr: Set default values for the real and
156             //          imaginary part.
157             //          2007-04-25
158         }
159
160         /// <summary>
161         /// Initializes a new instance of the ComplexNumber
162         /// type using the specified real and imaginary
163         /// values.
164         /// </summary>
165         /// <param name="realPart">The real part.</param>
```

```
166         /// <param name="imaginaryPart">The imaginary
167         /// part.</param>
168         public ComplexNumber(
169             float realPart, float imaginaryPart)
170         {
171             // TODO gr: Set default values for the real and
172             //          imaginary part.
173             //          2007-04-25
174         }
175         #endregion
176     }
177 }
```

Das Setzen der Werte, die in den Parametern übergeben wurden, erfolgt prinzipiell genauso wie in Eigenschaften. Der einzige Unterschied besteht darin, dass jeder Parameter einen eigenen Namen trägt und nicht über das Schlüsselwort *value* angesprochen wird.

Dabei besteht die Möglichkeit, den zu setzenden Wert dem Feld oder der Eigenschaft zuzuweisen. In der Regel ist es gleich, welche Variante genutzt wird, allerdings sollte die gewählte Variante durchgängig verwendet werden.

Unter Umständen kann es zu Namenskonflikten kommen, wenn beispielsweise ein Feld oder eine Eigenschaft den gleichen Namen trägt wie ein Parameter. Obwohl solche Konflikte nicht nur in Konstruktoren, sondern grundsätzlich in jeder Methode auftreten können, häufen sie sich in jenen. Schließlich existiert hier potenziell für jedes Feld ein entsprechender gleichnamiger Parameter.

Um in diesem Fall den Parameter auf der einen und das Feld oder die Eigenschaft auf der anderen Seite unterscheiden zu können, enthält C# das Schlüsselwort *this*, das eine Referenz auf das eigene Objekt zur Verfügung stellt. Im Konfliktfall muss daher jedem nicht eindeutigen Bezeichner, der ein Feld oder eine Eigenschaft beschreibt, *this* vorangestellt werden.

Auch wenn die Verwendung des Schlüsselwortes *this* ansonsten optional ist, gilt es als guter Stil, es bei jedem Verweis auf ein Feld, eine Eigenschaft oder eine Methode des eigenen Objektes anzugeben.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         #region Properties
11         /// <summary>
12         /// Gets or sets the real part.
13         /// </summary>
14         /// <value>The real part.</value>
```

```
15          public float RealPart
16          {
17              get;
18              set;
19          }
20
21          /// <summary>
22          /// Gets or sets the imaginary part.
23          /// </summary>
24          /// <value>The imaginary part.</value>
25          public float ImginaryPart
26          {
27              get;
28              set;
29          }
30
31          /// <summary>
32          /// Gets the absolute value.
33          /// </summary>
34          /// <value>The absolute value.</value>
35          public float AbsoluteValue
36          {
37              get
38              {
39                  // TODO gr: Calculate the absolute value
40                  //          and return it to the caller.
41                  //          2007-04-08
42              }
43          }
44          #endregion
45
46          #region Methods
47          /// <summary>
48          /// Calculates the conjugation.
49          /// </summary>
50          public void Conjugate()
51          {
52              // TODO gr: Calculate the conjugation.
53              //          2007-04-09
54          }
55
56          /// <summary>
57          /// Checks whether the complex number is zero.
58          /// </summary>
59          /// <returns><c>true</c> if the real and the
60          /// imaginary part are zero; <c>false</c>
61          /// otherwise.<returns>
62          public bool IsZero()
63          {
64              // TODO gr: Check whether the real and the
65              //          imaginary part are zero and return
66              //          the result to the caller.
67              //          2007-04-09
68          }
```

```
69
70          /// <summary>
71          /// Adds the specified summand to the current complex
72          /// number.
73          /// </summary>
74          /// <param name="summand">The complex number that is
75          /// used as summand.</param>
76          public void Add(ComplexNumber summand)
77          {
78              // TODO gr: Add the summand to the current
79              //          complex number.
80              //          2007-04-09
81          }
82
83          /// <summary>
84          /// Adds the specified summand to the current complex
85          /// number.
86          /// </summary>
87          /// <param name="summand">The real number that is
88          /// used as summand.</param>
89          public void Add(float summand)
90          {
91              // TODO gr: Add the summand to the current
92              //          complex number.
93              //          2007-04-10
94          }
95
96          /// <summary>
97          /// Multiplies the current complex number with the
98          /// specified factor.
99          /// </summary>
100         /// <param name="factor">The complex number that is
101         /// used as factor.</param>
102         public void Multiply(ComplexNumber factor)
103         {
104             // TODO gr: Multiply the factor with the current
105             //          complex number.
106             //          2007-04-09
107         }
108
109         /// <summary>
110         /// Multiplies the current complex number with the
111         /// specified factor.
112         /// </summary>
113         /// <param name="factor">The real number that is used
114         /// as factor.</param>
115         public void Multiply(float factor)
116         {
117             // TODO gr: Multiply the factor with the current
118             //          complex number.
119             //          2007-04-10
120         }
121
122         /// <summary>
```

```
123         /// Raises the current complex number to the power of
124         /// the specified real number.
125         /// </summary>
126         /// <param name="exponent">The real number that is
127         /// used as exponent.</param>
128         public void Pow(float exponent)
129         {
130             // TODO gr: Raise the current complex number to a
131             //          power.
132             //          2007-04-09
133         }
134         #endregion
135
136         #region Constructors
137         /// <summary>
138         /// Initializes a new instance of the ComplexNumber
139         /// type using default values.
140         /// </summary>
141         public ComplexNumber()
142         {
143             // Set default values for the real and
144             // imaginary part.
145             this.RealPart = 0;
146             this.ImaginaryPart = 0;
147         }
148
149         /// <summary>
150         /// Initializes a new instance of the ComplexNumber
151         /// type using the specified real value.
152         /// </summary>
153         /// <param name="realPart">The real part.</param>
154         public ComplexNumber(float realPart)
155         {
156             // Set default values for the real and
157             // imaginary part.
158             this.RealPart = realPart;
159             this.ImaginaryPart = 0;
160         }
161
162         /// <summary>
163         /// Initializes a new instance of the ComplexNumber
164         /// type using the specified real and imaginary
165         /// values.
166         /// </summary>
167         /// <param name="realPart">The real part.</param>
168         /// <param name="imaginaryPart">The imaginary
169         /// part.</param>
170         public ComplexNumber(
171             float realPart, float imaginaryPart)
172         {
173             // Set default values for the real and
174             // imaginary part.
175             this.RealPart = realPart;
176             this.ImaginaryPart = imaginaryPart;
```

```
177         }
178         #endregion
179     }
180 }
```

Ein unschöner Aspekt überladener Konstruktoren ist, dass sie unter Umständen redundanten Code enthalten. Daher können Konstruktoren andere Konstruktoren aufrufen, so dass sämtliche gemeinsam genutzte Funktionalität nur in einem Konstruktor enthalten sein muss. Der Aufruf erfolgt, indem hinter der Parameterliste durch einen Doppelpunkt getrennt das Schlüsselwort *this* mit den entsprechenden Parametern angegeben wird.

Jeder Konstruktor kann zusätzlich eigene Anweisungen enthalten, wobei diese erst dann ausgeführt werden, wenn sämtliche anderen Konstruktoraufrufe abgeschlossen sind. Wird ein Feld so wohl direkt wie auch im Konstruktor mit einem Wert versehen, so überschreibt die Zuweisung im Konstruktor die direkte Zuweisung.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         #region Properties
11         /// <summary>
12         /// Gets or sets the real part.
13         /// </summary>
14         /// <value>The real part.</value>
15         public float RealPart
16         {
17             get;
18             set;
19         }
20
21         /// <summary>
22         /// Gets or sets the imaginary part.
23         /// </summary>
24         /// <value>The imaginary part.</value>
25         public float ImginaryPart
26         {
27             get;
28             set;
29         }
30
31         /// <summary>
32         /// Gets the absolute value.
33         /// </summary>
```

```
34          /// <value>The absolute value.</value>
35          public float AbsoluteValue
36          {
37              get
38              {
39                  // TODO gr: Calculate the absolute value
40                  //          and return it to the caller.
41                  //          2007-04-08
42              }
43          }
44          #endregion
45
46          #region Methods
47          /// <summary>
48          /// Calculates the conjugation.
49          /// </summary>
50          public void Conjugate()
51          {
52              // TODO gr: Calculate the conjugation.
53              //          2007-04-09
54          }
55
56          /// <summary>
57          /// Checks whether the complex number is zero.
58          /// </summary>
59          /// <returns><c>true</c> if the real and the
60          /// imaginary part are zero; <c>false</c>
61          /// otherwise.<returns>
62          public bool IsZero()
63          {
64              // TODO gr: Check whether the real and the
65              //          imaginary part are zero and return
66              //          the result to the caller.
67              //          2007-04-09
68          }
69
70          /// <summary>
71          /// Adds the specified summand to the current complex
72          /// number.
73          /// </summary>
74          /// <param name="summand">The complex number that is
75          /// used as summand.</param>
76          public void Add(ComplexNumber summand)
77          {
78              // TODO gr: Add the summand to the current
79              //          complex number.
80              //          2007-04-09
81          }
82
83          /// <summary>
84          /// Adds the specified summand to the current complex
85          /// number.
86          /// </summary>
87          /// <param name="summand">The real number that is
```

```
 88          /// used as summand.</param>
 89          public void Add(float summand)
 90          {
 91              // TODO gr: Add the summand to the current
 92              //          complex number.
 93              //          2007-04-10
 94          }
 95
 96          /// <summary>
 97          /// Multiplies the current complex number with the
 98          /// specified factor.
 99          /// </summary>
100          /// <param name="factor">The complex number that is
101          /// used as factor.</param>
102          public void Multiply(ComplexNumber factor)
103          {
104              // TODO gr: Multiply the factor with the current
105              //          complex number.
106              //          2007-04-09
107          }
108
109          /// <summary>
110          /// Multiplies the current complex number with the
111          /// specified factor.
112          /// </summary>
113          /// <param name="factor">The real number that is
114          /// used as factor.</param>
115          public void Multiply(float factor)
116          {
117              // TODO gr: Multiply the factor with the current
118              //          complex number.
119              //          2007-04-10
120          }
121
122          /// <summary>
123          /// Raises the current complex number to the power of
124          /// the specified real number.
125          /// </summary>
126          /// <param name="exponent">The real number that is
127          /// used as exponent.</param>
128          public void Pow(float exponent)
129          {
130              // TODO gr: Raise the current complex number to a
131              //          power.
132              //          2007-04-09
133          }
134          #endregion
135
136          #region Constructors
137          /// <summary>
138          /// Initializes a new instance of the ComplexNumber
139          /// type using default values.
140          /// </summary>
141          public ComplexNumber()
```

```
142             : this(0, 0)
143         {
144         }
145
146         /// <summary>
147         /// Initializes a new instance of the ComplexNumber
148         /// type using the specified real value.
149         /// </summary>
150         /// <param name="realPart">The real part.</param>
151         public ComplexNumber(float realPart)
152             : this(realPart, 0)
153         {
154         }
155
156         /// <summary>
157         /// Initializes a new instance of the ComplexNumber
158         /// type using the specified real and imaginary
159         /// values.
160         /// </summary>
161         /// <param name="realPart">The real part.</param>
162         /// <param name="imaginaryPart">The imaginary
163         /// part.</param>
164         public ComplexNumber(
165             float realPart, float imaginaryPart)
166         {
167             // Set default values for the real and
168             // imaginary part.
169             this.RealPart = realPart;
170             this.ImaginaryPart = imaginaryPart;
171         }
172         #endregion
173     }
174 }
```

Im Zusammenhang mit Konstruktoren verfügen Felder zudem über eine Besonderheit – um Felder als konstant zu definieren, kann an Stelle des Schlüsselwortes *const* das Schlüsselwort *readonly* verwendet werden. Wird *readonly* der Definition eines Feldes vorangestellt, kann dessen Wert wie bei *const* nur direkt und zusätzlich noch im Konstruktor gesetzt werden. Alle weiteren Zugriffe danach können aber nur noch lesend stattfinden.

Konstruktoren können jedoch nicht nur dazu genutzt werden, um Objekte zu initialisieren. Gelegentlich kann es notwendig sein, eine Klasse an sich zu initialisieren, wobei dies insbesondere bei der Verwendung statischer Felder, Eigenschaften oder Methoden vorkommt. Dazu dienen statische Konstruktoren, die auch als Klassenkonstruktoren bezeichnet werden, und sich von den übrigen Konstruktoren durch das zusätzliche Schlüsselwort *static* unterscheiden.

Außerdem können statische Konstruktoren weder überladen noch parametrisiert werden, zudem verfügen sie nicht über einen Zugriffsmodifizierer. Ausgeführt werden statische Konstruktoren beim ersten Zugriff auf die Klasse – unabhängig von der Art des Zugriffs.

6.6 Strukturen

Neben Klassen verfügt C# über ein weiteres Konzept zur Definition von Typen, das Klassen sehr ähnlich ist, nämlich Strukturen. Der wesentliche Unterschied zwischen beiden ist, dass Klassen Verweistypen sind, Strukturen hingegen Wertetypen.

Dementsprechend bietet sich der Einsatz von Strukturen in der Regel dann an, wenn die enthaltenen Felder und Eigenschaften ausschließlich oder zumindest nahezu nur auf Wertetypen basieren.

Häufig werden Strukturen eingesetzt, wenn Daten per COM mit nicht verwalteten Anwendungen ausgetauscht werden, wobei deren Aufbau dann von der über COM angesprochenen Anwendung vorgegeben ist. In diesem Zusammenhang kann mit dem *sizeof*-Operator der Speicherbedarf einer Struktur in Bytes ermittelt werden, worauf allerdings an dieser Stelle nicht weiter eingegangen wird.

Für Strukturen kann kein parameterloser Konstruktor definiert werden, dieser existiert hingegen implizit immer und initialisiert alle Felder mit den Standardwerten der entsprechenden Typen. Sofern allerdings ein eigener Konstruktor definiert wird, muss dieser zum einen über mindestens einen Parameter verfügen, zum anderen müssen in ihm alle Felder der Struktur mit einem Wert initialisiert werden.

Die Definition einer Struktur erfolgt prinzipiell zu der einer Klasse, außer dass das entsprechende Schlüsselwort *struct* statt *class* lautet. In der Entwicklung rein objektorientierter Anwendungen sind Strukturen allerdings verhältnismäßig selten geworden, da in der Regel statt dessen eine Klasse definiert wird, die deutlich mehr Flexibilität bietet.

Kapitel 7
Vererbung

7.1 Was ist Vererbung?

Da die Entwicklung einer Klasse von Grund auf sehr aufwändig sein kann, kann statt dessen eine bestehende Klasse wiederverwendet und erweitert werden. Dieses Verfahren, das als Vererbung bezeichnet wird, erzeugt aus einer bestehenden Klasse – der sogenannten Basisklasse – eine neue Klasse – die sogenannte abgeleitete Klasse –, die über alle Felder, Eigenschaften und Methoden der Basisklasse verfügt und diese um eigene Elemente erweitern kann.

Vererbung wird in C# mit Hilfe des Operators : ausgedrückt, wobei dieser sowie der Name der Basisklasse dem Namen der abgeleiteten Klasse nachgestellt werden. Es wurde bereits der Typ *object* erwähnt, von dem jeder Typ ableitet. Dies kann nun präzisiert werden: Wird für eine Klasse nicht explizit eine Basisklasse angegeben, leitet sie implizit von *object* ab. Das heißt, dass alle Eigenschaften und Methoden, die für *object* definiert sind, auch in dieser Klasse zur Verfügung stehen.

Potenziell kann eine Klasse auch explizit von *object* abgeleitet werden, indem *object* als Basisklasse angegeben wird. Da dies auf Grund der impliziten Ableitung von *object* aber keinen Unterschied macht, wird diese Ableitung in der Regel nicht angegeben.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class that explicitly derives from
 7     /// object.
 8     /// </summary>
 9     public class Foo : object
10     {
11     }
12 }
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

ist also äquivalent zu

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class that implicitly derives from
 7     /// object.
 8     /// </summary>
 9     public class Foo
10     {
11     }
12 }
```

Ein Beispiel für eine Methode, die implizit in allen Typen enthalten ist, ist die Methode ToString. Sie dient dazu, einen *string* zurückzugeben, der eine für Menschen lesbare Repräsentation des Objekts darstellt. Die in dem Typ *object* definierte Methode kann zwar an einem Objekt einer beliebigen Klasse aufgerufen werden, allerdings kennt sie die spezifischen Details der Klasse nicht. Daher gibt diese Methode standardmäßig den vollqualifizierten Typ des Objekts zurück, an dem sie aufgerufen wird.

Wird ein Objekt vom Typ ComplexNumber, der im vorangegangenen Kapitel entwickelt wurde, instanziiert, gibt die Methode ToString beispielsweise

```
GoloRoden.GuideToCSharp.ComplexNumber
```

zurück. Um eine spezifische Version der Methode ToString für den Typ ComplexNumber zu erzeugen, muss diese Methode der Klasse ComplexNumber hinzugefügt werden. Im Gegensatz zu einer klassischen Methodendefinition muss dieser Definition zwischen dem Zugriffsmodifizierer und dem Typ des Rückgabewertes das Schlüsselwort *override* hinzugefügt werden, um sicherzustellen, dass das Überschreiben der Methode der Basisklasse nicht aus Versehen, sondern absichtlich geschieht.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public class ComplexNumber
 9     {
10         #region Properties
11         #endregion
12
```

```
13          #region Methods
14          // ...
15
16          /// <summary>
17          /// Gets a string representation of the current
18          /// instance.
19          /// </summary>
20          /// <returns>A string representation of the current
21          /// instance.</returns>
22          public override string ToString()
23          {
24              // TODO gr: Create the string representation and
25              //          return it to the caller.
26              //          2007-06-11
27          }
28          #endregion
29
30          #region Constructors
31          #endregion
32      }
33  }
```

Wird das Schlüsselwort *override* weggelassen, meldet der Compiler beim Übersetzen der Anwendung eine entsprechende Warnung und fordert den Entwickler auf, das fehlende Schlüsselwort zu ergänzen.

Eine wesentliche Eigenschaft in der objektorientierten Programmierung ist in diesem Zusammenhang die Polymorphie, also die Fähigkeit eines Objekts, je nach Kontext verschiedenen Typen zu entsprechen. Jeder Typ kann durch einen übergeordneten und damit allgemeineren Typ repräsentiert werden, da dieser eine Generalisierung darstellt.

In der Praxis heißt das, dass jeder Methode, die beispielsweise einen Parmeter vom Typ *object* erwartet, ein Objekt eines beliebigen Typs übergeben werden kann – da jeder Typ implizit von *object* abgeleitet ist und *object* damit eine Generalisierung dieses Typs darstellt. Umgekehrt funktioniert dies allerdings nicht: Wird ein Parameter eines bestimmten Typs erwartet, können nur Objekte dieses oder eines abgeleiteten Typs übergeben werden.

Dieses System der Generalisierung und Spezialisierung ist ein Kernkonzept der objektorientierten Programmierung und stellt auch den Grund dar, warum jeder Typ mit Hilfe von Boxing in *object* umgewandelt werden kann – intern wird hier auf Polymorphie zurückgegriffen.

Im Allgemeinen gilt für die Beziehung zwischen einem Typ und seiner Basisklasse eine „is a“-Beziehung: Jedes Objekt vom Typ ComplexNumber ist gleichzeitig auch vom Typ *object*, während ein abgeleiteter Typ von ComplexNumber sogar zugleich vom Typ ComplexNumber und vom Typ *object* ist.

Allerdings erfordert diese Beziehung bei der Modellierung der Klassenhierarchie mehr Aufmerksamkeit, als sie zunächst vermuten lässt. Der Grund hierfür liegt in einer wesentlichen Forderung der objektorientierten Programmierung, die von Barbara Liskov formuliert wurde und daher als Liskov-Prinzip bezeichnet wird. Die

Forderung besagt, dass das Verhalten einer abgeleiteten Klasse und das ihrer Basisklasse identisch sein müssen.

Dies bedeutet, dass entgegen dem umgangssprachlichen Gebrauch ein Quadrat kein Rechteck ist, weshalb eine Klasse zur Modellierung von Quadraten nicht von einer Klasse zur Modellierung von Rechtecken abgeleitet werden darf. Während die Höhe und die Breite eines Rechtecks unabhängig voneinander verändert werden können, ist dies bei einem Quadrat nicht möglich.

Angenommen, ein Typ Quadrat wäre abgeleitet von einem Typ Rechteck, dann könnte auf Grund der Polymorphie und der Generalisierung in jeder Methode, die ein Objekt vom Typ Rechteck als Parameter erwartet, auch ein Objekt vom Typ Quadrat übergeben werden. Diese Methode könnte eine Seite dieses Objektes verdoppeln, wodurch sich bei einem Objekt des Typs Rechteck der Flächeninhalt ebenfalls verdoppelt.

Wird statt dessen ein Objekt vom Typ Quadrat übergeben, gilt dies nicht – hier würde sich der Flächeninhalt vervierfachen, da die beiden Seiten nicht unabhängig voneinander verändert werden können. Weil dabei das Liskovsche Prinzip verletzt wird, ist diese Ableitung fehlerhaft.

Außer der bislang genannten Vererbung, bei der eine Klasse von genau einer Basisklasse ableitet, gibt es prinzipiell auch die Mehrfachverebung, bei der eine Klasse über mehrere Basisklassen verfügen kann. Dieses Konzept wird in C# allerdings nicht unterstützt, da Mehrfachvererbung unter Umständen keine eindeutigen Ableitungen erzeugt, und der Nutzen in keinem Verhältnis zu dem nötigen Aufwand und der hohen Komplexität steht.

Strukturen können im Gegensatz zu Klassen nicht vererbt werden.

7.2 Felder und Eigenschaften

Die einfachsten Elemente eines Typs, die vererbt werden können, sind Felder. Bisher wurden Felder in der Regel als *private* gekennzeichnet, um den direkten Zugriff von außerhalb der Klasse zu verhindern. Allerdings kann auf solche Felder auch aus einer Unterklasse nicht zugegriffen werden. Um dies in einem gegebenen Fall zu ermöglichen, gibt es verschiedene Alternativen.

Die einfachste Variante besteht darin, das Feld als *internal* oder gar als *public* zu kennzeichnen. Allerdings geht dabei der Zugriffsschutz von außerhalb der Klasse verloren, was der Objektorientierung in den meisten Fällen widerspricht. Eine andere Möglichkeit besteht darin, über die entsprechende Eigenschaft indirekt auf das Feld zuzugreifen, was eine im Hinblick auf die Objektorientierung deutlich sauberere Variante darstellt.

In der Praxis verfügt aber nicht jedes Feld über eine zugehörige Eigenschaft, da in der Regel nur solche Felder mit einer Eigenschaft ausgestattet werden, die für die Konfiguration eines Objektes von außen wichtig sind. Felder, die hingegen nur für interne Berechnungen oder sonstige interne Belange genutzt werden und

außerhalb eines Objektes nicht zugreifbar sein sollen, bleiben üblicherweise ohne entsprechende Eigenschaft.

Abhilfe schafft in einem solchen Fall das Schlüsselwort *protected*, das den Zugriff nicht nur aus der Klasse, welche die Felddefinition enthält, ermöglicht, sondern auch aus jeder Unterklasse dieser Klasse. Felder, die als *protected* gekennzeichnet sind, stehen also von der Ebene des Zugriffs zwischen *public* und *private*.

Außerdem gibt es noch die Erweiterung des Schlüsselwortes *protected* auf *protected internal*, wodurch der Zugriff ebenfalls aus abgeleiteten Klassen ermöglicht wird, allerdings nur, sofern diese sich innerhalb der gleichen Assembly befinden.

7.3 Methoden

Werden Methoden in einem abgeleiteten Typ überschrieben, muss in dem abgeleiteten Typ die Methode explizit als *override* gekennzeichnet werden, um anzuzeigen, dass das Überschreiben beabsichtigt und kein Versehen ist. Allerdings kann nicht jede beliebige Methode einer Basisklasse überschrieben werden – dort muss eine Methode zunächst als überschreibbar gekennzeichnet werden. Dies geschieht mit Hilfe des Schlüsselwortes *virtual*.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a base class.
 7     /// </summary>
 8     public class BaseClass
 9     {
10         /// <summary>
11         /// Represents a virtual foo method.
12         /// </summary>
13         public virtual void Foo();
14     }
15 }
```

Methoden, die nicht als *virtual* gekennzeichnet werden, können von abgeleiteten Typen nicht überschrieben werden. Damit eine Methode mit dem Schlüsselwort *virtual* markiert werden kann, darf sie nicht mit dem Zugriffsmodifizierer *private* markiert sein – da sie in diesem Fall in dem abgeleiteten Typ nicht sichtbar ist. Zudem kann *virtual* nicht gleichzeitig mit *override* angegeben werden.

Wird während der Ausführung einer Anwendung eine virtuelle Methode aufgerufen, ermittelt die Common Language Runtime den tatsächlichen Typ des Objektes, an dem die Methode aufgerufen wird und ruft die zugehörige Methode auf – falls

eine entsprechende überschriebene Variante verfügbar ist. Auf diese Art wird gewährleistet, dass für ein Objekt immer die korrekte Version einer Methode aufgerufen wird.

Außer *override* gibt es noch das Schlüsselwort *new*. Der Unterschied liegt in der Bindung der Methode an den Typ – bei *override* wird die Methode in jedem Fall für den zugehörigen Typ aufgerufen, da die Methode der Basisklasse überschrieben wurde, bei *new* wird die Methode unter Umständen für den Basistyp aufgerufen, da diese Methode nicht überschrieben, sondern nur ausgeblendet wurde.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents a base class.
    /// </summary>
    public class BaseClass
    {
        /// <summary>
        /// Represents a virtual foo method.
        /// </summary>
        public virtual void Foo()
        {
        }
    }

    /// <summary>
    /// Represents a class that derives from BaseClass.
    /// </summary>
    public class DerivedClassA : BaseClass
    {
        /// <summary>
        /// Represents a foo method that overwrites the base
        /// class's implementation.
        /// </summary>
        public override void Foo()
        {
        }
    }

    /// <summary>
    /// Represents another class that derives from
    /// BaseClass.
    /// </summary>
    public class DerivedClassB : BaseClass
    {
        /// <summary>
        /// Represents a foo method that shadows the base
        /// class's implementation.
        /// </summary>
        public new void Foo()
```

```
43          {
44          }
45      }
46  }
```

Beispielhaft lässt sich das an der Klasse ComplexNumber verdeutlichen. Die Methode ToString ist dort als *override* gekennzeichnet. Das heißt, wird die Methode ToString an einem Objekt dieser Klasse aufgerufen, dann wird der Code ausgeführt, der in der überschriebenen Methode definiert wurde. Dieser Code wird auch dann ausgeführt, wenn das Objekt beispielsweise als *object* geboxt wird.

Wäre die Methode ToString statt dessen als *new* gekennzeichnet, würde ebenfalls der in der Klasse ComplexNumber definierte Code ausgeführt – aber nur, wenn diese Methode an dem ungeboxten Objekt aufgerufen wird. Erfolgte der Aufruf statt dessen an einer geboxten Version des Objektes, so würde der Code des geboxten Typs ausgeführt.

Würde das Objekt also als *object* geboxt, würde bei einem Aufruf der Methode ToString die Version ausgeführt, die in der Klasse *object* definiert wurde. In der Praxis wird *new* allerdings eher selten verwendet, in der Regel kommt das Schlüsselwort *override* zum Einsatz.

In einigen Fällen soll aus einer überschriebenen Methode explizit die Methode der Basisklasse aufgerufen werden, zum Beispiel, um deren Funktionalität auch in der überschreibenden Methode nutzen zu können. Dazu dient das Schlüsselwort *base*, das analog zu *this* verwendet werden kann, allerdings statt auf das eigene Objekt immer auf den Typ der Basisklasse verweist.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a base class.
 7     /// </summary>
 8     public class BaseClass
 9     {
10         /// <summary>
11         /// Represents a virtual foo method.
12         /// </summary>
13         public virtual void Foo()
14         {
15         }
16     }
17
18     /// <summary>
19     /// Represents a class that derives from BaseClass.
20     /// </summary>
21     public class DerivedClass : BaseClass
22     {
23         /// <summary>
```

```
24          /// Represents a foo method that overwrites the base
25          /// class's implementation.
26          /// </summary>
27          public override void Foo()
28          {
29              // Call the base method.
30              base.Foo();
31          }
32      }
33  }
```

Prinzipiell kann eine Methode, die mit *override* oder *new* gekennzeichnet wurde, in einer weiteren abgeleiteten Klasse wiederum überschrieben werden. Das Schlüsselwort *virtual* bezieht sich also nicht nur auf die direkt nachfolgende Ableitung, sondern auf alle Klassen, die in der Ableitungshierarchie nachfolgen. Um dies zu verhindern und eine weitere Vererbung zu verhindern, kann eine Methode, die mit *override* oder *new* gekennzeichnet wurde, mit Hilfe des Schlüsselwortes *sealed* versiegelt werden, wodurch keine weitere Überschreibung dieser Methode mehr möglich ist.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents a base class.
 7      /// </summary>
 8      public class BaseClass
 9      {
10          /// <summary>
11          /// Represents a virtual foo method.
12          /// </summary>
13          public virtual void Foo()
14          {
15          }
16      }
17
18      /// <summary>
19      /// Represents a class that derives from BaseClass.
20      /// </summary>
21      public class DerivedClass : BaseClass
22      {
23          /// <summary>
24          /// Represents a foo method that overwrites the base
25          /// class's implementation and avoids any further
26          /// overwriting by sealing this method.
27          /// </summary>
28          public override sealed void Foo()
29          {
30          }
```

```
31      }
32  }
```

Außerdem können vollständige Klassen versiegelt werden, was bedeutet, dass eine solche Klasse nicht vererbt werden kann. Dies ist bei Klassen sinnvoll, die eine feststehende Funktionalität bereitstellen, wie beispielsweise Klassen mit mathematischen Methoden – eine Methode zur Berechnung der Sinusfunktion zu überschreiben, ergibt wenig Sinn, schließlich ist der Sinus bereits das endgültige Resultat.

Daher ist beispielsweise die von .NET bereitgestellte Klasse Math im Namensraum System versiegelt, ebenso kann die Klasse ComplexNumber versiegelt werden.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents a complex number.
 7      /// </summary>
 8      public sealed class ComplexNumber
 9      {
10          #region Properties
11          #endregion
12
13          #region Methods
14          #endregion
15
16          #region Constructors
17          #endregion
18      }
19  }
```

Unabhängig davon, ob die Klasse ComplexNumber versiegelt ist oder nicht, handelt es sich um eine konkrete Klasse. Das bedeutet, dass sie instanziiert werden kann, dass also Objekte von ihr erzeugt werden können.

Manchmal kann es sinnvoll sein, statt dessen eine sogenannte abstrakte Klasse zu erzeugen, die nicht instanziiert werden kann, die nur als Basisklasse für andere Klassen genutzt wird, um beispielsweise gemeinsam genutzte Funktionalität zentral zur Verfügung zu stellen. Eine solche Klasse wird mit dem Schlüsselwort *abstract* gekennzeichnet und kann nicht versiegelt werden.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents an abstract base class.
 7      /// </summary>
```

```
 8     public abstract class AbstractBaseClass
 9     {
10         /// <summary>
11         /// Represents a virtual foo method.
12         /// </summary>
13         public virtual void Foo()
14         {
15         }
16     }
17 }
```

In einer abstrakten Klasse können zudem abstrakte Methoden definiert werden, die keinen Methodenrumpf enthalten, sondern nur aus dem Methodenkopf bestehen. Solche Methoden müssen mit dem Schlüsselwort *abstract* versehen werden und sind implizit *virtual*. Statt eines Methodenrumpfes, der in geschweiften Klammern angegeben wird, wird deren Methodenkopf mit einem Semikolon abgeschlossen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents an abstract base class.
 7     /// </summary>
 8     public abstract class AbstractBaseClass
 9     {
10         /// <summary>
11         /// Represents an abstract foo method.
12         /// </summary>
13         public abstract void Foo();
14     }
15
16     /// <summary>
17     /// Represents a class that derived from
18     /// AbstractBaseClass.
19     /// </summary>
20     public class DerivedClass : AbstactBaseClass
21     {
22         /// <summary>
23         /// Represents a method that implements the base
24         /// class's abstract method.
25         /// </summary>
26         public override void Foo()
27         {
28             // TODO gr: Implement abstract method.
29             //          2008-01-03
30         }
31     }
32 }
```

In einer abgeleiteten Klasse müssen abstrakte Methoden in jedem Fall implementiert werden, es sei denn, die abgeleitete Klasse wird ihrerseits wiederum als *abstract* gekennzeichnet.

Gelegentlich kann es notwendig sein, eine bestehende Klasse ohne Erzeugung einer abgeleiteten Klasse zu erweitern, ohne allerdings Zugriff auf ihren Quelltext zu haben. Beispielsweise würde eine Erweiterung des Typs *string* diesem Vorhaben entsprechen.

Zu diesem Zweck gibt es seit der Version 3.0 von C# sogenannte Erweiterungsmethoden, mit denen vorhandene Typen ergänzt werden können. Da diese Möglichkeit äußerst mächtig ist und schnell zu unübersichtlichem Code führt, wird ihr Einsatz in der Praxis als schlechter Stil angesehen. Dass Erweiterungsmethoden in C# 3.0 überhaupt in Erscheinung treten, gründet sich in der Abfragetechnik Linq, die mit C# 3.0 eingeführt wurde und auf Erweiterungsmethoden basiert.

Um einen bestehenden Typ zu erweitern, wird innerhalb einer statischen Klasse eine statische Methode definiert, welche die entsprechende Funktionalität bereitstellt. Als erster Parameter wird dieser Methode der zu erweiternde Typ übergeben, allerdings ergänzt um das Schlüsselwort *this*, woran C# erkennen kann, dass es sich nicht um eine normale, sondern um eine Erweiterungsmethode handelt.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Contains extension methods.
    /// </summary>
    public static class ExtensionMethods
    {
        /// <summary>
        /// Converts the specified string to its XML
        /// representation.
        /// </summary>
        /// <param name="source">The string that shall be
        /// converted to XML.</param>
        /// <returns>The XML representation of the specified
        /// string.</returns>
        public static string ToXml(this string source)
        {
            // TODO gr: Transform the source string to XML
            //          and return the result to the caller.
            //          2007-12-26
        }
    }
}
```

Die auf diese Art definierte Erweiterungsmethode für den Typ *string* kann nun an jeder Zeichenkette aufgerufen werden, als ob sie eine vordefinierte Methode wäre.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a foo string.
16             string foo = "Hello world!";
17
18             // Get the XML representation of the string.
19             string xml = foo.ToXml();
20         }
21     }
22 }
```

Intern prüft C# beim Aufruf einer Methode zunächst, ob eine entsprechende Methode an dem jeweiligen Typ definiert ist. Wenn nicht, wird überprüft, ob es eine statische Methode innerhalb einer statischen Klasse gibt, deren Name dem der aufgerufenen Methode entspricht, und deren erster Parameter dem gewünschten Typ entspricht, der außerdem mit dem Schlüsselwort *this* gekennzeichnet wurde. Falls eine solche Methode existiert, wird diese ausgeführt, andernfalls wird ein Fehler gemeldet.

7.4 Konstruktoren

Die einzigen Elemente eines Typs, die nicht an einen abgeleiteten Typ vererbt werden, sind Konstruktoren. Der Grund dafür liegt in einer Definition der objektorientierten Programmierung, in der die Aufgabe von Konstruktoren beschrieben wird. Diese liegt darin, ein vollständig initialisiertes Objekt zurückzugeben.

Da ein abgeleiteter Typ in der Regel weitere Felder einführt, die der Konstruktor des Basistyps nicht berücksichtigt, würde dieser der Anforderung nicht mehr gerecht, ein vollständig initialisiertes Objekt zurückzugeben. Eine abgeleitete Klasse verfügt daher zunächst nur über einen parameterlosen, leeren Standardkonstruktor.

Allerdings können entsprechende Konstruktoren definiert werden. Analog zu Methoden ist auch in den Konstruktoren der Zugriff auf die Konstrutoren des Basistyps möglich, wiederum mit Hilfe des Schlüsselwortes *base*, das mit der gleichen Syntax wie das Schlüsselwort *this* bei Konstruktoren angegeben werden kann. Wird

es angegeben, wird zunächst der Konstruktor des Basistyps aufgerufen, bevor der Konstruktor des zu instanziierenden Typs ausgeführt wird. Allerdings kann nur entweder *base* oder *this* angegeben werden.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents a base class.
    /// </summary>
    public class BaseClass
    {
        /// <summary>
        /// Initializes an instance of the BaseClass type.
        /// </summary>
        public BaseClass()
        {
        }
    }

    /// <summary>
    /// Represents a class that derives from BaseClass.
    /// </summary>
    public class DerivedClass : BaseClass
    {
        /// <summary>
        /// Initializes an instance of the DerivedClass
        /// type.
        /// </summary>
        public DerivedClass()
            : base()
        {
        }
    }
}
```

Kapitel 8
Schnittstellen

8.1 Was sind Schnittstellen?

Abstrakte Basisklassen werden häufig eingesetzt, um semantisch verwandte abgeleitete Klassen mit einer gemeinsamen Basisklasse auszustatten und gemeinsam genutzte Methoden in einem einzigen Typ zur Verfügung zu stellen. Allerdings birgt der Einsatz abstrakter Klassen einen entscheidenden Nachteil: Gelegentlich ist es notwendig, eine Klasse von einer Basisklasse der Framework Class Library abzuleiten.

Da eine Klasse aber nur über eine Basisklasse verfügen kann, können solche abgeleiteten Klassen nicht mehr unter einer benutzerdefinierten abstrakten Basisklasse angeordnet werden. In Sprachen, die Mehrfachvererbung unterstützen, können einer Klasse in einem solchen Fall einfach mehrere Basisklassen zugeordnet werden, in C# ist dies jedoch nicht möglich.

Die Lösung liegt in sogenannten Schnittstellen, die abstrakten Klassen sehr ähnlich sind, da sie ebenfalls Methodendefinitionen enthalten, aber im Gegensatz zu Klassen mehrfach vererbt werden können. Die einzige Einschränkung einer Schnittstelle ist, dass sie keine Implementierung enthalten können, sondern auf die Methodendefinitionen beschränkt sind. Insofern entspricht eine Schnittstelle einer vollständig abstrakten Klasse.

In der modernen, komponentenorientierten Entwicklung von Anwendungen spielen Schnittstellen noch eine weitere, zusätzliche Rolle. Da sie mit den in ihnen enthaltenen Methodendefinitionen nicht nur eine syntaktische Vorgabe leisten, sondern auch eine gewisse Semantik vorgeben, werden sie als eine Art Vertrag für Komponenten eingesetzt – sofern zwei verschiedene Komponenten die gleiche Schnittstelle implementieren, können sie als semantisch äquivalent eingestuft werden und sind damit untereinander austauschbar.

Wenn dieser Aspekt von Schnittstellen besonders hervorgehoben werden soll, wird an Stelle von Schnittstelle häufig auch von Kontrakt gesprochen. In der Regel werden bei der Entwicklung von Komponenten zunächst die Kontrakte definiert, bevor Komponenten entwickelt werden, die deren abstrakte Semantik kon-

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

kret umsetzen. Daher spricht man auch von Contract First Design oder Design by Contract.

Contract First Design bietet noch einen weiteren Vorteil. Da die Semantik vollständig über den Kontrakt definiert ist, ist es möglich, den Zugriff auf eine Komponente ausschließlich über deren Schnittstelle zu gestalten. Wenn die Komponente eines Tages gegen eine andere, aber semantisch äquivalente Komponente ausgetauscht werden soll, muss an der Anwendung an sich nichts geändert werden, da die Schnittstelle gleich geblieben ist.

8.2 Benutzerdefinierte Schnittstellen

Schnittstellen werden in C# mit Hilfe des Schlüsselwortes *interface* definiert, wobei ihr sonstiger Aufbau dem einer abstrakten Klasse ähnelt. Das bedeutet, dass in einer Schnittstelle wie in einer vollständig abstrakten Klasse nur Methodendefinitionen enthalten sein können, im Gegensatz zu diesen allerdings keine Zugriffsmodifizierer angegeben werden können. Alle Methoden sind implizit *public*, um den Charakter eines Kontraktes zu erfüllen.

Als Namensrichtlinie für Schnittstellen gibt es zwei Varianten. Für beide Varianten gilt, dass der Name in Pascal Case genannt wird, wobei ihm zusätzlich ein großes I vorangestellt wird. Der Name besteht entweder aus einem Adjektiv, das eine Eigenschaft beschreibt, die mit Hilfe der Schnittstelle umgesetzt wird, oder aus einem Substantiv, sofern die Schnittstelle an Stelle einer Klasse verwendet wird.

Im Namensraum System gibt es zahlreiche Beispiele für beide Varianten: Die Schnittstelle ICloneable wird von allen Klassen implementiert, deren Objekte klonbar sind – die Schnittstelle beschreibt also eine Eigenschaft, weshalb für ihren Namen ein Adjektiv gewählt wurde. Hingegen wird die Schnittstelle IServiceProvider von solchen Klassen implementiert, die Mechanismen zum Abrufen von Services bereitstellen. In diesem Fall ersetzt IServiceProvider eine entsrechende Basisklasse, weshalb für den Namen ein Substantiv gewählt wurde.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     /// <summary>
6     /// Provides methods for persisting an object.
7     /// </summary>
8     public interface IPersistable
9     {
10    }
11 }
```

Dieser Code erzeugt eine Schnittstelle IPersistable, die dazu dient, das Entwurfsmuster Memento zu implementieren. Memento ermöglicht es beliebigen Objekten,

ihren Zustand zu speichern und diesen zu einem späteren Zeitpunkt wieder abzurufen. Dazu werden die beiden Methoden Store und Restore definiert, welche die Aufgabe des Speicherns und des Wiederherstellens übernehmen.

Das Speichern der Daten übernimmt dabei ein spezielles Objekt, das sogenannte Memento. Häufig wird dieses Entwurfsmuster eingesetzt, wenn die Absicht besteht, ein Objekt zu ändern, vor der Änderung allerdings eine Kopie angefertigt werden soll, um im Falle des Falles einen Rollback ausführen und damit auf den gespeicherten Stand zurückgreifen zu können.

Da alle Methoden einer Schnittstelle implizit *public* sind, kann die Angabe eines Zugriffsmodifizierers entfallen. Da die Methoden einer Schnittstelle zudem implizit abstrakt sind, werden ihre Definitionen jeweils mit einem Semikolon abgeschlossen, wie es in einer vollständig abstrakten Klasse ebenfalls der Fall wäre.

Der Typ des Mementos, welches die zu speichernden Daten aufnimmt und den beiden Methoden als Parameter übergeben wird, wird ebenfalls als Schnittstelle angegeben – auf diese Art kann die konkrete Klasse, welche die Funktionalität des Mementos bereitstellt, problemlos ausgetauscht werden. Die einzige Voraussetzung dafür ist, dass sie die Schnittstelle IMemento implementiert.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for persisting an object.
 7     /// </summary>
 8     public interface IPersistable
 9     {
10         /// <summary>
11         /// Stores the current instance to the specified
12         /// memento.
13         /// </summary>
14         /// <param name="memento">The memento.</param>
15         void Store(IMemento memento);
16
17         /// <summary>
18         /// Restores the current instance to the specified
19         /// memento.
20         /// </summary>
21         /// <param name="memento">The memento.</param>
22         void Restore(IMemento memento);
23     }
24 }
```

Damit der Code kompiliert werden kann, muss zusätzlich noch die Schnittstelle IMemento definiert werden, die Methoden zum Speichern und Wiederherstellen von Daten enthält. Da das Memento zunächst nur in Verbindung mit der Klasse ComplexNumber eingesetzt werden soll, sind Methoden zum Speichern und Wiederherstellen von Daten des Typs *float* ausreichend.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     public interface IMemento
 9     {
10         /// <summary>
11         /// Stores the specified value using the specified
12         /// key.
13         /// </summary>
14         /// <param name="key">The key.</param>
15         /// <param name="value">The value.</param>
16         void Store(string key, float value);
17
18         /// <summary>
19         /// Restores the value stored with the specified
20         /// key.
21         /// </summary>
22         /// <param name="key">The key.</param>
23         /// <returns>The value.</returns>
24         float Restore(string key);
25     }
26 }
```

Das Prizip der Vererbung ist auch bei Schnittstellen möglich: Schnittstellen können als Basisschnittstelle für abgeleitete Schnittstellen dienen. Dies geschieht wie bei Klassen, indem bei der Definition der Schnittstelle die Basisschnittstelle durch den Operator : angehängt wird. Es kann also eine spezialisierte Version von IMemento für die Klasse ComplexNumber namens IMementoComplexNumber erzeugt werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     public interface IMemento
 9     {
10         /// <summary>
11         /// Stores the specified value using the specified
12         /// key.
13         /// </summary>
14         /// <param name="key">The key.</param>
15         /// <param name="value">The value.</param>
```

```
16          void Store(string key, float value);
17
18          /// <summary>
19          /// Restores the value stored with the specified
20          /// key.
21          /// </summary>
22          /// <param name="key">The key.</param>
23          /// <returns>The value.</returns>
24          float Restore(string key);
25      }
26
27      /// <summary>
28      /// Provides methods for a memento for the
29      /// ComplexNumber class.
30      /// </summary>
31      public interface IMementoComplexNumber : IMemento
32      {
33          /// <summary>
34          /// Stores the real value of a complex number.
35          /// </summary>
36          /// <param name="value">The real value.</param>
37          void StoreRealValue(float value);
38
39          /// <summary>
40          /// Restores the real value of a complex number.
41          /// </summary>
42          /// <returns>The real value.</returns>
43          float RestoreRealValue();
44
45          /// <summary>
46          /// Stores the imaginary value of a complex number.
47          /// </summary>
48          /// <param name="value">The imaginary value.</param>
49          void StoreImaginaryValue(float value);
50
51          /// <summary>
52          /// Restores the imaginary value of a complex number.
53          /// </summary>
54          /// <returns>The imaginary value.</returns>
55          float RestoreImaginaryValue();
56      }
57  }
```

Neben Methoden können in Schnittstellen auch Eigenschaften mit Definitionen für *get* und *set* vorgegeben werden. Felder und Konstruktoren sind hingegen ausgeschlossen, diese können nur in einer abstrakten oder konkreten Klasse definiert werden.

Schließlich stellt sich die Frage, wann eine Schnittstelle und wann eine abstrakte Basisklasse eingesetzt werden sollte. Prinzipiell bieten Schnittstellen den Vorteil, dass sie mehr Flexibilität bereitstellen, da eine Klasse zum einen von mehreren Schnittstellen ableiten kann – aber nur von einer Basisklasse –, und zum anderen eine Trennung zwischen Kontrakt und eigentlicher Implementierung besteht.

Des weiteren lässt der Einsatz von Schnittstellen die Möglichkeit bestehen, nach wie vor von einer Klasse ableiten zu können, was unter Umständen nötig ist, wenn eine Klasse beispielsweise eine bestimmte Klasse der Framework Class Library abgeleitet werden soll.

Eine abstrakte Basisklasse verfügt jedoch über einen wesentlichen Vorteil: Im Gegensatz zu Schnittstellen kann sie nicht nur Methodendefinitionen, sondern auch Code enthalten. Falls also von zahlreichen Klassen gemeinsam genutzter Code besteht, kann eine abstrakte Basisklasse helfen, die Redundanz zu vermindern und die Wartbarkeit zu verbessern.

8.3 Schnittstellen implementieren

Nachdem die Schnittstellen IPersistable, IMemento und IMementoComplexNumber definiert wurden, können diese nun von der Klasse ComplexNumber verwendet werden. Werden Schnittstellen von einer Klasse implementiert, werden diese genauso wie Basisklassen mit dem Operator : angegeben, wobei mehrere Schnittstellen kommasepariert aufgezählt werden. Wird so wohl eine Basisklasse wie auch mindestens eine Schnittstelle angegeben, muss die Basisklasse vor den Schnittstellen genannt werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public sealed class ComplexNumber : IPersistable
 9     {
10         #region Properties
11         #endregion
12
13         #region Methods
14         #endregion
15
16         #region Constructors
17         #endregion
18     }
19 }
```

Da die Klasse ComplexNumber nun die Schnittstelle IPersistable implementiert, muss sie die beiden Methoden Store und Restore der Schnittstelle bereitstellen und mit Inhalt füllen. Dazu werden die beiden Methoden implementiert, als handele es sich um native Methoden der Klasse ComplexNumber.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents a complex number.
    /// </summary>
    public sealed class ComplexNumber : IPersistable
    {
        #region Properties
        #endregion

        #region Methods
        // ...

        /// <summary>
        /// Stores the current instance in the specified
        /// memento.
        /// </summary>
        /// <param name="memento">The memento.</param>
        public void Store(IMemento memento)
        {
            // TODO gr: Store the current instance.
            //          2007-06-25
        }

        /// <summary>
        /// Restores the current instance from the specified
        /// memento.
        /// </summary>
        /// <param name="memento">The memento.</param>
        public void Restore(IMemento memento)
        {
            // TODO gr: Restore the current instance.
            //          2007-06-25
        }
        #endregion

        #region Constructors
        #endregion
    }
}
```

Diese Variante der Implementierung wird implizit genannt, da implizit gegeben ist, aus welcher Schnittstelle die Definition der entsprechenden Methode stammt. Werden von einer Klasse mehrere Schnittstellen implementiert, kann es allerdings zu Mehrdeutigkeiten kommen, wenn zwei Schnittstellen beispielsweise eine gleichnamige Methode definieren.

Für diesen Fall gibt es die explizite Implementierung, bei der dem Methodennamen der Name der Schnittstelle samt dem Operator . vorangestellt wird. Wird eine Methode explizit implementiert, darf kein Zugriffsmodifizierer angegeben werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a complex number.
 7     /// </summary>
 8     public sealed class ComplexNumber : IPersistable
 9     {
10         #region Properties
11         #endregion
12
13         #region Methods
14         // ...
15
16         /// <summary>
17         /// Stores the current instance in the specified
18         /// memento.
19         /// </summary>
20         /// <param name="memento">The memento.</param>
21         void IPersistable.Store(IMemento memento)
22         {
23             // TODO gr: Store the current instance.
24             //          2007-06-25
25         }
26
27         /// <summary>
28         /// Restores the current instance from the specified
29         /// memento.
30         /// </summary>
31         /// <param name="memento">The memento.</param>
32         void IPersistable.Restore(IMemento memento)
33         {
34             // TODO gr: Restore the current instance.
35             //          2007-06-25
36         }
37         #endregion
38
39         #region Constructors
40         #endregion
41     }
42 }
```

Kapitel 9
Delegaten

9.1 Was sind Delegaten?

Delegaten sind Verweistypen, die im Gegensatz zu den übrigen Verweistypen nicht auf Datenstrukturen, sondern auf Methoden verweisen. Delegaten ermöglichen es unter anderem, einer aufzurufenden Methode eine weitere Methode als Parameter zu übergeben. Diese übergebene Methode kann im weiteren Verlauf von der ursprünglich aufgerufenen Methode ausgeführt werden, ohne dass bekannt sein muss, in welcher Klasse diese Methode enthalten ist.

Häufig wird dies verwendet, um bei aufwändigen Berechnungen einem überwachenden Objekt zu signalisieren, dass die Berechnung abgeschlossen wurde. Dafür wird eine Methode des überwachenden Objektes als Delegat an die berechnende Klasse übergeben. Sobald die Berechnung beendet ist, wird die Methode als sogenannte Rückrufmethode an dem überwachenden Objekt aufgerufen, ohne dass die überwachende Klasse der berechnenden Klasse überhaupt bekannt sein muss.

Sobald einem Delegaten eine Methode zugewiesen wurde, verhält er sich genau wie diese Methode. Da die Bindung einer Methode an einen Delegaten allerdings nicht feststehend ist, kann dies dynamisch zur Laufzeit geändert werden, so dass sich das Verhalten der Anwendung ändern lässt. Die einzige Voraussetzung zur Bindung einer Methode an einen Delegaten ist, dass beide im Hinblick auf den Typ des Rückgabewertes und der Parameter übereinstimmen.

Ein Delegat wird ähnlich einer abstrakten Methode definiert, allerdings wird zwischen dem Zugriffsmodifizierer und dem Rückgabewert zusätzlich das Schlüsselwort *delegate* angegeben. Für die Namensgebung gilt als Richtlinie, dass der Name eines Delegaten um das Suffix Callback ergänzt wird, für die Schreibweise gilt Pascal Case. Diese Syntax wird zwar von Microsoft empfohlen, in der Framework Class Library allerdings nicht konsistent eingehalten, weshalb es einige Delegaten gibt, deren Namen dieser Konvention nicht folgen.

Im folgenden sollen ergänzend zu der Schnittstelle IPersistable Delegaten eingesetzt werden, um den Beginn und das Abschließen so wohl des Speicherns wie auch

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

des Wiederherstellens zu signalisieren. Daher werden zunächst die entsprechenden Delegaten definiert:

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Executes when storing begins.
 7     /// </summary>
 8     public delegate void StoringCallback();
 9
10     /// <summary>
11     /// Executes when storing has finished.
12     /// </summary>
13     public delegate void StoredCallback();
14
15     /// <summary>
16     /// Executes when restoring begins.
17     /// </summary>
18     public delegate void RestoringCallback();
19
20     /// <summary>
21     /// Executes when restoring has finished.
22     /// </summary>
23     public delegate void RestoredCallback();
24 }
```

Obwohl Delegaten so wohl außerhalb wie auch innerhalb einer Klasse definiert werden können, ist es üblich, sie außerhalb einer Klasse zu definieren, da sie ansonsten nur innerhalb der sie umgebenden Klasse verwendbar sind.

9.2 Multicast-Delegaten

Nachdem ein Delegat definiert wurde, kann er ebenso wie eine Klasse instanziiert werden. Während der Delegat als Typ an Hand seiner Signatur nur beschreibt, auf welche Methoden mit ihm verwiesen werden kann, verweist eine Instanz hingegen auf eine konkrete Methode. Prinzipiell entspricht diese Unterscheidung zwischen Delegat und Delegatinstanz der Unterscheidung zwischen Klasse und Objekt.

Eine Delegatinstanz wird ebenso wie ein Feld erzeugt, indem innerhalb einer Klasse ein entsprechendes Element definiert wird. Um ihr eine Methode zuzuweisen, gibt es zwei verschiedene Varianten. Zum einen kann direkt die Methode angegeben werden, zum anderen wird die Methode einem Delegatenkonstruktor übergeben. Da eine Delegatinstanz zunächst auf genau eine Methode verweist, wird sie häufig auch als Unicast-Delegat bezeichnet.

Im folgenden Code werden vier Delegatinstanzen in der Klasse ComplexNumber definiert, die auf klasseninterne Methoden verweisen. Da den Delegaten statt dessen auch Methoden anderer Objekte oder Klassen zugeordnet werden könnten, kann beliebiger Code auf die Ereignisse des Speicherns und des Wiederherstellens reagieren, ohne dass der Code in der Klasse ComplexNumber dafür speziell angepasst werden müsste. Delegaten sind dabei nicht auf objektgebundene Methoden beschränkt, sondern können ebenfalls Verweise auf klassengebundene Methoden aufnehmen.

Der Aufruf eines Delegaten gleicht dem Aufruf einer Methode. Zudem gelten für einen objektbezogenen Delegaten die gleichen Richtlinien wie für objektbezogene Methoden, für einen klassenbezogenen Delegaten gelten die gleichen Richtlinien wie für klassenbezogene Methoden.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Executes when storing begins.
    /// </summary>
    public delegate void StoringCallback();

    /// <summary>
    /// Executes when storing has finished.
    /// </summary>
    public delegate void StoredCallback();

    /// <summary>
    /// Executes when restoring begins.
    /// </summary>
    public delegate void RestoringCallback();

    /// <summary>
    /// Executes when restoring has finished.
    /// </summary>
    public delegate void RestoredCallback();

    /// <summary>
    /// Represents a complex number.
    /// </summary>
    public sealed class ComplexNumber : IPersistable
    {
        #region Properties
        #endregion

        #region Delegates
        /// <summary>
        /// Executes when storing begins.
        /// </summary>
```

```
37          private StoringCallback StoringCallback =
38              this.Storing;
39
40          /// <summary>
41          /// Executes when storing has finished.
42          /// </summary>
43          private StoredCallback StoredCallback =
44              this.Stored;
45
46          /// <summary>
47          /// Executes when restoring begins.
48          /// </summary>
49          private RestoringCallback RestoringCallback =
50              this.Restoring;
51
52          /// <summary>
53          /// Executes when restoring has finished.
54          /// </summary>
55          private RestoredCallback RestoredCallback =
56              this.Restored;
57          #endregion
58
59          #region Methods
60          // ...
61
62          /// <summary>
63          /// Stores the current instance in the specified
64          /// memento.
65          /// </summary>
66          /// <param name="memento">The memento.</param>
67          public void Store(IMemento memento)
68          {
69              // Call the storing callback.
70              this.StoringCallback();
71
72              // TODO gr: Store the current instance.
73              //          2007-06-25
74
75              // Call the stored callback.
76              this.StoredCallback();
77          }
78
79          /// <summary>
80          /// Restores the current instance from the specified
81          /// memento.
82          /// </summary>
83          /// <param name="memento">The memento.</param>
84          public void Restore(IMemento memento)
85          {
86              // Call the restoring callback.
87              this.RestoringCallback();
88
89              // TODO gr: Restore the current instance.
90              //          2007-06-25
```

```
91
92              // Call the restored callback.
93              this.RestoredCallback();
94          }
95
96          /// <summary>
97          /// Executes when storing begins.
98          /// </summary>
99          public void Storing()
100         {
101             // TODO gr: Insert code here.
102             //          2007-06-26
103         }
104
105         /// <summary>
106         /// Executes when storing has finished.
107         /// </summary>
108         public void Stored()
109         {
110             // TODO gr: Insert code here.
111             //          2007-06-26
112         }
113
114         /// <summary>
115         /// Executes when restoring begins.
116         /// </summary>
117         public void Restoring()
118         {
119             // TODO gr: Insert code here.
120             //          2007-06-26
121         }
122
123         /// <summary>
124         /// Executes when restoring has finished.
125         /// </summary>
126         public void Restored()
127         {
128             // TODO gr: Insert code here.
129             //          2007-06-26
130         }
131         #endregion
132
133         #region Constructors
134         #endregion
135     }
136 }
```

Allerdings können einer Delegatinstanz problemlos weitere Methoden zugeordnet werden. Wird ein solcher Delegat aufgerufen, werden nacheinander alle ihm zugeordneten Methoden aufgerufen. Die Aufrufreihenfolge der einzelnen Methoden ist dabei allerdings unbekannt ist, weshalb Abhängigkeiten zwischen den Methoden vermieden werden sollten. Solche Delegatinstanzen werden, da sie auf mehrere Me-

thoden verweisen, als Multicast-Delegaten bezeichnet. Hingegen werden Delegaten, die auf lediglich eine Methode verweisen, als Singlecast-Delegaten bezeichnet.

Um einer Delegatinstanz eine weitere, zusätzliche Methode zuzuordnen, wird der Operator += verwendet. Dabei kann die gleiche Methode einem Delegaten auch mehrfach zugeordnet werden, so dass sie mehrfach ausgeführt wird, sobald der Delegat aufgerufen wird. Sofern als Rückgabewert eines Delegaten nicht *void* definiert wird, wird der Rückgabewert der intern zuletzt aufgerufenen Methode zurückgegeben. Alle anderen Rückgabewerte gehen verloren.

C#

```
1 // Assign a method to the delegate.
2 MyDelegate Foo = this.Bar1;
3
4 // Assign an additional method to the delegate.
5 Foo += this.Bar2;
```

Analog zu += kann die Bindung von Methoden an einen Delegaten mit dem Operator -= wieder aufgelöst werden, wobei keine Prüfung stattfindet, ob die zu entfernende Methode tatsächlich an den Delegaten gebunden ist. Wurde eine Methode mehrfach an einen Delegaten gebunden, so muss jede Bindung einzeln aufgehoben werden. Alternativ kann einem Delegaten explizit der Wert *null* zugewiesen werden, wodurch alle Bindungen an jegliche Methoden aufgehoben werden.

C#

```
1 // Assign a method to the delegate.
2 MyDelegate Foo = this.Bar1;
3
4 // Assign an additional method to the delegate.
5 Foo += this.Bar2;
6
7 // Remove the first method from the delegate.
8 Foo -= this.Bar1;
9
10 // Assign null to the delegate and remove all methods from
11 // the delegate.
12 Foo = null;
```

9.3 Anonyme Methoden

Unter Umständen kann es aufwändig sein, eine Methode für einen Delegaten zur Verfügung zu stellen. Dies ist insbesondere dann der Fall, wenn die Methode zum einen nur an den Delegaten gebunden und ansonsten nirgends verwendet wird, und wenn sie zum anderen nur sehr wenig Code enthält.

Seit der Version 2.0 von C# gibt es daher die Möglichkeit, Code direkt an einen Delegaten zu binden, ohne dafür eine eigenständige Methode definieren zu müssen.

Ein solches Konstrukt wird – da der auszuführende Code sich wie eine Methode verhält, allerdings namenlos ist – als anonyme Methode bezeichnet, wohingegen tatsächliche Methoden als benannte Methoden bezeichnet werden.

Um einem Delegaten eine anonyme Methode zuzuweisen, wird wiederum das Schlüsselwort delegate verwendet. Der Methodenrumpf wird wie bei der Definition einer Methode durch geschweifte Klammern umschlossen, wobei die schließende geschweifte Klammer bei einer anonymen Methode durch ein zusätzliches Semikolon abgeschlossen werden muss.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
  /// <summary>
  /// Executes when storing begins.
  /// </summary>
  public delegate void StoringCallback();

  /// <summary>
  /// Executes when storing has finished.
  /// </summary>
  public delegate void StoredCallback();

  /// <summary>
  /// Executes when restoring begins.
  /// </summary>
  public delegate void RestoringCallback();

  /// <summary>
  /// Executes when restoring has finished.
  /// </summary>
  public delegate void RestoredCallback();

  /// <summary>
  /// Represents a complex number.
  /// </summary>
  public sealed class ComplexNumber : IPersistable
  {
    #region Properties
    #endregion

    #region Delegates
    /// <summary>
    /// Executes when storing begins.
    /// </summary>
    private StoringCallback StoringCallback = delegate()
    {
        // TODO gr: Insert code here.
        //          2007-06-27
    };

```

```
43      /// <summary>
44      /// Executes when storing has finished.
45      /// </summary>
46      private StoredCallback StoredCallback = delegate()
47      {
48          // TODO gr: Insert code here.
49          //          2007-06-27
50      };
51
52      /// <summary>
53      /// Executes when restoring begins.
54      /// </summary>
55      private RestoringCallback RestoringCallback = delegate()
56      {
57          // TODO gr: Insert code here.
58          //          2007-06-27
59      };
60
61      /// <summary>
62      /// Executes when restoring has finished.
63      /// </summary>
64      private RestoredCallback RestoredCallback = delegate()
65      {
66          // TODO gr: Insert code here.
67          //          2007-06-27
68      };
69      #endregion
70
71      #region Methods
72      // ...
73      /// <summary>
74      /// Stores the current instance in the specified memento.
75      /// </summary>
76      /// <param name="memento">The memento.</param>
77      public void Store(IMemento memento)
78      {
79          // Call the storing callback.
80          this.StoringCallback();
81
82          // TODO gr: Store the current instance.
83          //          2007-06-25
84
85          // Call the stored callback.
86          this.StoredCallback();
87      }
88
89      /// <summary>
90      /// Restores the current instance from the specified
91      /// memento.
92      /// </summary>
93      /// <param name="memento">The memento.</param>
94      public void Restore(IMemento memento)
95      {
96          // Call the restoring callback.
```

```
 97          this.RestoringCallback();
 98
 99          // TODO gr: Restore the current instance.
100          //          2007-06-25
101
102          // Call the restored callback.
103          this.RestoredCallback();
104      }
105      #endregion
106
107      #region Constructors
108      #endregion
109    }
110 }
```

Sofern ein Delegat über Parameter verfügt, können diese innerhalb der runden Klammern wie bei der Definition einer Methode angegeben werden. Insgesamt sollten anonyme Methoden allerdings sehr sparsam und gezielt eingesetzt werden, da sie dazu verführen, sämtliche Delegaten vor Ort zu behandeln, statt eine Anwendung sauber zu strukturieren.

9.4 Lambdaausdrücke

Seit der Version 3.0 von C# gibt es mit Hilfe der sogenannten Lambdaausdrücke eine noch weiter verkürzte Möglichkeit, anonyme Methoden zu definieren. Ein Lambdaausdruck kann überall dort verwendet werden, wo auch eine anonyme Methode möglich wäre. An Stelle des Schlüsselwortes *delegate* wird ein Lambdaausdruck innerhalb runder Klammern angegeben, die den eigentlichen Ausdruck enthalten.

Ein Lambdaausdruck bildet dabei einen Eingangsparameter auf einen Ausgangsparameter ab, wobei der Operator => verwendet wird. Um beispielsweise eine komplexe Zahl auf ihren Absolutbetrag abzubilden, kann der Lambdaausdruck

C#

```
1 (c => c.AbsoluteValue)
```

verwendet werden. Der Typ des Ein- und Ausgangsparameters ergibt sich dabei dynamisch, ebenso spielt die Wahl des Bezeichners zur Identifikation der komplexen Zahl keine Rolle, er dient nur dazu, die komplexe Zahl überhaupt ansprechen zu können.

Kapitel 10
Ereignisse

10.1 Was sind Ereignisse?

Delegaten sind nützlich, um von Objekten benachrichtigt zu werden, wenn bestimmte Ereignisse eintreffen. Allerdings gibt es ein Problem, wenn sich ein beobachtendes Objekt an einen Delegaten anhängen will – entweder müssen die Delegaten für den Zugriff von außen freigegeben werden, oder es müssen entsprechende Methoden zum Hinzufügen und Entfernen einer Methode existieren.

Beide Varianten verfügen jeweils über einige Nachteile. Während bei der Freigabe für den Zugriff von außen die Kontrolle verloren geht, so dass beispielsweise sämtliche gebundenen Methoden von außen entfernt werden könnten, erzeugt die Bereitstellung entsprechender Methoden zusätzlichen Entwicklungs- und Wartungsaufwand.

Um den Zugriff sauber kapseln zu können und den Entwicklungsaufwand möglichst gering zu halten, verfügt C# über das Konzept der Ereignisse. Prinzipiell ist ein Ereignis nichts anderes als eine für interne Delegaten öffentlich verfügbare Schnittstelle, über die beliebige Methoden an den zugehörigen Delegaten gebunden werden können. Insofern fußen Ereignisse in C# auf der Basis der Delegaten.

Damit ein Ereignis definiert werden kann, muss zunächst ein entsprechender Delegat bestehen, der als Vorlage für die Rückrufmethoden des Ereignisses fungiert. Delegaten, die für Ereignisse eingesetzt werden, folgen einer anderen Namenskonvention als die übrigen Delegaten: Ihr Name besteht aus dem Namen des Ereignisses in Pascal Case, ergänzt um das Suffix EventHandler.

Die Ereignisse an sich werden mit Hilfe des Schlüsselwortes *event* in der Klasse ComplexNumber definiert, wobei der zu verwendende Delegat in der Definition angegeben wird. Für Ereignisse gilt die Namenskonvention, dass ihr Name einem Verb entspricht – in der Verlaufsform, falls das Ereignis ausgelöst wird, bevor die eigentliche Aktion ausgeführt wird, in der Vergangenheitsform, falls danach. Für die Schreibweise gilt Pascal Case.

In der Regel sollen Methoden, die durch ein Ereignis aufgerufen werden, einige Informationen über das das Ereignis auslösende Objekt zur Verfügung gestellt wer-

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

den. Daher wird ein Delegat, der als ereignisbehandelnde Methode fungiert, selten parameterlos definiert. Es gilt als guter Stil, zwei Parameter mitzugeben, von denen der erste eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, zur Verfügung stellt, der zweite hingegen zusätzliche Informationen zu dem Ereignis an sich enthält.

Für den ersten Parameter wird zumeist der Typ *object* verwendet, wobei der Parameter mit dem Namen sender versehen wird. Der Typ des zweiten Parameters entspricht häufig einer eigens zu diesem Zweck definierten Klasse, die lediglich Felder und zugehörige Eigenschaften enthält, um Daten auszutauschen, wobei diese Klasse üblicherweise von der Klasse EventArgs aus dem Namensraum System abgeleitet wird.

Der Name der Klasse folgt den für Klassen üblichen Namenskonventionen, wobei als Suffix zusätzlich noch EventArgs angehängt wird. Sofern keine eigene Klasse zum Datenaustausch benötigt wird, kann auch direkt auf die Klasse EventArgs zurückgegriffen werden. Als Name für den Parameter wird in beiden Fällen üblicherweise der Buchstabe e verwendet, gängig sind allerdings auch ea, eventArgs und eventArguments.

Obwohl in der Framework Class Library durchgängig e verwendet wird, entspricht dies am wenigsten den Namenskonventionen von C#. Unter diesem Gesichtspunkt sollte am ehesten eventArguments eingesetzt werden.

C#

```
1 public delegate void Bar(object sender, EventArgs e);
2
3 public event Bar FooEvent;
```

Sofern ein Ereignis auf Grund einer Datenänderung auftritt, werden im ersten Parameter häufig so wohl der alte wie auch der neue Wert an alle ereignisbehandelnden Methoden übergeben. Diese haben dann die Möglichkeit, auf Basis dieser beiden Werte eigene Aktionen auszuführen. Gelegentlich wird dieser Parameter zudem eingesetzt, um die Ausführung des Ereignisses abzubrechen, indem eine entsprechende Eigenschaft namens Cancel auf *true* gesetzt wird, die schließlich vor der eigentlichen Ausführung des Ereignisses abgefragt wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Executes when storing begins.
 7     /// </summary>
 8     /// <param name="sender">The sender.</param>
 9     /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
```

```
13      /// <summary>
14      /// Executes when storing has finished.
15      /// </summary>
16      /// <param name="sender">The sender.</param>
17      /// <param name="e">The event arguments.</param>
18      public delegate void StoredEventHandler(
19          object sender, EventArgs eventArguments);
20
21      /// <summary>
22      /// Executes when restoring begins.
23      /// </summary>
24      /// <param name="sender">The sender.</param>
25      /// <param name="e">The event arguments.</param>
26      public delegate void RestoringEventHandler(
27          object sender, EventArgs eventArguments);
28
29      /// <summary>
30      /// Executes when restoring has finished.
31      /// </summary>
32      /// <param name="sender">The sender.</param>
33      /// <param name="e">The event arguments.</param>
34      public delegate void RestoredEventHandler(
35          object sender, EventArgs eventArguments);
36
37      /// <summary>
38      /// Represents a complex number.
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          #endregion
44
45          #region Events
46          /// <summary>
47          /// Fires when storing begins.
48          /// </summary>
49          public event StoringEventHandler Storing;
50
51          /// <summary>
52          /// Fires when storing has finished.
53          /// </summary>
54          public event StoredEventHandler Stored;
55
56          /// <summary>
57          /// Fires when restoring begins.
58          /// </summary>
59          public event RestoringEventHandler Restoring;
60
61          /// <summary>
62          /// Fires when restoring has finished.
63          /// </summary>
64          public event RestoredEventHandler Restored;
65          #endregion
66
```

```
67          #region Methods
68          // ...
69
70          /// <summary>
71          /// Stores the current instance in the specified
72          /// memento.
73          /// </summary>
74          /// <param name="memento">The memento.</param>
75          public void Store(IMemento memento)
76          {
77              // TODO gr: Store the current instance.
78              //          2007-06-25
79          }
80
81          /// <summary>
82          /// Restores the current instance from the specified
83          /// memento.
84          /// </summary>
85          /// <param name="memento">The memento.</param>
86          public void Restore(IMemento memento)
87          {
88              // TODO gr: Restore the current instance.
89              //          2007-06-25
90          }
91          #endregion
92
93          #region Constructors
94          #endregion
95      }
96  }
```

Obwohl die in diesem Beispiel verwendeten Ereignisse sämtlich objektgebunden sind, können Ereignisse mit Hilfe des Schlüsselwortes *static* wie auch Felder, Eigenschaften und Methoden als klassengebunden definiert werden. Klassengebundene Ereignisse erlauben es, auf Aktionen des gesamten Typs und nicht eines speziellen Objekts zu reagieren.

10.2 Auslösen von Ereignissen

Nachdem ein Ereignis definiert wurde, kann es ausgelöst werden. Prinzipiell geschieht dies, indem es wie eine Methode aufgerufen wird, wobei die gleichen Konventionen wie für den direkten Aufruf einer Methode oder eines Delegaten gelten. Intern wird dabei der Aufruf an den Delegaten weitergereicht, der dem Ereignis zugeordnet ist. Dieser wiederum löst – je nachdem, ob es sich um einen Singlecast- oder einen Multicast-Delegaten handelt, eine oder mehrere Methoden aus, die an den Delegaten gebunden worden sind.

Da ein Ereignis in der Regel von dem Objekt ausgelöst wird, das auch die Ursache für das Ereignis darstellt, wird zumeist *this* als erster Parameter angegeben.

Der zweite Parameter muss allerdings kontextbezogen erzeugt werden. Da sich dies aufwändiger gestalten kann, wird das Auslösen eines Ereignisses in eine eigene Methode ausgelagert, deren Aufruf sich an den entsprechenden Stellen dann deutlich kompakter als das direkte Auslösen des Ereignisses gestaltet.

Als Name trägt eine solche Methode den Namen des Ereignisses, ergänzt um das Präfix On. Die Methode, die also beispielsweise das Ereignis Stored auslöst, hieße OnStored. Häufig werden in der Praxis die Methoden, die auf ein Ereignis reagieren, derart benannt, was nach den Namensrichtlinien von C# allerdings falsch ist.

Da diese Methoden nur von innerhalb der Klasse ausgelöst werden sollten, werden sie in der Regel mit dem Zugriffsmodifizierer *protected* und zusätzlich mit dem Schlüsselwort *virtual* versehen. Dies geschieht, damit eine abgeleitete Klasse die ereignisauslösende Methode gegebenenfalls überschreiben kann. Im folgenden Beispiel ist die Klasse allerdings versiegelt, weshalb der Zugriffsmodifizierer *private* gewählt wurde.

Bevor ein Ereignis in einer solchen Methode aufgerufen wird, sollte zunächst noch geprüft werden, ob sich überhaupt Objekte zur Überwachung des Ereignisses registriert haben. Da der Delegat ansonsten *null* ist, würde der Aufruf ohne eine solche Prüfung ins Leere laufen und einen Fehler erzeugen, der zum Abbruch der Anwendung führt.

Obwohl noch nicht alle Konzepte vorgestellt wurden, die für diese Prüfung benötigt werden, wird sie an dieser Stelle dennoch eingeführt, da sie zum einen zwingend benötigt wird, zum anderen Ereignisaufrufe sich nur durch das konkrete, auszulösende Ereignis unterscheiden – der Rest folgt immer dem gleichen Schema. Nähere Informationen finden sich in den Kapiteln zu Operatoren und Anweisungen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Executes when storing begins.
 7     /// </summary>
 8     /// <param name="sender">The sender.</param>
 9     /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
13     /// <summary>
14     /// Executes when storing has finished.
15     /// </summary>
16     /// <param name="sender">The sender.</param>
17     /// <param name="e">The event arguments.</param>
18     public delegate void StoredEventHandler(
19         object sender, EventArgs eventArguments);
20
21     /// <summary>
22     /// Executes when restoring begins.
23     /// </summary>
```

```
24      /// <param name="sender">The sender.</param>
25      /// <param name="e">The event arguments.</param>
26      public delegate void RestoringEventHandler(
27          object sender, EventArgs eventArguments);
28
29      /// <summary>
30      /// Executes when restoring has finished.
31      /// </summary>
32      /// <param name="sender">The sender.</param>
33      /// <param name="e">The event arguments.</param>
34      public delegate void RestoredEventHandler(
35          object sender, EventArgs eventArguments);
36
37      /// <summary>
38      /// Represents a complex number.
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          #endregion
44
45          #region Events
46          /// <summary>
47          /// Fires when storing begins.
48          /// </summary>
49          public event StoringEventHandler Storing;
50
51          /// <summary>
52          /// Fires when storing has finished.
53          /// </summary>
54          public event StoredEventHandler Stored;
55
56          /// <summary>
57          /// Fires when restoring begins.
58          /// </summary>
59          public event RestoringEventHandler Restoring;
60
61          /// <summary>
62          /// Fires when restoring has finished.
63          /// </summary>
64          public event RestoredEventHandler Restored;
65          #endregion
66
67          #region Methods
68          // ...
69
70          /// <summary>
71          /// Raises the storing event.
72          /// </summary>
73          private void OnStoring()
74          {
75              // Check if there are any event handlers.
76              if(this.Storing != null)
77              {
```

```
78                  // Raise the storing event.
79                  this.Storing(this, null);
80              }
81          }
82
83          /// <summary>
84          /// Raises the stored event.
85          /// </summary>
86          private void OnStored()
87          {
88              // Check if there are any event handlers.
89              if(this.Stored != null)
90              {
91                  // Raise the stored event.
92                  this.Stored(this, null);
93              }
94          }
95
96          /// <summary>
97          /// Raises the restoring event.
98          /// </summary>
99          private void OnRestoring()
100         {
101             // Check if there are any event handlers.
102             if(this.Restoring != null)
103             {
104                 // Raise the restoring event.
105                 this.Restoring(this, null);
106             }
107         }
108
109         /// <summary>
110         /// Raises the restored event.
111         /// </summary>
112         private void OnRestored()
113         {
114             // Check if there are any event handlers.
115             if(this.Restored != null)
116             {
117                 // Raise the restored event.
118                 this.Restored(this, null);
119             }
120         }
121
122         /// <summary>
123         /// Stores the current instance in the specified
124         /// memento.
125         /// </summary>
126         /// <param name="memento">The memento.</param>
127         public void Store(IMemento memento)
128         {
129             // Raise the storing event.
130             this.OnStoring();
131
```

```
132             // TODO gr: Store the current instance.
133             //          2007-06-25
134
135             // Raise the stored event.
136             this.OnStored();
137         }
138
139         /// <summary>
140         /// Restores the current instance from the specified
141         /// memento.
142         /// </summary>
143         /// <param name="memento">The memento.</param>
144         public void Restore(IMemento memento)
145         {
146             // Raise the restoring event.
147             this.OnRestoring();
148
149             // TODO gr: Restore the current instance.
150             //          2007-06-25
151
152             // Raise the restored event.
153             this.OnRestored();
154         }
155         #endregion
156
157         #region Constructors
158         #endregion
159     }
160 }
```

10.3 Reagieren auf Ereignisse

Damit ein außenstehendes Objekt auf ein Ereignis reagieren kann, muss es eine Methode als ereignisbehandelnde Methode an dem Ereignis registrieren. Da Ereignisse intern nichts anderes als Delegaten sind, entspricht die Vorgehensweise zum Registrieren und Deregistrieren der zum Binden und Lösen von Methoden an Delegaten – der einzige Unterschied ist, dass für Ereignisse nur die Varianten mit den Operatoren += und -= zulässig sind. Eine direkte Zuweisung einer Methode oder das Zuweisen des Wertes *null* sind nicht möglich.

Als Namensrichtlinie gilt, dass eine ereignisbehandelnde Methode dem Namen des ereignisauslösenden Objekts, ergänzt um einen Unterstrich und den Namen des Ereignisses entspricht, wobei jeweils Pascal Case angewandt wird. Eine Methode, die das Ereignis Stored der Klasse ComplexNumber behandelt, hieße also ComplexNumber_Stored.

Kapitel 11
Generika

11.1 Was sind Generika?

Die Schnittstelle IMemento, die zum Speichern und Wiederherstellen von Daten dient, verfügt über einen eklatanten Nachteil: In der bislang verwendeten Form ist sie auf Daten vom Typ *float* beschränkt.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     public interface IMemento
 9     {
10         /// <summary>
11         /// Stores the specified value using the specified
12         /// key.
13         /// </summary>
14         /// <param name="key">The key.</param>
15         /// <param name="value">The value.</param>
16         void Store(string key, float value);
17
18         /// <summary>
19         /// Restores the value stored with the specified
20         /// key.
21         /// </summary>
22         /// <param name="key">The key.</param>
23         /// <returns>The value.</returns>
24         float Restore(string key);
25     }
26 }
```

Bei der Verwendung der Schnittstelle mit der Klasse ComplexNumber hat sich diese Einschränkung nicht ausgewirkt, da dort nur Daten vom Typ *float* verwen-

DOI: 10.1007/978-3-540-27889-4, © Springer 2008

det werden. Falls die Schnittstelle jedoch mehr Datentypen unterstützen soll, was spätestens dann benötigt wird, wenn die Schnittstelle allgemeingültig für zahlreiche verschiedene Klassen eingesetzt werden soll, macht sich diese Einschränkung deutlich bemerkbar.

Die einfachste Variante, die Schnittstelle um die benötigten Datentypen zu erweitern, liegt darin, die entsprechenden Methoden zu ergänzen. Bei der Methode Store bedeutet dies zwar einigen Aufwand, prinzipiell ist es aber überhaupt möglich, da sich die einzelnen überladenen Methoden im Typ des zweiten Parameters unterscheiden. Im folgenden Code wurde die Schnittstelle um eine Methode zum Speichern von Zeichenketten erweitert.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     public interface IMemento
 9     {
10         /// <summary>
11         /// Stores the specified value using the specified
12         /// key.
13         /// </summary>
14         /// <param name="key">The key.</param>
15         /// <param name="value">The value.</param>
16         void Store(string key, float value);
17
18         /// <summary>
19         /// Stores the specified value using the specified
20         /// key.
21         /// </summary>
22         /// <param name="key">The key.</param>
23         /// <param name="value">The value.</param>
24         void Store(string key, string value);
25
26         /// <summary>
27         /// Restores the value stored with the specified
28         /// key.
29         /// </summary>
30         /// <param name="key">The key.</param>
31         /// <returns>The value.</returns>
32         float Restore(string key);
33     }
34 }
```

Abgesehen von dem notwendigen Aufwand, eine prinzipiell immer gleiche Methode zu definieren, funktioniert dieser Ansatz bei der Methode Restore nicht: Da als Parameter immer ein *string* übergeben wird und sich die Methoden nur durch den Typ des Rückgabewertes unterscheiden würden, ist ein Überladen nicht möglich.

Als Ausweg bietet es sich an, den Typ des Rückgabewertes in den Methodennamen aufzunehmen, um die Methoden unterscheidbar zu machen.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Provides methods for memento classes.
    /// </summary>
    public interface IMemento
    {
        /// <summary>
        /// Stores the specified value using the specified
        /// key.
        /// </summary>
        /// <param name="key">The key.</param>
        /// <param name="value">The value.</param>
        void Store(string key, float value);

        /// <summary>
        /// Stores the specified value using the specified
        /// key.
        /// </summary>
        /// <param name="key">The key.</param>
        /// <param name="value">The value.</param>
        void Store(string key, string value);

        /// <summary>
        /// Restores the value stored with the specified
        /// key.
        /// </summary>
        /// <param name="key">The key.</param>
        /// <returns>The value.</returns>
        float RestoreAsFloat(string key);

        /// <summary>
        /// Restores the value stored with the specified
        /// key.
        /// </summary>
        /// <param name="key">The key.</param>
        /// <returns>The value.</returns>
        string RestoreAsString(string key);
    }
}
```

Auch wenn dieser Ansatz funktioniert, ist dies nicht sonderlich elegant. Seit C# 2.0 gibt es für derartige Probleme eine Lösung, nämlich die sogenannten generischen Datentypen, die kurz auch als Generika bezeichnet werden. Generika stellen immer dann eine gangbare elegante Lösung dar, wenn der gleiche Algorithmus oder

die gleiche Datenstruktur mehrfach implementiert werden muss, wobei sich die einzelnen Varianten nur durch den Typ der zu verarbeitenden Daten unterscheiden.

In einem solchen Fall ermöglichen es Generika, den Algorithmus oder die Datenstruktur nur ein einziges Mal implementieren zu müssen, ohne von vornherein einen konkreten Typ festzulegen. Statt dessen wird der Typ abstrahiert und an seiner Stelle ein Platzhalter eingefügt, der erst von dem Compiler durch den tatsächlichen Typ ersetzt wird. Da der Compiler den tatsächlichen Typ in den MSIL-Code schreibt, sind generische Datentypen trotz ihres abstrakten Ansatzes typsicher.

Der Platzhalter kann so wohl bei Klassen und Schnittstellen wie auch bei beliebigen Elementen wie Feldern, Eigenschaften oder Methoden eingesetzt werden und wird durch ein paar Spitzklammern begrenzt. Als Name wird üblicherweise der Buchstabe T, der als Kürzel für Type steht, verwendet. Falls mehr als ein Platzhalter benötigt wird, wird jeder einzelne Typparameter mit dem Buchstaben T als Suffix und einem folgenden Substantiv in Pascal Case benannt, wobei die zusätzlichen Platzhalter durch Kommata getrennt innerhalb der Spitzklammern aufgelistet werden.

Um also die Schnittstelle IMemento als generischen Datentyp zur Verfügung zu stellen, muss ihre Definition um den Platzhalter für den tatsächlich zu verarbeitenden Typ ergänzt werden. Innerhalb der Schnittstelle kann an Stelle der Typangabe dann der Platzhalter T verwendet werden. Der Typparameter wird dabei im XML-Kommentar mit Hilfe des Elementes typeparam beschrieben.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     /// <typeparam name="T">The type.</typeparam>
 9     public interface IMemento<T>
10     {
11         /// <summary>
12         /// Stores the specified value using the specified
13         /// key.
14         /// </summary>
15         /// <param name="key">The key.</param>
16         /// <param name="value">The value.</param>
17         void Store(string key, T value);
18
19         /// <summary>
20         /// Restores the value stored with the specified
21         /// key.
22         /// </summary>
23         /// <param name="key">The key.</param>
24         /// <returns>The value.</returns>
25         T Restore(string key);
26     }
```

```
27 }
```

Die Schnittstelle IMemento kann nun für beliebige Typen eingesetzt werden, indem sie über ihren Namen ergänzt um einen konkreten Typ angesprochen wird. Statt IMemento muss in der Schnittstelle IPersistable nun IMemento<*float*> angegeben werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for persisting an object.
 7     /// </summary>
 8     public interface IPersistable
 9     {
10         /// <summary>
11         /// Stores the current instance to the specified
12         /// memento.
13         /// </summary>
14         /// <param name="memento">The memento.</param>
15         void Store(IMemento<float> memento);
16
17         /// <summary>
18         /// Restores the current instance to the specified
19         /// memento.
20         /// </summary>
21         /// <param name="memento">The memento.</param>
22         void Restore(IMemento<float> memento);
23     }
24 }
```

Nachteilig an dieser Variante ist allerdings, dass nun für jeden einzelnen Datentyp eine eigene Schnittstelle IMemento mit dem jeweiligen Typ definiert werden muss. Daher kann der Typ auch nur für eine Methode angegeben werden, so dass die Schnittstelle IMemento nach wie vor allgemein gültig bleibt, ihre Methoden aber unter Angabe eines Typs aufgerufen werden müssen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Provides methods for memento classes.
 7     /// </summary>
 8     public interface IMemento
 9     {
10         /// <summary>
```

```
11          /// Stores the specified value using the specified
12          /// key.
13          /// </summary>
14          /// <typeparam name="T">The type.</typeparam>
15          /// <param name="key">The key.</param>
16          /// <param name="value">The value.</param>
17          void Store<T>(string key, T value);
18
19          /// <summary>
20          /// Restores the value stored with the specified
21          /// key.
22          /// </summary>
23          /// <typeparam name="T">The type.</typeparam>
24          /// <param name="key">The key.</param>
25          /// <returns>The value.</returns>
26          T Restore<T>(string key);
27      }
28
29      /// <summary>
30      /// Provides methods for persisting an object.
31      /// </summary>
32      public interface IPersistable
33      {
34          /// <summary>
35          /// Stores the current instance to the specified
36          /// memento.
37          /// </summary>
38          /// <param name="memento">The memento.</param>
39          void Store(IMemento<float> memento);
40
41          /// <summary>
42          /// Restores the current instance to the specified
43          /// memento.
44          /// </summary>
45          /// <param name="memento">The memento.</param>
46          void Restore(IMemento<float> memento);
47      }
48  }
```

11.2 Typparameter

Allen bisher verwendeten generischen Typparametern ist gemein, dass es keine Einschränkungen gibt, welche Typen an Stelle des Platzhalters eingesetzt werden können. Solche Typparameter werden daher auch als nicht gebundene oder ungebundene Typparameter bezeichnet. Allerdings verfügen ungebundene Typparameter – eben weil es keine Einschränkung der potenziellen Typen gibt – ihrerseits über einige Einschränkungen.

Unabhängig davon, dass der Typ bei ungebundenen Typparametern unbekannt ist, lassen sich auch keinerlei Annahmen über die Art des Typs machen: Es ist unbe-

kannt, welche Schnittstellen dieser Typ implementiert, es ist unbekannt, ob der Typ von einer bestimmten Basisklasse ableitet, es ist nicht einmal bekannt, ob es sich bei dem Typ um einen Verweis- oder einen Wertetyp handelt.

In einigen Fällen kann es erforderlich sein, die potenziellen Typen einzuschränken. Dazu dient in C# das Schlüsselwort *where*, mit dem zusätzliche Angaben zu einem Typ gemacht werden können. Typparameter, die mit diesem Schlüsselwort näher spezifiziert wurden, werden als gebundene Typparameter bezeichnet.

Sofern mehr als ein Typparameter verwendet wird, muss für jeden dieser Typparameter, der gebunden werden soll, ein eigenes *where* angegeben werden.

Die einfachste Variante einer Typeinschränkung gibt an, ob es sich bei dem Typparameter um einen Verweis- oder einen Wertetyp handelt. Für Wertetypen wird als Basisklasse des Typparameters das Schlüsselwort *struct* angegeben, für Verweistypen *class*.

C#

```
1 public void Foo<T> where T : class
2 {
3 }
```

Ebenso kann an Stelle des Schlüsselwortes *class* auch eine konkrete Klasse oder Schnittstelle angegeben werden, welcher der Typparameter entsprechen muss. Wie bei der Vererbung von Klassen können mehrere Schnittstellen angegeben werden, zudem können sie mit der Angabe einer Klasse kombiniert werden. In diesem Fall werden die einzelnen Angaben durch Kommata getrennt.

C#

```
1 public void Foo<T> where T : Bar, IBar1, IBar2
2 {
3 }
```

Schließlich kann der Ausdruck new() angegeben werden, um zu definieren, dass der Typparameter über einen öffentlichen parameterlosen Konstruktor verfügen muss. Falls dieser Ausdruck angegeben wird, muss er als letzter angegeben werden.

C#

```
1 public void Foo<T> where T : class, new()
2 {
3 }
```

Als Spezialfall gibt es des weiteren noch Typparameter, die wiederum durch einen Typparameter eingeschränkt werden, indem dieser weitere Typparameter beispielsweise als notwendige Basisklasse angegeben wird. Solche Typeinschränkungen werden als naked bezeichnet.

C#

```
1 public void Foo<TDerived, TBase> where TDerived : TBase
```

```
2 {
3 }
```

Da bei einem Typparameter nicht notwendigerweise bekannt ist, ob es sich um einen Verweis- oder einen Wertetyp handelt, ist es nicht möglich, ihn mit dem Standardwert zu initialisieren. Um einen Typparameter dennoch mit dem Standardwert seines Typs initialisieren zu können, gibt es das Schlüsselwort *default*, das wie eine Methode verwendet wird, und dem als Parameter der entsprechende Typ übergeben werden muss.

C#

```
1 public T Foo<T>()
2 {
3     return default(T);
4 }
```

Neben Schnittstellen und Methoden können auch Klassen, Strukturen und Delegaten mit Typparametern versehen werden.

11.3 Lambdaausdrücke

Generika eignen sich jedoch nicht nur dazu, Typen mit Hilfe von Typparametern flexibel gestalten zu können, sie ermöglichen auch die Definition von Lambdaausdrücken während der Ausführung. Dazu bietet C# seit der Version 3.0 den vorgefertigten Delegaten Func im Namensraum System an, dem als Typparameter die Typen der Parameter und des Rückgabewertes des zu erzeugenden Lambdaausdrucks übergeben werden.

Soll beispielsweise ein Lambdaausdruck definiert werden, der eine komplexe Zahl in ihren Absolutbetrag überführt, so ist dies mit Hilfe dieses Delegaten möglich. Als Typparameter werden in diesem Fall die Klasse ComplexNumber sowie *float* als Typ des Absolutbetrags angegeben.

C#

```
1 Func<ComplexNumber, float> GetAbsoluteValue =
2     (c => c.AbsoluteValue);
```

Die auf diese Art erzeugte Delegatinstanz kann im weiteren Verlauf wie jeder andere Delegat aufgerufen werden. Sollen nicht nur ein, sondern mehrere Parameter angegeben werden, müssen diese zum einen dem Delegaten Func wie auch innerhalb des Lambdaausdrucks kommasepariert innerhalb von runden Klammern angegeben werden.

C#

```
1 Func<ComplexNumber, ComplexNumber, ComplexNumber> Add =
2     ((c1, c2) => c1 + c2);
```

Kapitel 12
Nullbare Wertetypen

12.1 Was sind nullbare Wertetypen?

Neben Verweis- und Wertetypen verfügt C# seit der Version 2.0 über eine weitere Art von Typen, die nullbaren Wertetypen. Diese entsprechen einem Hybriden zwischen Verweis- und Wertetypen, da sie in ihrer Funktion den Wertetypen entsprechen, zusätzlich allerdings den Wert *null* annehmen können, der üblicherweise Verweistypen vorbehalten ist.

Mit nullbaren Wertetypen ist es beispielsweise möglich, den Wert eines Wertetyps als unbekannt zu kennzeichnen. Ohne die Möglichkeit, *null* zuordnen zu können, müsste dafür ein konkreter Wert verwendet werden, wie beispielsweise die Zahl Null oder eine leere Zeichenkette. Allerdings entfiele in diesem Fall die Möglichkeit, zwischen dem tatsächlichen Wert Null beziehungsweise der leeren Zeichenkette und einem unbekannten Wert zu unterscheiden.

Intern werden nullbare Wertetypen durch einen Verweistyp dargestellt, indem dieser als Container für den Wertetyp dient und zusätzliche Eigenschaften bereitstellt, um mit dem Wert *null* umgehen zu können.

Definiert wird ein nullbarer Wertetyp, indem an die Typdefinition ein ? angehängt wird. Um die Klasse ComplexNumber derart zu erweitern, dass der Real- und der Imaginärteil einer komplexen Zahl der Wert *null* angegeben werden kann, muss in den entsprechenden Definitionen der Typ *float?* an Stelle von *float* verwendet werden.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     /// <summary>
6     /// Executes when storing begins.
7     /// </summary>
8     /// <param name="sender">The sender.</param>
9     /// <param name="e">The event arguments.</param>
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

```
10      public delegate void StoringEventHandler(
11          object sender, EventArgs eventArguments);
12
13      /// <summary>
14      /// Executes when storing has finished.
15      /// </summary>
16      /// <param name="sender">The sender.</param>
17      /// <param name="e">The event arguments.</param>
18      public delegate void StoredEventHandler(
19          object sender, EventArgs eventArguments);
20
21      /// <summary>
22      /// Executes when restoring begins.
23      /// </summary>
24      /// <param name="sender">The sender.</param>
25      /// <param name="e">The event arguments.</param>
26      public delegate void RestoringEventHandler(
27          object sender, EventArgs eventArguments);
28
29      /// <summary>
30      /// Executes when restoring has finished.
31      /// </summary>
32      /// <param name="sender">The sender.</param>
33      /// <param name="e">The event arguments.</param>
34      public delegate void RestoredEventHandler(
35          object sender, EventArgs eventArguments);
36
37      /// <summary>
38      /// Represents a complex number.
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          // ...
44
45          /// <summary>
46          /// Gets or sets the real part.
47          /// </summary>
48          /// <value>The real part.</value>
49          public float? RealPart
50          {
51              get
52              {
53                  return this._realPart;
54              }
55
56              set
57              {
58                  this._realPart = value;
59              }
60          }
61
62          /// <summary>
63          /// Gets or sets the imaginary part.
```

```
64          /// </summary>
65          /// <value>The imaginary part.</value>
66          public float? ImginaryPart
67          {
68              get
69              {
70                  return this._imaginaryPart;
71              }
72
73              set
74              {
75                  this._imaginaryPart = value;
76              }
77          }
78          #endregion
79
80          #region Events
81          #endregion
82
83          #region Methods
84          #endregion
85
86          #region Constructors
87          /// <summary>
88          /// Initializes a new instance of the ComplexNumber
89          /// type using default values.
90          /// </summary>
91          public ComplexNumber()
92              : this(null, null)
93          {
94          }
95
96          /// <summary>
97          /// Initializes a new instance of the ComplexNumber
98          /// type using the specified real value.
99          /// </summary>
100         /// <param name="realPart">The real part.</param>
101         public ComplexNumber(float? realPart)
102             : this(realPart, null)
103         {
104         }
105
106         /// <summary>
107         /// Initializes a new instance of the ComplexNumber
108         /// type using the specified real and imaginary
109         /// values.
110         /// </summary>
111         /// <param name="realPart">The real part.</param>
112         /// <param name="imaginaryPart">The imaginary
113         /// part.</param>
114         public ComplexNumber(
115             float? realPart, float? imaginaryPart)
116         {
117             // Set default values for the real and
```

```
118             // imaginary part.
119             this.RealPart = realPart;
120             this.ImaginaryPart = imaginaryPart;
121         }
122         #endregion
123     }
124 }
```

Da die Typen *float* und *float?* für C# verschieden sind, müssen nicht nur die Definitionen der Felder, sondern auch die der zugehörigen Eigenschaften, Methoden und Konstruktoren angepasst werden.

Insbesondere in den Konstruktoren muss entschieden werden, mit welchen Standardwerten die Felder initialisiert werden sollen – bislang war es die Zahl Null, in der neuen Version werden die Felder statt dessen mit *null* initialisiert, falls kein konkreter Wert angegeben wird. Dies entspricht der Bedeutung des Literals *null*, dass der eigentliche Wert nämlich nicht bekannt ist.

Kapitel 13
Enumerationen

13.1 Was sind Enumerationen?

Häufig besteht Bedarf, für einen Parameter einer Methode nur eine gewisse Auswahl an vorgegebenen Werten zuzulassen. Handelt es sich dabei nur um einen Wahrheitswert, also einen Wert, der entweder wahr oder falsch ist, bietet sich dafür als Typ *bool* an. Dieser Typ sollte allerdings nur verwendet werden, wenn *true* und *false* tatsächlich die beiden Alternativen darstellen.

In der Praxis wird *bool* häufig auch dann verwendet, wenn nur eine der beiden Alternativen selbstbeschreibend ist. Der Konstruktor der Klasse ComplexNumber könnte beispielsweise derart erweitert werden, dass ihm ein Parameter vom Typ *bool* übergeben wird, der angibt, ob es sich bei der zu initialisierenden komplexen Zahl um eine konjugierte Zahl handelt. In diesem Fall entsprechen die möglichen Werte *true* und *false* den beiden Alternativen, da eine komplexe Zahl entweder konjugiert ist oder nicht.

In einem anderen Fall könnte die Schnittstelle IPersistent um einen Parameter erweitert werden, der angibt, ob die Daten auf Festplatte geschrieben und von dort wieder geladen werden sollen. Hierfür ergibt der Typ *bool* wenig Sinn, denn *true* gibt als Wert zwar an, dass die Festplatte verwendet werden soll, aber die Angabe von *false* ist sinnlos – es wird zwar festgelegt, dass die Festplatte nicht verwendet werden soll, die Angabe des Speicherortes ist aber nicht gegeben.

Für diese Fälle, in denen mehr als eine Option angegeben werden sollen, verfügt C# über sogenannte Enumerationen. Eine Enumeration kann verschiedene Werte enthalten, die über ihren jeweiligen Namen angesprochen werden können. Definiert wird eine Enumeration mit Hilfe des Schlüsselwortes *enum*, wobei die einzelnen Werte kommasepariert innerhalb geschweifter Klammern aufgezählt werden. Im Gegensatz zu einer Klasse muss allerdings hinter der schließenden geschweiften Klammer ein Semikolon angegeben werden.

Die Namenskonventionen für Enumerationen entsprechen prinzipiell denen von Klassen, allerdings wird der Name einer Enumeration im Plural angegeben.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Contains storage locations for the persistence
 7     /// interface.
 8     /// </summary>
 9     public enum StorageLocations
10     {
11         /// <summary>
12         /// The disk as storage location.
13         /// </summary>
14         Disk,
15
16         /// <summary>
17         /// A databse as storage location.
18         /// </summary>
19         Database
20     };
21 }
```

Der Vorteil in der Verwendung einer Enumeration an Stelle einer Zahl oder einer Zeichenkette zur Identifikation des Wertes liegt zum einen in der Verständlichkeit, da der entsprechende Wert über seinen Namen angesprochen wird, zum anderen in der Überprüfbarkeit durch den Compiler. Ein Schreibfehler eines Wertes aus einer Enumeration wird vom Compiler entdeckt, während ein Schreibfehler in einer Zeichenkette erst zur Laufzeit durch einen auftretenden Fehler entdeckt wird.

Intern wird eine Enumeration allerdings durch den Datentyp *int* repräsentiert, wobei die einzelnen Werte der Enumeration von Null beginnend nummeriert werden. Dieses standardmäßige Verhalten kann allerdings überschrieben werden, indem einem oder mehreren Werten explizit eine Ganzzahl zugeordnet wird. Alle Werte, denen keine eigene Zahl zugeordnet wird, erhalten dabei als interne Repräsentation eine automatisch inkrementierte Nummer.

Im folgenden Beispiel beginnt die Enumeration bei eins, da für den Wert Database keine eigene Repräsentation angegeben wird, erhält er automatisch den nächsthöheren Wert, nämlich zwei.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     /// <summary>
6     /// Contains storage locations for the persistence
7     /// interface.
8     /// </summary>
9     public enum StorageLocations
```

```
10      {
11          /// <summary>
12          /// The disk as storage location.
13          /// </summary>
14          Disk = 1,
15
16          /// <summary>
17          /// A databse as storage location.
18          /// </summary>
19          Database
20      };
21  }
```

Anders als Klassen leiten Enumerationen nicht direkt von dem Typ *object* ab, sondern von der Struktur Enum im Namensraum System.

Kapitel 14
Variablen

14.1 Was sind Variablen?

Die bisher einzige Möglichkeit, Daten zu speichern, besteht in der Verwendung von Feldern. Diese sind dann nützlich, wenn die entsprechenden Daten relevant für den Status des Objektes an sich sind. Allerdings besteht manchmal die Notwendigkeit, Daten temporär zu speichern, wenn diese beispielsweise als Zwischenergebnis einer Berechnung für eine spätere Verarbeitung zur Verfügung stehen sollen, nach dem Abschluss der Berechnung aber nicht mehr benötigt werden.

Für diese Fälle verfügt C# über ein ähnliches Konzept wie Felder, nämlich Variablen. Im Gegensatz zu Feldern werden Variablen allerdings nicht innerhalb eines Typs, sondern innerhalb einer Methode definiert und stehen dort auch nur so lange zur Verfügung, wie die Methode ausgeführt wird. Deshalb werden sie auch als lokale Variablen bezeichnet.

Nachdem die Ausführung der Methode, welche die lokalen Variablen enthält, beendet wurde, wird der Speicher der lokalen Variablen wieder freigegeben, wodurch diese ihren Wert verlieren und sich beim nächsten Aufruf der Methode wieder derart verhalten, als wären sie noch nie verwendet worden.

Da der Zugriff auf lokale Variablen nur aus der Methode möglich ist, welche die lokalen Variablen enthält, wird bei deren Deklaration auf die Angabe eines Zugriffsmodifizierers verzichtet. Wie bei Feldern kann auch lokalen Variablen ein Standardwert zugewiesen werden. Falls ein Standardwert für eine lokale Variable angegeben wird, wird ihre Erzeugung als Definition bezeichnet, andernfalls als Deklaration.

Als Namenskonventionen gelten die Regeln von Feldern, mit der Ausnahme, dass auf den führenden Unterstrich verzichtet wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

```
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Declare the mathematical constant pi.
16             double pi;
17         }
18     }
19 }
```

Nachdem eine Variable deklariert wurde, kann auf sie und damit auf ihren Wert zugegriffen werden. Die Zuweisung eines neuen Wertes erfolgt analog der Zuweisung eines Wertes an ein Feld mit Hilfe des Operators =.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define the mathematical constant pi.
16             double pi = 3.1415926;
17         }
18     }
19 }
```

Mit Variablen ist es nun auch möglich, den Rückgabewert von Methoden zu verarbeiten, indem bei der Zuweisung an Stelle eines konkreten Wertes der Methodenaufruf angegeben wird. In diesem Fall wird zunächst die Methode aufgerufen und ausgeführt und anschließend ihr Rückgabewert der Variablen zugewiesen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
```

```
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // TODO gr: Create a complex number.
16              //          2007-07-10
17
18              // Determine the absolute value and assign it
19              // to a local variable.
20              float absoluteValue =
21                  complexNumber.AbsoluteValue;
22          }
23      }
24  }
```

Die lokale Variable kann im folgenden Verlauf der Methode verwendet werden, um weitere Berechnungen auszuführen, oder um ihren Wert auf die Konsole auszugeben. Zu diesem Zweck enthält die Framework Class Library die Klasse Console im Namensraum System, deren Methode WriteLine den Wert des übergebenen Parameters auf der Konsole ausgibt.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // TODO gr: Create a complex number.
16              //          2007-07-10
17
18              // Determine the absolute value and assign it
19              // to a local variable.
20              float absoluteValue =
21                  complexNumber.AbsoluteValue;
22
23              // Print the absolute value to the console.
24              Console.WriteLine(absoluteValue);
25          }
26      }
27  }
```

Variablen eignen sich jedoch nicht nur dazu, den Rückgabewert eines einfachen Methodenaufrufs aufzunehmen. Mit ihrer Hilfe kann eine weitere, neue Art von Methoden definiert werden, die vorher nicht möglich war: Rekursive Methoden. Dabei handelt es sich um Methoden, die sich intern selbst aufrufen, um ihren Rückgabewert zu berechnen.

Ein bekanntes Beispiel für eine rekursive Berechnung ist die Folge der Fibonacci-Zahlen. In dieser Folge werden nur für die beiden ersten Elemente die Werte 0 und 1 vorgegeben, alle folgenden Elemente berechnen sich aus der Summe ihrer beiden Vorgänger. Das dritte Element entspricht also der Summe aus 0 und 1, das vierte Element der Summe aus 1 und 1, das fünfte der Summe aus 1 und 2, ...

Hierdurch ergibt sich die Folge:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

Diese Folge lässt sich rekursiv berechnen, da die n-te Fibonacci-Zahl der Summe aus der n-1-ten und n-2-ten Fibonacci-Zahl entspricht, wobei diese wiederum aus ihren Vorgängern berechnet werden können. Prinzipiell folgt eine Methode zur Berechnung der Fibonacci-Zahlen also dem folgenden Schema:

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Calculates the n-th Fibonacci number.
12         /// </summary>
13         public int CalculateFibonacci(int n)
14         {
15             // Declare a variable for the sum of the
16             // predecessors.
17             int sum;
18
19             // TODO gr: Calculate the number by adding
20             //          its predecessors.
21             //          2007-07-17
22
23             // Return the sum to the caller.
24             return sum;
25         }
26     }
27 }
```

Würde man diese Methode allerdings in dieser Form aufrufen, käme es zu einem Überlauf im Methodenstapel, da sich die Methode ohne Abbruch immer wieder selbst aufriefe und somit in eine endlose Schleife geriete. Die Lösung stellt ein Ab-

bruchkriterium dar, das im Fall von n gleich 1 oder 2 die entsprechend definierten Startwerte der Fibonacci-Folge zurückgibt.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Calculates the n-th Fibonacci number.
12         /// </summary>
13         public int CalculateFibonacci(int n)
14         {
15             // TODO gr: Check whether the first or the
16             //          second Fibonacci number is requested.
17             //          If so, return the appropriate values.
18             //          2007-07-17
19
20             // Declare a variable for the sum of the
21             // predecessors.
22             int sum;
23
24             // TODO gr: Calculate the number by adding its
25             //          predecessors.
26             //          2007-07-17
27
28             // Return the sum to the caller.
29             return sum;
30         }
31     }
32 }
```

Prinzipiell kann eine lokale Variable an jeder beliebigen Stelle einer Methode definiert werden, sofern die Deklaration vor der ersten Verwendung der Variablen statt findet. Es gilt allerdings als guter Stil, eine lokale Variable so spät wie möglich vor ihrer ersten Verwendung zu definieren.

Da die Variable sum, welche die Summe der beiden vorangegangenen Fibonacci-Zahlen aufnimmt, erst ab der Berechnung der dritten Fibonacci-Zahl benötigt wird, wird sie erst nach der entsprechenden Prüfung definiert.

14.2 Zuweisungen an Variablen

Bislang wurde bereits einige Male der Operator = verwendet, um einem Feld oder einer Variablen einen Wert zuzuweisen, weshalb dieser Operator als Zuweisungs-

operator bezeichnet wird. Bei einer Zuweisung wird immer der Wert rechts des Operators dem Element zu seiner Linken zugeordnet. Bisher wurde bei den Zuweisungen allerdings immer nur auf Wertetypen zugegriffen.

Die Zuweisung an Verweistypen funktioniert prinzipiell gleich: Einem Element auf der linken Seite kann ein auf der rechten Seite des Operators stehendes Objekt zugewiesen werden. Allerdings muss – damit ein Objekt zugewiesen werden kann – zunächst ein Objekt erzeugt werden. Es wurde bereits erwähnt, dass die entsprechende Methode, die bei der sogenannten Instanziierung von Objekten ausgeführt wird, der Konstruktor der entsprechenden Klasse ist.

Um also eine neue Instanz zu erzeugen, muss der Konstruktor aufgerufen werden, dem allerdings zusätzlich das Schlüsselwort *new* vorangestellt wird. Obwohl ein Konstruktor nicht über einen Rückgabewert verfügt, wird durch das Schlüsselwort *new* ein Verweis auf das neu erzeugte Objekt zurückgegeben, der in einem Element gespeichert werden kann.

Um also ein Objekt der Klasse ComplexNumber zu erzeugen, müsste der Aufruf folgendermaßen lauten:

C#

```
1 new ComplexNumber();
```

Falls der neu erzeugten komplexen Zahl direkt Werte für den Real- und den Imaginärteil zugewiesen werden sollen, können diese als Parameter übergeben werden – vorausgesetzt, es wurde ein entsprechender Konstruktor definiert.

C#

```
1 new ComplexNumber(23, 42);
```

Damit schließlich der Verweis auf das neu erzeugte Objekt gespeichert wird, muss die Anweisung noch um eine Zuweisung an eine Variable ergänzt werden, die zuvor deklariert werden muss.

C#

```
1 ComplexNumber myNumber;
2 myNumber = new ComplexNumber(23, 42);
```

Alternativ kann, wie bei Wertetypen auch, die Deklaration mit einer Zuweisung zu einer Definition verbunden werden:

C#

```
1 ComplexNumber myNumber = new ComplexNumber(23, 42);
```

Mit der Möglichkeit, Objekte erzeugen zu können, lässt sich die Klasse ComplexNumber nun auch in einer Anwendung nutzen, um beispielsweise die Summe zweier komplexer Zahlen zu berechnen.

C#

```
using System

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public static class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Create two complex numbers.
            ComplexNumber first = new ComplexNumber(23, 42);
            ComplexNumber second = new ComplexNumber(17, 2);

            // Add the second number to the first one.
            first.Add(second);

            // Print the result to the console.
            Console.WriteLine(first.Real);
            Console.WriteLine(first.Imaginary);
        }
    }
}
```

Die Zuweisung an nullbare Wertetypen hingegen funktioniert wiederum so wie die Zuweisung an normale Wertetypen. Der einzige Unterschied liegt darin, dass nullbaren Wertetypen das Literal *null* zugewiesen werden kann, was bei normalen Wertetypen nicht möglich ist.

Seit der Version 3.0 von C# gibt es mit Hilfe der sogenannten Objektinitialisierer eine weitere Möglichkeit, Objekte zu erzeugen. Mit Objektinitialisierern ist es nicht mehr nötig, für jede potenzielle Initialisierung einen eigenen Konstruktur bereitzustellen. Statt dessen werden die zu initialisierenden Eigenschaften und ihre Werte direkt beim Aufruf von *new* mit angegeben. An Stelle von

C#

```
// Create an instance of the Person type.
Person person = new Person();

// Set the values.
person.LastName = "Roden";
person.FirstName = "Golo";
person.EMail = "webmaster@goloroden.de";
```

kann mit Hilfe von Objektinitialisierern also auch

C#

```
// Create an instance of the Person type and set its
// values.
Person person = new Person {
    LastName = "Roden", FirstName = "Golo",
    EMail = "webmaster@goloroden.de" };
```

geschrieben werden. Um das Ganze noch weiter zu vereinfachen, kann sogar die Angabe des Typs entfallen. C# erzeugt in diesem Fall im Hintergrund einen passenden Typ, dessen Name dem Entwickler nicht bekannt ist, und der deshalb als anonymer Typ bezeichnet wird. Um ein solches Objekt eines anonymen Typs in einer Variablen speichern zu können, gibt es das Schlüsselwort *var*.

C#

```
// Create a new instance of an anonymous type for persons
// and set its values.
var person =
    new { LastName = "Roden", FirstName = "Golo",
        EMail = "webmaster@goloroden.de" };
```

Obwohl der Typ in diesem Beispiel dem Entwickler nicht bekannt ist, ist der Zugriff auf das Objekt trotzdem typsicher. *var* steht also nicht austauschbar für jeden beliebigen Typ, sondern leitet den zu verwendenden Typ aus dem Ausdruck auf der rechten Seite des Zuweisungsoperators ab.

Wird ein Typ hergeleitet, dessen Eigenschaften namentlich und von ihrem Typ einem bestehenden Typ entsprechen, wird dieser Typ verwendet. Es wird also nicht bei jedem Aufruf von *new* ohne Angabe eines Typs ein neuer Typ erzeugt, sondern nur dann, wenn kein passender Typ gefunden wird.

Das Schlüsselwort *var* kann prinzipiell auch für eingebaute Typen verwendet werden, so kann an Stelle der Zeile

C#

```
// Initialize a variable of type int.
int i = 23;
```

auch die Zeile

C#

```
// Initialize a variable of type int by using type
// inference.
var i = 23;
```

verwendet werden. In beiden Fällen wird eine Variable des Typs *int* erzeugt. Zu beachten ist bei anonymen Typen, dass ihr Einsatz nur für lokale Variablen möglich ist, sie können insbesondere nicht als Rückgabewert für Methoden verwendet werden.

Kapitel 15
Arrays

15.1 Was sind Arrays?

Felder ermöglichen zwar das Speichern von Daten, aber ein Feld kann jeweils nur einen einzelnen Wert aufnehmen. Besteht die Notwendigkeit, mehrere gleichartige Werte speichern zu müssen, so müssen mehrere Felder definiert werden. Insbesondere bei einer hohen Anzahl an Werten neigt dieses Verfahren aber dazu, unübersichtlich zu werden. Außerdem sind Fälle denkbar, in denen nicht bereits zur Entwicklungszeit bekannt ist, wie viele Werte gespeichert werden sollen, da sich dies erst zur Laufzeit ergibt.

Die Lösung für dieses Problem stellen sogenannte Arrays dar, die mehrere Werte eines Typs aufnehmen können. Anstatt also jeden Wert in einem eigenen Feld zu speichern, wird statt dessen ein Array als Feld angelegt, dessen Dimension ausreichend ist, um alle Werte aufzunehmen.

Der Typ eines Arrays besteht aus dem Typ der Daten, die das Array aufnehmen soll, dem ein Paar eckige Klammern folgen. Da ein Array ein Verweistyp ist, wird es ebenso wie ein Objekt mit Hilfe des Schlüsselwortes *new* erzeugt, wobei dort innerhalb eckiger Klammern die Größe des Arrays definiert wird. Im Gegensatz zu den übrigen Klassen leitet ein Array allerdings nicht direkt von *object* ab, sondern von der Klasse System.Array.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

```
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Define an array for fibonacci numbers.
16              int[] fibonacci = new int[10];
17          }
18      }
19  }
```

Auf die einzelnen Elemente des Arrays kann im weiteren Verlauf der Anwendung wiederum mit Hilfe der eckigen Klammern zugegriffen werden, indem in ihnen der Index des Elements angegeben wird, auf das zugegriffen werden soll. In C# beginnen Indizes von Arrays immer bei Null, das heißt, der höchste Index in einem Array mit n Elementen trägt die Nummer n-1.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Define an array for fibonacci numbers.
16              int[] fibonacci = new int[10];
17
18              // Initialize the array with the first two
19              // numbers.
20              fibonacci[0] = 1;
21              fibonacci[1] = 1;
22
23              // Calculate all following numbers.
24              for(int i = 2; i < 10; i++)
25              {
26                  fibonacci[i] =
27                      fibonacci[i - 1] + fibonacci[i - 2];
28              }
29          }
30      }
31  }
```

Da es aufwändig und unübersichtlich sein kann, Arrays auf diese Art zu initialisieren, können die enthaltenen Werte auch direkt innerhalb geschweifter Klammern

angegeben werden. In diesem Fall entfällt die Angabe der Größe des Arrays, sie wird statt dessen aus der Anzahl der übergebenen Werte ermittelt.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Define an array for fibonacci numbers and
            // fill it with  numbers.
            int[] fibonacci =
               new int[] { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
        }
    }
}
```

Arrays, die wie die bisherigen Beispiele einer Aufzählung von Werten entsprechen, werden auch als eindimensionale Arrays bezeichnet, da ihre Elemente nur über einen Index verfügen. In C# können Arrays jedoch auch mehrdimensional definiert werden, so dass beispielsweise bei zwei Dimensionen eine Tabelle und bei dreien ein Würfel von Daten entsteht.

Um ein Array mit mehreren Indizes auszustatten, genügt es, bei seiner Definition mehrere Größen anzugeben, die jeweils durch ein Komma voneinander getrennt werden. Hierbei muss allerdings beachtet werden, dass die Kommata innerhalb der eckigen Klammern der Typdefinition ebenfalls angegeben werden müssen. Im folgenden Beispiel wird ein Schachbrett als Feld von acht mal acht Feldern definiert, auf dem Figuren platziert werden können.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Contains chess figures.
    /// </summary>
    public enum ChessFigure
    {
        /// <summary>
        /// The figure castle.
```

```
12          /// </summary>
13          Castle,
14
15          /// <summary>
16          /// The figure knight.
17          /// </summary>
18          Knight
19
20          // TODO gr: Define the other chess figures.
21          //          2008-01-03
22      }
23
24      /// <summary>
25      /// Represents the application class.
26      /// </summary>
27      public class Program
28      {
29          /// <summary>
30          /// Executes the application.
31          /// </summary>
32          public static void Main()
33          {
34              // Create a chess board.
35              ChessFigure[,] chessBoard =
36                  new ChessFigure[8, 8];
37
38              // Put chessmen onto the board.
39              chessBoard[0, 0] = ChessFigure.Castle;
40              chessBoard[0, 1] = ChessFigure.Knight;
41          }
42      }
43  }
```

Neben der Möglichkeit, Arrays ein- oder mehrdimensional zu definieren, besteht zusätzlich die Option, Arrays ineinander zu verschachteln. Prinzipiell entspricht ein verschachteltes Array einem mehrdimensionalen Array, allerdings können verschachtelte Arrays beispielsweise für jede einzelne Zeile eine individuelle Anzahl an Spalten definieren.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents the application class.
7      /// </summary>
8      public class Program
9      {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
```

```
13          public static void Main()
14          {
15              // Create a nested array with two lines and
16              // three columns in the first row and two
17              // columns in the second row.
18              string[] colors = new string[2];
19              colors[0] = new string[3];
20              colors[1] = new string[2];
21
22              // Fill the array.
23              colors[0][0] = "Blau";
24              colors[0][1] = "Blue";
25              colors[0][2] = "Bleu";
26              colors[1][0] = "Rot";
27              colors[1][1] = "Red";
28          }
29      }
30  }
```

Bei verschachtelten Arrays ist zu beachten, dass die einzelnen Dimensionen in jeweils einem eigenen Paar eckiger Klammern angegeben werden, und die Dimensionen nicht wie bei den mehrdimensionalen Arrays kommasepariert sind.

Der Einsatz von Arrays ermöglicht nicht nur, mehrere Werte in einer einzelnen Variablen beziehungsweise einem Feld zu speichern, sondern auch, Parameter von der Kommandozeile an die Methode Main zu übergeben. Als Parameter kann für diese in C# nämlich ein Array von Strings angegeben werden, das die einzelnen auf der Kommandozeile angegebenen Parameter enthält.

Auf die einzelnen Elemente kann analog zu den Elementen aller anderen Arrays mit Hilfe des Indizes zugegriffen werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         /// <param name="arguments">The arguments.</param>
14         public static void Main(string[] arguments)
15         {
16             Console.WriteLine(
17                 "The first argument is " + arguments[0]);
18         }
19     }
20 }
```

Arrays können außerdem dazu eingesetzt werden, eine vorher nicht festgelegte Anzahl von Parametern an eine Methode zu übergeben, indem die Werte in ein Array verpackt werden, und nur dieses Array übergeben wird. Da der Typ des Arrays nicht die Größenangabe enthält, kann das übergebene Array eine beliebige Größe haben.

Allerdings kann es aufwändig sein, zur Übergabe einiger Parameter ein Array erzeugen zu müssen. Daher stellt C# das Schlüsselwort *params* zur Verfügung, mit dem ein Array alternativ auch als Liste einzelner Werte übergeben werden kann. Sofern das Schlüsselwort *params* einem Parameter vorangestellt wird, muss dieser Parameter zum einen ein Array sein, zum zweiten dürfen ihm keine weiteren Parameter folgen, und er muss der einzige Parameter sein, der über das *params*-Schlüsselwort verfügt.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents a foo class.
    /// </summary>
    public class Foo
    {
        /// <summary>
        /// Sorts the specified numbers.
        /// </summary>
        /// <param name="numbers">An array of
        /// numbers.</param>
        /// <returns>A sorted array of the specified
        /// numbers.</returns>
        public int[] Sort(params int[] numbers)
        {
            // TODO gr: Sort the numbers and return them
            //          as int array to the caller.
            //          2008-01-03
        }
    }
}
```

Wird eine solche Methode aufgerufen, so kann ihr an Stelle eines Arrays

C#

```
this.Sort(new int[] { 3, 13, 1, 8, 1, 5, 2, 34, 21, 55 });
```

auch eine Auflistung einzelner Zahlen übergeben werden.

C#

```
this.Sort(3, 13, 1, 8, 1, 5, 2, 34, 21, 55);
```

15.2 Indexer

Verwandt mit Arrays sind die sogenannten Indexer, die den indizierten Zugriff auf eine Klasse ermöglichen. Sie entsprechen technisch gesehen einer Eigenschaft, die allerdings immer den Namen *this* trägt und der als Parameter ein Index innerhalb eckiger Klammern übergeben wird.

Auf diesen kann dann innerhalb der Methoden *get* und *set* zugegriffen werden, um beispielsweise gezielt auf ein bestimmtes Element eines Arrays zuzugreifen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents an IP address.
 7     /// </summary>
 8     public class IPAddress
 9     {
10         /// <summary>
11         /// Contains the individual parts of an IP address.
12         /// </summary>
13         private int[] _ipPart;
14
15         /// <summary>
16         /// Gets or sets an individual part of the IP
17         /// address.
18         /// </summary>
19         /// <param name="i">The index of the individual
20         /// part.</param>
21         /// <returns>The individual part of the IP
22         /// address.</returns>
23         public int this[int i]
24         {
25             get
26             {
27                 return this._ipPart[i];
28             }
29
30             set
31             {
32                 this._ipPart[i] = value;
33             }
34         }
35
36         /// <summary>
37         /// Initializes a new instance of the IPAddress
38         /// type.
39         /// </summary>
40         public IPAddress()
41         {
```

```
42              this._ipPart = new int[4];
43          }
44
45          /// <summary>
46          /// Initializes a new instance of the IPAddress
47          /// type.
48          /// </summary>
49          /// <param name="ipPart1">The first part of the IP
50          /// address.</param>
51          /// <param name="ipPart2">The second part of the IP
52          /// address.</param>
53          /// <param name="ipPart3">The thired part of the IP
54          /// address.</param>
55          /// <param name="ipPart4">The fourth part of the IP
56          /// address.</param>
57          public IPAddress(int ipPart1, int ipPart2,
58              int ipPart3, int ipPart4)
59              : this()
60          {
61              // Set the individual parts.
62              this._ipPart[0] = ipPart1;
63              this._ipPart[1] = ipPart2;
64              this._ipPart[2] = ipPart3;
65              this._ipPart[3] = ipPart4;
66          }
67      }
68  }
```

Um diesen Indexer nun von außen zu verwenden, muss die Eigenschaft nicht mehr explizit angegeben werden, sondern es genügt, die eckigen Klammern direkt hinter dem Namen des Objekts anzugeben.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Create a new instance of the IP address and
16              // initialize it to 192.168.0.1.
17              IPAddress ipAddress =
18                  new IPAddress(192, 168, 0, 1);
19
20              // Print the first part of the address to the
```

```
21              // console.
22              Console.WriteLine(ipAddress[0]);
23          }
24      }
25  }
```

Kapitel 16
Operatoren

16.1 Was sind Operatoren?

Nachdem Felder und Variablen nun bekannt sind, können Anwendungen Daten speichern und auch wieder abrufen. Allerdings fehlt noch eine Möglichkeit, Daten zu verändern, um beispielsweise Berechnungen ausführen oder Daten miteinander verknüpfen zu können. Die Änderung von Daten wird in C# durch sogenannte Operatoren unterstützt, deren einfachster der bereits bekannte Zuweisungsoperator = ist.

16.2 Arithmetische Operatoren

Den ersten Typ von Operatoren stellen in C# die arithmetischen Operatoren dar, die zum Rechnen mit Daten dienen. Arithmetische Operatoren können mit allen Wertetypen verwendet werden, die Zahlen darstellen, wobei darauf geachtet werden muss, dass die beiden miteinander zu verrechnenden Werte den gleichen Typ aufweisen.

Operator	Funktion	Beispiel
+	Addition	C#

```
1 int x = 2;
2 int y = 3;
3
4 // The sum is 5.
5 int sum = x + y;
```

-	Subtraktion	C#

```
1 int x = 2;
2 int y = 3;
3
4 // The difference is -1.
5 int difference = x - y;
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

*	Multiplikation	C#

```
1 int x = 2;
2 int y = 3;
3
4 // The product is 6.
5 int product = x * y;
```

/	Division	C#

```
1 int x = 2;
2 int y = 3;
3
4 // The quotient is 0.
5 int quotient = x / y;
```

%	Modulo	C#

```
1 int x = 2;
2 int y = 3;
3
4 // The remainder is 2.
5 int remainder = x % y;
```

C# rechnet dabei nach den üblichen mathematischen Regeln, das heißt, es gilt Punkt- vor Strichrechnung. Allerdings kann diese Regelung – wie in der Mathematik auch – durch das Setzen von Klammern geändert werden.

C#

```
1 int x = 2;
2 int y = 3;
3 int z = 5;
4
5 // The result without brackets is 17, with brackets it is 25.
6 int resultA = x + y * z;
7 int resultB = (x + y) * z;
```

Während sich die Operatoren +, - und * so verhalten, wie man es erwarten würde, gibt es bei den beiden Divisionsoperatoren / und % einige Sonderfälle zu beachten. Zunächst ist das Ergebnis einer Verknüpfung von zwei Operanden mit einem arithmetischen Operator wieder vom gleichen Typ wie die beiden Operanden.

Wenn allerdings zwei Operanden von einem ganzzahligen Typ wie beispielsweise *int* oder *long* dividiert werden sollen, ist das Ergebnis unter Umständen nicht ganzzahlig. Deshalb werden in diesem Fall die Nachkommastellen abgeschnitten und nur der ganzzahlige Anteil als Ergebnis zurückgegeben.

C#

```
1 int x = 4;
2 int y = 2;
3
4 // The quotient is 2, since the result is an integer (2).
5 int quotient = x / y;
```

```
 6 x = 3;
 7
 8 // The quotient is 1, since the result is not an integer
 9 // (1,5) and hence the decimal part is cut off.
10 quotient = x / y;
```

Außerdem muss darauf geachtet werden, dass bei der Division nicht durch die Zahl Null geteilt wird, da dies mathematisch nicht definiert ist und zur Laufzeit der Anwendung ein entsprechender Fehler ausgelöst wird.

Die einzige Ausnahme von der Regel, dass arithmetische Operatoren mit jedem Wertetyp verwendet werden können, der Zahlen darstellt, ist der Modulo-Operator %, der nur für ganzzahlige Operanden definiert ist. Der Modulo-Operator gibt den Rest zurück, der entsteht, wenn der eine Operand durch den anderen geteilt wird.

C#

```
 1 int x = 1;
 2 int y = 3;
 3
 4 // The remainder is 1, since 3 is not contained in 1,
 5 // so there is 1 left.
 6 int remainder = x % y;
 7
 8 x = 2;
 9
10 // The remainder is 2, since 3 is not contained in 2,
11 // so there is 2 left.
12 remainder = x % y;
13
14 x = 3;
15
16 // The remainder is 0, since 3 is contained one time in 3,
17 // so there is 0 left.
18 remainder = x % y;
```

Die Modulo-Division entspricht also in gewisser Weise der Art, wie Uhrzeiten berechnet werden. Eine Uhr könnte die Stunden intern nämlich fortlaufend zählen, diese für die Ausgabe allerdings modulo zwölf rechnen.

Häufig kommt es vor, dass eine Variable mit sich selbst verrechnet wird, indem ihr Wert beispielsweise verdoppelt werden soll. Um den Wert einer Variablen i zu verdoppeln, muss dieser mit zwei multipliziert und das Ergebnis anschließend wieder der Variablen i zugewiesen werden.

C#

```
1 i = i * 2;
```

Da Berechnungen dieser Art häufig auftreten, gibt es eine kürzere Schreibweise für diesen Fall, bei dem eine Nennung der Variablen entfallen kann, und der Zuwei-

sungsoperator seinen Platz mit dem arithmetischen Operator tauscht, wobei dieses Verfahren mit jedem arithmetischen Operator funktioniert.

C#

```
1 // Adequate to i = i * 2.
2 i *= 2;
```

Für die besonders häufig auftretenden Fälle, dass eine Variable um eins erhöht oder vermindert werden muss, gibt es sogar eine noch kürzere Schreibweise, indem die Variable mit dem Operator ++ oder -- verknüpft wird. Da dieser Operator keinen zweiten Operanden benötigt, wird er als unärer Operator bezeichnet, wohingegen die anderen arithmetischen Operatoren binäre Operatoren sind.

C#

```
1 // Adequate to i = i + 1.
2 i++;
```

Für die beiden unären Operatoren ++ und -- gibt es allerdings zwei Varianten -- der Operator kann nämlich entweder hinter der Variablen, in der sogenannten Postfix-Notation, oder vor der Variablen, in der sogenannten Präfix-Notation angegeben werden. Der Unterschied liegt darin, ob zuerst der Wert verändert oder zuerst das Ergebnis zurückgegeben wird.

C#

```
1 int i = 3;
2
3 // Prints 3 to the console and increments i afterwards to 4.
4 Console.WriteLine(i++);
5
6 // Prints 5 to the console, since i is incremented before it
7 // gets printed.
8 Console.WriteLine(++i);
```

Schließlich gibt es noch zwei spezielle Fälle, die berücksichtigt werden müssen: Der mathematische Über- beziehungsweise Unterlauf. Ein Überlauf tritt immer dann auf, wenn das Resultat einer Berechnung zu groß für den entsprechenden Typ ist, ein Unterlauf analog dazu, wenn das Resultat zu klein ist.

Sofern diese beiden Fälle nicht gesondert berücksichtigt werden, treten Rechenfehler auf, sobald der größt- oder kleinstmögliche Wert über- oder unterschritten wurden. Die Anwendung wird ansonsten aber weiterhin ausgeführt. Falls eine explizite Überprüfung erforderlich ist, kann diese mit dem Schlüsselwort *checked* für einen abgeschlossenen Codeabschnitt aktiviert werden.

C#

```
1 int x = 2;
2 int y = 3;
```

```
3
4 // Activate checked calculations.
5 checked
6 {
7     Console.WriteLine(x + y);
8 }
```

Im Fall eines Über- oder Unterlaufs tritt wie bei einer Division durch Null ein Fehler auf. So nützlich der Einsatz von *checked* ist, so sollte dennoch berücksichtigt werden, dass diese Prüfung Rechenzeit erfordert und die Anwendung daher an den zu prüfenden Stellen verlangsamt, und dass diese Prüfung einen Spezialfall prüft, der in der Praxis nicht all zu häufig auftritt. Ob *checked* verwendet wird oder nicht, hängt also vom konkreten Bedarf ab.

Sofern eine Anwendung generell als *checked* ausgeführt werden soll, kann dem Compiler dies durch den Parameter /checked mitgeteilt werden. Auf diese Art ist es nicht notwendig, alle Stellen innerhalb des Codes mit dem Schlüsselwort *checked* zu kennzeichnen. Allerdings ist es möglich, einzelne Stellen innerhalb des Codes mit dem Schlüsselwort *unchecked* zu kennzeichnen, um sie von der generellen Prüfung auszuschließen, wobei dieses Schlüsselwort genauso verwendet wird wie *checked*.

C#

```
1 int x = 2;
2 int y = 3;
3
4 // Activate unchecked calculations.
5 unchecked
6 {
7     Console.WriteLine(x + y);
8 }
```

16.3 Relationale Operatoren

Im Gegensatz zu arithmetischen Operatoren dienen die relationalen Operatoren dazu, etwas über das Verhältnis zweier Operanden auszusagen. Mit ihnen kann geprüft werden, ob die beiden Operanden gleich, ungleich, größer, kleiner, größer gleich oder kleiner gleich sind. Als Resultat wird immer ein Wahrheitswert zurückgegeben, der angibt, ob die angegebene Relation wahr oder falsch ist.

Operator	Funktion	Beispiel
==	gleich	C#

```
1 int x = 2;
2 int y = 3;
3
4 // x and y are not equal, hence false.
5 bool result = x == y;
```

`!=`	ungleich	C#
		`1 int x = 2;`
		`2 int y = 3;`
		`3`
		`4 // x and y are not equal, hence true.`
		`5 bool result = x != y;`

`>`	größer	C# `1 int x = 2;` `2 int y = 3;` `3` `4 // x is not greater than y, hence false. ` `5 bool result = x > y;`

`<`	kleiner	C#
		`1 int x = 2;`
		`2 int y = 3;`
		`3`
		`4 // x is smaller than y, hence true.`
		`5 bool result = x < y;`

`>=`	größer oder gleich	C#
		`1 int x = 2;`
		`2 int y = 3;`
		`3`
		`4 // x is not greater than or equal to y, hence`
		`5 // false.`
		`6 bool result = x >= y;`

`<=`	kleiner oder gleich	C#
		`1 int x = 2;`
		`2 int y = 3;`
		`3`
		`4 // x is smaller than or equal to y, hence true.`
		`5 bool result = x <= y;`

Es gilt als guter Stil, die beiden Operanden mitsamt dem relationalen Operator zu klammern, um die Lesbarkeit zu verbessern. An Stelle von

C#

```
1 bool result = foo == bar;
```

würde man also

C#

```
1 bool result = (foo == bar);
```

schreiben.

Relationale Operatoren können prinzipiell zwar auf alle Wertetypen angewandt werden, allerdings ist dies nur begrenzt sinnvoll. Da Dezimalzahlen von Prozes-

soren intern nicht exakt dargestellt werden können, kann man sich nicht darauf verlassen, dass zwei anscheinend gleich große Zahlen des Typs *float*, *double* oder *decimal* bei einem Vergleich mit dem Operator == das Literal *true* als Ergebnis liefern. Dezimalzahlen sollten immer nur mit Hilfe von >, <, >= und <= verglichen werden.

Verweistypen können zumindest mit Hilfe der Operatoren == und != verglichen werden, wobei dies eine andere Semantik als bei Wertetypen hat. Während bei Wertetypen der tatsächliche Wert verglichen wird, wird bei Verweistypen lediglich die Referenz verglichen. Sofern zwei Variablen also eine Referenz auf das identische Objekt enthalten, wird bei einem Vergleich mit == das Literal *true* zurückgeliefert. Enthalten sie aber Referenzen auf zwei verschiedene Objekte, die zwar in ihren Werten, aber nicht in ihrer Objektidentität übereinstimmen, so liefert der Vergleich das Literal *false*.

C#

```
ComplexNumber foo = new ComplexNumber(23, 42);
ComplexNumber bar = foo;

// Returns true, since foo and bar reference the identical
// object.
Console.WriteLine(foo == bar);

y = new ComplexNumber(23, 42);

// Returns false, since foo and bar reference different
// objects, even if they have the same value.
Console.WriteLine(foo == bar);
```

16.4 Logische Operatoren

Während relationale Operatoren einen Vergleich zwischen den beiden Operanden durchführen, verknüpfen logische Operatoren diese. Logische Operatoren können im Gegensatz zu den anderen Operatoren nur auf Operanden des Typs *bool* angewandt werden und liefern auch als Ergebnis einen Wert des Typs *bool*.

Operator	Funktion	Beschreibung	Beispiel

&&	und	Ergibt true, wenn beide Operanden true sind.	C#
			`bool x = true;`
			`bool y = false;`
			
			`// x and y are not both true,`
			`// hence false.`
			`bool result = x && y;`

`||`	oder	Ergibt true, wenn mindestens einer der beiden Operanden true ist.	C#
			`1 bool x = true;`
			`2 bool y = false;`
			`3`
			`4 // At least one of x and y`
			`5 // is true, hence true.`
			`6 bool result = x || y;`

`^`	exklusives oder	Ergibt true, wenn genau einer der beiden Operanden true ist.	C#
			`1 bool x = true;`
			`2 bool y = false;`
			`3`
			`4 // x is true, y is not,`
			`5 // hence true.`
			`6 bool result = x ^ y;`

`!`	nicht	Ergibt true, wenn der Operand false ist, und umgekehrt.	C#
			`1 bool x = true;`
			`2`
			`3 // x is true, hence false.`
			`4 bool result = !x;`

C# verwendet bei der Auswertung logischer Operatoren die sogenannte Kurzschlussevaluierung. Dies bedeutet, dass für die Auswertung eines Operators unter Umständen nicht alle Operanden überprüft werden – ist beispielsweise bei einer und-Verknüpfung bereits der erste Operand *false*, so kann das Ergebnis nicht *true* sein, unabhängig davon, welchen Wert der zweite Operand aufweist. Daher wird dieser nicht mehr überprüft und direkt *false* zurückgegeben.

16.5 Bitweise Operatoren

Bitweise Operatoren ähneln logischen Operatoren sehr stark, allerdings werden sie nicht für Wahrheitswerte, sondern für Ganzzahlen verwendet. Die Überprüfung findet dementsprechend auch nicht auf den Wahrheitswerten der Operanden statt, sondern auf Bitebene der Operanden.

Operator	Funktion	Beschreibung	Beispiel

`&`	und	Ergibt 1, wenn beide Bits 1 sind.	C#
			`1 int x = 23;`
			`2 int y = 42;`
			`3`
			`4 // x is binary 010111 and y`
			`5 // is binary 101010, hence`
			`6 // 000010, which is 2.`
			`7 int result = x {\&} y;`

|	oder	Ergibt 1, wenn mindestens eines der beiden Bits 1 ist.	C#
			`1 int x = 23;`
			`2 int y = 42;`
			`3 // x is binary 010111 and y`
			`4 // is binary 101010, hence`
			`5 // 111111, which is 63.`
			`6 int result = x | y;`

^	exklusives oder	Ergibt 1, wenn genau eines der beiden Bits 1 ist.	C#
			`1 int x = 23;`
			`2 int y = 42;`
			`3`
			`4 // x is binary 010111 and y`
			`5 // is binary 101010, hence`
			`6 // 111101, which is 61.`
			`7 int result = x ^ y;`

~	nicht	Ergibt 1, wenn das Bit 0 ist, und umgekehrt.	C#
			`1 int x = 23;`
			`2`
			`3 // x is binary 010111, hence`
			`4 // 101000, which is -24 due`
			`5 // to internal reasons.`
			`6 int result = ~ x;`

<<	verschieben nach links	Schiebt alle Bits um die angegebene Anzahl nach links.	C#
			`1 int x = 23;`
			`2`
			`3 // x is binary 010111, hence`
			`4 // 101110, which is 46.`
			`5 int result = x << 1;`

>>	verschieben nach rechts	Schiebt alle Bits um die angegebene Anzahl nach rechts.	C#
			`1 int x = 23;`
			`2`
			`3 // x is binary 010111, hence`
			`4 // 001011, which is 11.`
			`5 int result = x >> 1;`

Bitweise Operatoren werden häufig verwendet, um zu überprüfen, ob einzelne Bits gesetzt sind, oder um diese zu setzen beziehungsweise zu löschen. Ebenso wie arithmetische Operatoren können bitweise Operatoren mit dem Zuweisungsoperator zu einer verkürzten Schreibweise zusammengezogen werden.

16.6 Zeichenkettenoperatoren

Zeichenketten erfahren in C# eine Sonderbehandlung. Obwohl sie technisch gesehen Verweistypen sind, verhalten sie sich größtenteils wie Wertetypen, was ihre Handhabung teilweise deutlich erleichtert.

So liefert der Vergleich von zwei Strings mit Hilfe von == und != ein Ergebnis, als wären Strings Wertetypen – enthalten sie den gleichen Text, sind sie gleich. Außerdem können Strings mit Hilfe von <, >, <= und >= alphabetisch miteinander verglichen werden. Ein String gilt dann als kleiner als ein anderer, wenn er im Alphabet vorher einzuordnen ist.

C#

```
string foo = "Hello";
string bar = "World";

// foo and bar do not contain the same text, hence false.
bool result = (foo == bar);

// foo and bar do not contain the same text, hence true.
result = (foo != bar);

// foo is alphabetically prior to bar, hence true.
result = (foo < bar);

// foo is alphabetically not superior to bar, hence false.
result = (foo > bar);

// foo is alphabetically prior or equal to bar, hence true.
result = (foo <= bar);

// foo is alphabetically neither superior nor equal to bar,
// hence false.
result = (foo >= bar);
```

Zudem können Strings mit dem Operator + aneinander gehängt werden, so dass sie einen neuen zusammenhängenden String ergeben. Dieser Vorgang wird auch als Konkatenation bezeichnet.

C#

```
string foo = "Hello ";
string bar = "world!";

// The result is "Hello world!".
string result = foo + bar;
```

Ob ein String leer ist, kann geprüft werden, indem er mit dem leeren String verglichen wird. Alternativ kann auch die Eigenschaft Empty der Klasse String verwendet werden. Eine weitere Möglichkeit ist, die Eigenschaft Length des Strings zu prüfen, ob diese dem Wert Null entspricht.

Da der Vergleich auf die Länge auf Grund der internen Organisation von Strings am schnellsten ausgeführt werden kann, gilt es als guter Stil, diese Variante zu verwenden.

C#

```
string foo = "Hello";

// foo is not empty, hence false.
bool result = (foo == "");
result = (foo == String.Empty);
result = (foo.Length == 0);
```

Ob ein String leer oder eventuell sogar *null* ist, kann mit der statischen Methode IsNullOrEmpty der Klasse string ermittelt werden.

C#

```
string foo = null;

// foo is null, hence true.
bool result = String.IsNullOrEmpty(foo);

foo = "";

// foo is empty, hence true.
result = String.IsNullOrEmpty(foo);

foo = "Hello";

// foo is neither null nor empty, hence false.
result = String.IsNullOrEmpty(foo);
```

16.7 Operatorreihenfolge

Falls mehrere Operatoren gleichzeitig in einer Anweisung verwendet werden, werden diese zunächst von links nach rechts verarbeitet. Allerdings verfügen einige Operatoren über eine höhere Priorität als andere, so dass die Verarbeitung diesen Regeln folgt – ähnlich den Regeln bei den arithmetischen Operatoren.

Die Operatoren haben folgende Priorität, wobei die höchstpriorisierten Operatoren an oberster Stelle stehen:

Operatoren
(), []
++ (postfix), -- (postfix), ++ (präfix), --(präfix), ~, !
*, /, %
+, -
>>, >>>, <<
>, >=;, <, <=

```
==, =!
&
^
|
&&
||
?:
=, <operator>=
```

Mit Hilfe von Operatoren können nun die meisten Methoden der Klasse ComplexNumber implementiert werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Executes when storing begins.
 7     /// </summary>
 8     /// <param name="sender">The sender.</param>
 9     /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
13     /// <summary>
14     /// Executes when storing has finished.
15     /// </summary>
16     /// <param name="sender">The sender.</param>
17     /// <param name="e">The event arguments.</param>
18     public delegate void StoredEventHandler(
19         object sender, EventArgs eventArguments);
20
21     /// <summary>
22     /// Executes when restoring begins.
23     /// </summary>
24     /// <param name="sender">The sender.</param>
25     /// <param name="e">The event arguments.</param>
26     public delegate void RestoringEventHandler(
27         object sender, EventArgs eventArguments);
28
29     /// <summary>
30     /// Executes when restoring has finished.
31     /// </summary>
32     /// <param name="sender">The sender.</param>
33     /// <param name="e">The event arguments.</param>
34     public delegate void RestoredEventHandler(
35         object sender, EventArgs eventArguments);
36
37     /// <summary>
38     /// Represents a complex number.
```

```
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          #endregion
44
45          #region Events
46          #endregion
47
48          #region Methods
49          // ...
50
51          /// <summary>
52          /// Calculates the conjugation.
53          /// </summary>
54          public void Conjugate()
55          {
56              // Calculate the conjugation.
57              this._imaginaryPart *= -1;
58          }
59
60          /// <summary>
61          /// Adds the specified summand to the current complex
62          /// number.
63          /// </summary>
64          /// <param name="summand">The complex number that is
65          /// used as summand.</param>
66          public void Add(ComplexNumber summand)
67          {
68              // Add the summand to the current complex
69              this._realPart += summand.RealPart;
70              this._imaginaryPart += summand.ImaginaryPart;
71          }
72
73          /// <summary>
74          /// Adds the specified summand to the current complex
75          /// number.
76          /// </summary>
77          /// <param name="summand">The real number that is
78          /// used as summand.</param>
79          public void Add(float summand)
80          {
81              // Add the summand to the current complex
82              this._realPart += summand;
83          }
84
85          /// <summary>
86          /// Multiplies the current complex number with the
87          /// specified factor.
88          /// </summary>
89          /// <param name="factor">The complex number that is
90          /// used as factor.</param>
91          public void Multiply(ComplexNumber factor)
92          {
```

```
 93              // Multiply the factor with the current complex
 94              // number.
 95              float? newRealPart =
 96                  (this.RealPart * factor.RealPart) -
 97                  (this.ImaginaryPart * factor.ImaginaryPart);
 98              float? newImaginaryPart =
 99                  (this.RealPart * factor.ImaginaryPart) +
100                  (this.ImaginaryPart * factor.RealPart);
101
102              // Assign the new values to the current complex
103              // number.
104              this.RealPart = newRealPart;
105              this.ImaginaryPart = newImaginaryPart;
106          }
107
108          /// <summary>
109          /// Multiplies the current complex number with the
110          /// specified factor.
111          /// </summary>
112          /// <param name="factor">The real number that is
113          /// used as factor.</param>
114          public void Multiply(float factor)
115          {
116              // Multiply the factor with the current complex
117              // number.
118              this.Multiply(new ComplexNumber(factor));
119          }
120          #endregion
121
122          #region Constructors
123          #endregion
124      }
125  }
```

16.8 Überladen von Operatoren

Zwar ist es mit Hilfe dieser Operatoren nun möglich, Berechnungen mit komplexen Zahlen durchzuführen, allerdings muss für jede einzelne Operation eine eigene Methode aufgerufen werden. So muss beispielsweise für die Addition zweier komplexer Zahlen die entsprechende Methode Add verwendet werden, welche die als Parameter übergebene komplexe Zahl zu der addiert, an der die Methode aufgerufen wird.

C#

```
1 firstComplexNumber.Add(secondComplexNumber);
```

Obwohl dieses Vorgehen funktioniert, entspricht die sich dadurch ergebende Syntax nicht der aus der Mathematik gewohnten Schreibweise, in der zwischen den beiden zu addierenden Zahlen ein + angegeben wird.

Der Grund, warum eine Addition komplexer Zahlen mit Hilfe des Symbols + in C# nicht funktioniert, ist offensichtlich: Die Klasse ComplexNumber ist für .NET eine beliebige, vom Benutzer definierte Klasse, deren mathematische Eigenheiten nur dem Entwickler bekannt sind. In C# ist also schlichtweg nicht definiert, welche Bedeutung dem Symbol + für komplexe Zahlen innewohnt.

Allerdings lassen sich Operatoren – und nichts anderes stellt das Symbol + in C# dar – für benutzerdefinierte Klassen überladen, so dass eigene Datentypen in mathematischen Ausdrücken unter Verwendung der klassischen Syntax miteinander verrechnet werden können.

Um einen Operator zu überladen, genügt es, eine entsprechende Methode innerhalb der Klasse zu definieren, für die der Operator gelten soll. Als Methodenname wird dabei der Operator an sich angegeben, zusätzlich muss ihm allerdings noch das Schlüsselwort *operator* vorangestellt werden. Außerdem muss beachtet werden, dass operatorüberladende Methoden immer klassengebunden, also mit dem Schlüsselwort *static* gekennzeichnet werden müssen.

Als Parameter werden dabei die einzelnen Operanden angegeben, die miteinander verrechnet werden sollen. Die Anzahl der Parameter bestimmt sich dabei aus der Anzahl der Operanden, die für den jeweiligen Operator benötigt werden. Die Operatoren + und * erwarten beispielsweise zwei Operanden, der Operator ! hingegen nur einen.

Der Typ des Rückgabewerts entspricht in jedem Fall der Klasse, in welcher der überladene Operator definiert wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Executes when storing begins.
 7     /// </summary>
 8     /// <param name="sender">The sender.</param>
 9     /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
13     /// <summary>
14     /// Executes when storing has finished.
15     /// </summary>
16     /// <param name="sender">The sender.</param>
17     /// <param name="e">The event arguments.</param>
18     public delegate void StoredEventHandler(
19         object sender, EventArgs eventArguments);
20
```

```
21      /// <summary>
22      /// Executes when restoring begins.
23      /// </summary>
24      /// <param name="sender">The sender.</param>
25      /// <param name="e">The event arguments.</param>
26      public delegate void RestoringEventHandler(
27          object sender, EventArgs eventArguments);
28
29      /// <summary>
30      /// Executes when restoring has finished.
31      /// </summary>
32      /// <param name="sender">The sender.</param>
33      /// <param name="e">The event arguments.</param>
34      public delegate void RestoredEventHandler(
35          object sender, EventArgs eventArguments);
36
37      /// <summary>
38      /// Represents a complex number.
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          #endregion
44
45          #region Events
46          #endregion
47
48          #region Operators
49          /// <summary>
50          /// Adds the specified complex numbers.
51          /// </summary>
52          /// <param name="firstSummand">The complex number
53          /// that is used as first summand.</param>
54          /// <param name="secondSummand">The complex number
55          /// that is used as second summand.</param>
56          /// <returns>The sum of the specified complex
57          /// numbers.</returns>
58          public static ComplexNumber operator +(
59              ComplexNumber firstSummand,
60              ComplexNumber secondSummand)
61          {
62              // Add the two complex numbers.
63              ComplexNumber result =
64                  new ComplexNumber(
65                      firstSummand.RealPart,
66                      firstSummand.ImaginaryPart);
67              result.Add(secondSummand);
68
69              // Return the result to the caller.
70              return result;
71          }
72
73          /// <summary>
74          /// Multiplies the specified complex numbers.
```

```
75          /// </summary>
76          /// <param name="firstFactor">The complex number
77          /// that is used as first factor.</param>
78          /// <param name="secondFactor">The complex number
79          /// that is used as second factor.</param>
80          /// <returns>The product of the specified complex
81          /// numbers.</returns>
82          public static ComplexNumber operator *(
83              ComplexNumber firstFactor,
84              ComplexNumber secondFactor)
85          {
86              // Multiply the two complex numbers.
87              ComplexNumber result =
88                  new ComplexNumber(
89                      firstFactor.RealPart,
90                      firstFactor.ImaginaryPart);
91              result.Multiply(secondFactor);
92
93              // Return the result to the caller.
94              return result;
95          }
96          #endregion
97
98          #region Methods
99          #endregion
100
101         #region Constructors
102         #endregion
103     }
104 }
```

Überladene Operatoren ermöglichen in C# nicht nur, gleichartige Operanden miteinander zu verrechnen, sondern es können auch Operatoren verschiedener Typen angegeben werden. Allerdings muss mindestens einer der Operanden immer der Klasse entsprechen, in welcher der Operator überladen wird. Es ist also beispielsweise nicht möglich, in der Klasse ComplexNumber die Addition für zwei Operanden des Typs *int* zu überladen.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Executes when storing begins.
7      /// </summary>
8      /// <param name="sender">The sender.</param>
9      /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
13     /// <summary>
```

```
14      /// Executes when storing has finished.
15      /// </summary>
16      /// <param name="sender">The sender.</param>
17      /// <param name="e">The event arguments.</param>
18      public delegate void StoredEventHandler(
19          object sender, EventArgs eventArguments);
20
21      /// <summary>
22      /// Executes when restoring begins.
23      /// </summary>
24      /// <param name="sender">The sender.</param>
25      /// <param name="e">The event arguments.</param>
26      public delegate void RestoringEventHandler(
27          object sender, EventArgs eventArguments);
28
29      /// <summary>
30      /// Executes when restoring has finished.
31      /// </summary>
32      /// <param name="sender">The sender.</param>
33      /// <param name="e">The event arguments.</param>
34      public delegate void RestoredEventHandler(
35          object sender, EventArgs eventArguments);
36
37      /// <summary>
38      /// Represents a complex number.
39      /// </summary>
40      public sealed class ComplexNumber : IPersistable
41      {
42          #region Properties
43          #endregion
44
45          #region Events
46          #endregion
47
48          #region Operators
49          /// <summary>
50          /// Adds the specified complex numbers.
51          /// </summary>
52          /// <param name="firstSummand">The complex number
53          /// that is used as first summand.</param>
54          /// <param name="secondSummand">The complex number
55          /// that is used as second summand.</param>
56          /// <returns>The sum of the specified complex
57          /// numbers.</returns>
58          public static ComplexNumber operator +(
59              ComplexNumber firstSummand,
60              ComplexNumber secondSummand)
61          {
62              // Add the two complex numbers.
63              ComplexNumber result =
64                  new ComplexNumber(
65                      firstSummand.RealPart,
66                      firstSummand.ImaginaryPart);
67              result.Add(secondSummand);
```

```
68
69              // Return the result to the caller.
70              return result;
71          }
72
73          /// <summary>
74          /// Adds the specified summand to the specified
75          /// complex number.
76          /// </summary>
77          /// <param name="complexNumber">The complex number
78          /// that is used as first summand.</param>
79          /// <param name="summand">The summand that is used
80          /// as second summand.</param>
81          /// <returns>The sum of the specified complex number
82          /// and the specified summand.</returns>
83          public static ComplexNumber operator +(
84              ComplexNumber complexNumber, float summand)
85          {
86              // Add the two complex numbers.
87              ComplexNumber result =
88                  new ComplexNumber(
89                      complexNumber.RealPart,
90                      complexNumber.ImaginaryPart);
91              result.Add(summand);
92
93              // Return the result to the caller.
94              return result;
95          }
96
97          /// <summary>
98          /// Multiplies the specified complex numbers.
99          /// </summary>
100         /// <param name="firstFactor">The complex number
101         /// that is used as first factor.</param>
102         /// <param name="secondFactor">The complex number
103         /// that is used as second factor.</param>
104         /// <returns>The product of the specified complex
105         /// numbers.</returns>
106         public static ComplexNumber operator *(
107             ComplexNumber firstFactor,
108             ComplexNumber secondFactor)
109         {
110             // Multiply the two complex numbers.
111             ComplexNumber result =
112                 new ComplexNumber(
113                     firstFactor.RealPart,
114                     firstFactor.ImaginaryPart);
115             result.Multiply(secondFactor);
116
117             // Return the result to the caller.
118             return result;
119         }
120
121         /// <summary>
```

```
122         /// Multiplies the specified complex number with
123         /// the specified factor.
124         /// </summary>
125         /// <param name="complexNumber">The complex number
126         /// that is used as first factor.</param>
127         /// <param name="factor">The factor that is used
128         /// as second factor.</param>
129         /// <returns>The product of the specified complex
130         /// number and the specified factor.</returns>
131         public static ComplexNumber operator *(
132             ComplexNumber complexNumber, float factor)
133         {
134             // Multiply the two complex numbers.
135             ComplexNumber result =
136                 new ComplexNumber(
137                     complexNumber.RealPart,
138                     complexNumber.ImaginaryPart);
139             result.Multiply(factor);
140
141             // Return the result to the caller.
142             return result;
143         }
144         #endregion
145
146         #region Methods
147         #endregion
148
149         #region Constructors
150         #endregion
151     }
152 }
```

Bei der Überladung von Operatoren gibt es drei Einschränkungen, die beachtet werden müssen: Zum einen können in C# einige Operatoren nicht überladen werden, dazu zählen insbesondere der Zuweisungsoperator, sämtliche Klammern und auch alle Operatoren, die nicht durch ein Symbol wie + oder *, sondern durch ein Schlüsselwort repräsentiert werden.

Zum zweiten können einige Operatoren nur paarweise überladen werden, was insbesondere für die relationalen Operatoren gilt. Das heißt, wird beispielsweise der Operator > überladen, so muss auch der entsprechende Operator < überladen werden.

Zu guter letzt ist es nicht möglich, die verkürzte Schreibweise, die einen Operator mit dem Zuweisungsoperator verbindet, getrennt von dem eigentlichen Operator zu überladen. Wird also zum Beispiel der Operator + überladen, so wird dadurch implizit auch der Operator += überladen.

Kapitel 17
Ausdrücke

17.1 Konvertieren

Bei der Division und Modulodivision von Typen wurde erwähnt, dass das Ergebnis einer Verknüpfung von zwei Operanden mit Hilfe eines arithmetischen Operators immer dem Typ der beiden Operanden entspricht, weshalb es insbesondere bei der Division von ganzzahligen Datentypen zu Problemen kommen kann, da der Dezimalteil verloren geht.

Die vermeintliche Lösung, das Ergebnis einer solchen Division einer Variablen vom Typ *float* oder *double* zuzuweisen, erweist sich bei näherer Betrachtung als unzureichend, da die Dezimalstellen des Ergebnisses bereits im Speicher abgeschnitten werden, noch bevor die Zuweisung an die aufnehmende Variable ausgeführt wird.

Eine Möglichkeit, diesem Problem zu begegnen, liegt darin, mindestens einen der Operanden in einen Typ zu wandeln, der über Dezimalstellen verfügt. Da es aber nicht in jedem Fall möglich ist, einen Operanden von vornherein als entsprechenden Typ zu deklarieren, muss dies gegebenenfalls während der Ausführung zur Laufzeit geschehen. Dieser Vorgang wird als konvertieren oder casten bezeichnet.

Die einfachste Möglichkeit, einen Typ in einen anderen zu konvertieren, ist, den eigentlichen Wert dem neuen Typen zuzuweisen. Da hierbei die Umwandlung in den neuen Typ implizit geschieht, wird diese Art der Konvertierung als implizite Konvertierung bezeichnet.

C#

```
1 // Assign a value to an int variable.
2 int x = 23;
3
4 // Assign the value to a long variable. The value is
5 // implicitly casted from int to long.
6 long y = x;
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Sofern der Wertebereich des Typs, in den konvertiert wird, umfangreicher ist als der des Typs, der den ursprünglichen Wert enthält, funktioniert dieses Verfahren ohne weiteres. Auf diese Art können beispielsweise *int* in *long* und *float* in *double* konvertiert werden. Beim Versuch, eine Konvertierung in umgekehrter Richtung durchzuführen, meldet C# allerdings einen Fehler, da potenziell ein Werteverlust eintreten könnte.

Soll eine solche Konvertierung dennoch ausgeführt werden, muss dies mit Hilfe einer expliziten Konvertierung geschehen. Bei dieser wird dem umzuwandelnden Wert der Typ, in den konvertiert wird, innerhalb runder Klammern vorangestellt.

C#

```
1 // Assign a value to a long variable.
2 long x = 23;
3
4 // Assign the value to an int variable. The value needs to
5 // be casted explicitly.
6 int y = (long)x;
```

Die explizite Konvertierung ermöglicht auch die Division zweier Ganzzahlen unter Beibehaltung des Dezimalteils des Ergebnisses. Dazu muss lediglich einer der beiden Operanden in einen Dezimaltyp konvertiert werden.

C#

```
1 int x = 23;
2 int y = 42;
3
4 // The quotient is 0, since no cast has been done.
5 float quotient = x / y;
6
7 // The quotient is 0.547619, since one of the operands has
8 // been casted explicitly to a decimal type.
9 quotient = (float)x / y;
```

17.2 Boxing

Im Rahmen der Vererbung wurde erwähnt, dass alle Typen von *object* ableiten. Aus diesem Grund ist es möglich, jeden beliebigen Typ nach *object* zu konvertieren, sogar dann, wenn es sich bei dem ursprünglichen Typ um einen Werte- und nicht um einen Verweistyp handelt.

Während sich bei einem Verweistyp lediglich der Typ des Verweises ändert, ändert sich bei einem Wertetyp zusätzlich die Art, wie der Wert gespeichert wird, da die Anwendung bei einem Verweistyp nur mit einem Verweis auf die eigentlichen Daten, bei einem Wertetyp aber direkt mit den eigentlichen Daten arbeitet. Daher ist es notwendig, einen Wertetyp, der nach *objekt* konvertiert werden soll, zunächst

in einen zusätzlichen Verweistyp zu verpacken, auf den dann wiederum ein Verweis vom Typ *object* angelegt werden kann.

Dieses Verpacken wird als Boxing bezeichnet und von C# intern automatisch durchgeführt, sobald ein Wertetyp in einen Verweistyp konvertiert wird. Obwohl man sich also nicht händisch um das Boxing kümmern muss, sollte man sich während der Entwicklung dieses Vorgangs im Hintergrund immer bewusst sein, da dieser nicht nur zusätzlichen Speicher verbraucht, sondern auch Zeit benötigt. Insofern sollte Boxing nur mit Bedacht und gezielt an einigen Stellen eingesetzt werden.

Nachdem ein Wertetyp in einen Verweistyp verpackt wurde, kann dieser Vorgang auch wieder umgekehrt werden, um aus dem Verweistyp den ursprünglichen Wertetyp zu erhalten. Dies wird als Unboxing bezeichnet und folgt den gleichen Regeln wie das Boxing.

C#

```
1 int valueType = 23;
2
3 // Box the value type and create a reference type.
4 object referenceType = valueType;
5
6 // Unbox the reference type.
7 valueType = (int)referenceType;
```

17.3 Benutzerdefiniertes Konvertieren

Prinzipiell können mit diesen Möglichkeiten zum einen beliebige Typen in *object* konvertiert werden, zum anderen können Typen ineinander konvertiert werden, die über eine gemeinsame Basis verfügen oder die in einer Vererbungshierarchie zueinander stehen. Gelegentlich kann es jedoch nützlich sein, eine eigene Konvertierung definieren zu können, um beispielsweise eine komplexe Zahl mit Hilfe ihres Absolutbetrags in float? zu konvertieren.

Diese Konvertierung kann so wohl implizit wie auch explizit implementiert werden. In beiden Fällen muss die bestehende Klasse durch eine weitere Operatorüberladung ergänzt werden, wobei der Zieltyp der Konvertierung als Operatorname dient. Außerdem muss eines der beiden Schlüsselwörter *implicit* und *explicit* angegeben werden, um zu definieren, ob die Konvertierung in den Zieltyp implizit ausgeführt werden kann, oder ob zwingend eine explizite Konvertierung benötigt wird.

C#

```
1 using System;
2
3 namespace GoloRoden.GuideToCSharp
4 {
5     /// <summary>
```

```
 6     /// Executes when storing begins.
 7     /// </summary>
 8     /// <param name="sender">The sender.</param>
 9     /// <param name="e">The event arguments.</param>
10     public delegate void StoringEventHandler(
11         object sender, EventArgs eventArguments);
12
13     /// <summary>
14     /// Executes when storing has finished.
15     /// </summary>
16     /// <param name="sender">The sender.</param>
17     /// <param name="e">The event arguments.</param>
18     public delegate void StoredEventHandler(
19         object sender, EventArgs eventArguments);
20
21     /// <summary>
22     /// Executes when restoring begins.
23     /// </summary>
24     /// <param name="sender">The sender.</param>
25     /// <param name="e">The event arguments.</param>
26     public delegate void RestoringEventHandler(
27         object sender, EventArgs eventArguments);
28
29     /// <summary>
30     /// Executes when restoring has finished.
31     /// </summary>
32     /// <param name="sender">The sender.</param>
33     /// <param name="e">The event arguments.</param>
34     public delegate void RestoredEventHandler(
35         object sender, EventArgs eventArguments);
36
37     /// <summary>
38     /// Represents a complex number.
39     /// </summary>
40     public sealed class ComplexNumber : IPersistable
41     {
42         #region Properties
43         #endregion
44
45         #region Events
46         #endregion
47
48         #region Operators
49         // ...
50
51         /// <summary>
52         /// Casts the specified complex number to float?.
53         /// </summary>
54         /// <param name="complexNumber">The complex number
55         /// that shall be casted.</param>
56         /// <returns>A float? representation of the specified
57         /// complex number.</returns>
58         public static implicit operator float?(
59             ComplexNumber complexNumber)
```

```
60          {
61              // Return the complex number as float? by using
62              // its absolute value.
63              return complexNumber.AbsoluteValue;
64          }
65          #endregion
66
67          #region Methods
68          #endregion
69
70          #region Constructors
71          #endregion
72      }
73  }
```

Ein Rückgabetyp muss im Gegensatz zu den bisherigen Operatorüberladungen nicht angegeben werden, da sich dieser aus dem Operator an sich bereits ergibt. Zu beachten ist bei der Definition benutzerdefinierter Konvertierungsoperatoren noch, dass es nicht für einen Zieltyp zugleich so wohl einen impliziten wie auch einen expliziten Operator geben kann. Die Definition mehrerer Konvertierungsoperatoren ist nur möglich, sofern sich diese durch ihren Zieltyp unterscheiden.

17.4 Konvertierbarkeit

Obwohl C# zahlreiche Möglichkeiten bietet, zwischen verschiedenen Typen zu konvertieren, kann es dennoch vorkommen, dass ein bestimmter Typ schlichtweg nicht in einen anderen Typ konvertierbar ist. Wird ein solcher Versuch trotzdem unternommen, tritt ein Fehler auf und die Ausführung der Anwendung wird abgebrochen.

Um dies zu verhindern, enthält C# drei Schlüsselwörter, mit denen geprüft werden kann, ob sich ein Typ in einen bestimmten Zieltyp konvertiert lässt. Das einfachste dieser Schlüsselwörter ist *typeof*, das ein Objekt der Klasse Type zurückgibt, das Informationen zu dem jeweiligen Typ enthält. Eine Analyse dieses Typobjekts ermöglicht dann im weiteren Verlauf, zu bestimmen, ob und auf welche Art konvertiert werden kann.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
```

```
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Get type information on the Program type.
16              Type type = typeof(Program);
17
18              // Print the type's full name to the console.
19              Console.WriteLine(type.FullName);
20          }
21      }
22  }
```

Häufig ist der Einsatz eines kompletten Typobjekts allerdings zu aufwändig, da nur von Interesse ist, ob ein Typ überhaupt in einen bestimmten Zieltyp konvertiert werden kann. Dazu dient das Schlüsselwort *is*, das je nach Konvertierbarkeit *true* oder *false* an den Aufrufer zurückgibt.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Define a string and store it within an object
16              // reference.
17              object value = "Hello world!";
18
19              // Execute code depending on the type of the
20              // value.
21              if (value is string)
22              {
23                  // Cast the value to a string.
24                  string valueAsString = (string)value;
25
26                  // TODO gr: Do something ...
27                  //          2008-01-03
28              }
29          }
30      }
31  }
```

Schließlich gibt es noch das Schlüsselwort *as*, das prinzipiell ebenfalls eine explizite Konvertierung durchführt, im Fehlerfall aber nicht die Ausführung der Anwendung abbricht, sondern das Literal *null* zurückgibt.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a string and store it within an object
16             // reference.
17             object value = "Hello world!";
18
19             // Try to cast the value to string.
20             string valueAsString = value as string;
21
22             // If the cast was successful, execute some
23             // code.
24             if (valueAsString != null)
25             {
26                 // TODO gr: Do something ...
27                 //          2008-01-03
28             }
29         }
30     }
31 }
```

Kapitel 18
Anweisungen

18.1 Bedingungen

Allen Beispielen in den vergangenen Kapiteln ist gemein, dass sie noch keine einzige Zeile Code enthalten, der im klassischen Sinn ausgeführt werden kann. Sämtliche Konzepte, die bislang thematisiert wurden, dienen lediglich der Modellierung und Strukturierung von Daten und Anwendungen. Sie bilden also nur den äußeren Rahmen für eine Anwendung, deren Inneres aber noch mit konkretem Code gefüllt und damit zum Leben erweckt werden muss.

Zur Steuerung des Ablaufs einer Anwendung gibt es in C# zwei wesentliche Konzepte: Bedingungen und Schleifen. Während Codeabschnitte mit Hilfe von Bedingungen nur unter bestimmten Umständen ausgeführt werden und die Ausführung dadurch dynamisch an den äußeren Kontext angepasst werden kann, dienen Schleifen der wiederholten Ausführung von Code, um beispielsweise eine Menge gleichartiger Daten zu verarbeiten.

Die einfachste Anweisung zur Abfrage einer Bedingung wurde im Kapitel zu Ereignissen bereits erwähnt, da dort überprüft werden musste, ob an den Delegaten eines Ereignisses überhaupt Methoden angehängt worden sind.

Der Anweisung *if* wird dabei in runden Klammern ein Ausdruck übergeben, der von C# ausgewertet und entweder zu *true* oder *false* evaluiert wird. Sofern der Ausdruck *true* ergibt, wird der Rumpf von *if* ausgeführt, andernfalls nicht.

In dem folgenden Beispiel wird also überprüft, ob der Storing-Delegat ungleich dem Literal *null* ist. Wenn dem so ist, wird der Rumpf ausgeführt und das entsprechende Ereignis ausgelöst.

C#

```
1 // Check if there are any event handlers.
2 if(this.Storing != null)
3 {
4     // Raise the storing event.
5     this.Storing(this, null);
6 }
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Die geschweiften Klammern um den Rumpf sind optional, solange der Rumpf nur aus einer einzelnen Zeile besteht, allerdings ist es guter Stil, die geschweiften Klammern in jedem Fall zu verwenden.

Die einfache *if*-Anweisung ermöglicht zwar bereits die bedingte Ausführung von Code, allerdings erfordert eine exklusive Ausführung zweier Codeabschnitte zwei Abfragen, die einander außer in der Prüfung auf *true* oder *false* gleichen.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Define a condition.
            bool condition = 23 < 42;

            // Check whether the condition evaluates to
            // true. If so, run the specified code.
            if (condition == true)
            {
                // TODO gr: Do something ...
                //          2008-01-03
            }

            // Check whether the condition evaluates to
            // false. If so, run the specified code.
            if (condition == false)
            {
                // TODO gr: Do something else ...
                //          2008-01-03
            }
        }
    }
}
```

Daher gibt es das Schlüsselwort *else*, das einen weiteren Rumpf einleitet, der ausgeführt wird, wenn die bei *if* genannte Bedingung eben nicht zu *true* evaluiert wird. Die erneute Angabe einer weiteren Bedingung kann somit also entfallen.

C#

```
using System;

```

```
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             bool condition = 23 < 42;
17 
18             // Check whether the condition evaluates to
19             // true. If so, run the specified code. If not,
20             // run the second block.
21             if (condition == true)
22             {
23                 // TODO gr: Do something ...
24                 //          2008-01-03
25             }
26             else
27             {
28                 // TODO gr: Do something else ...
29                 //          2008-01-03
30             }
31         }
32     }
33 }
```

Ein wichtiger Aspekt bei der Überprüfung der Bedingung ist, dass die explizite Angabe des Vergleichs mit *true* oder *false* entfallen kann, da eine logische Bedingung automatisch zu einem der beiden Literale evaluiert wird. Ein Vergleich auf *false* kann dabei mit Hilfe des Operators ! durchgeführt werden.

C#

```
 1 using System;
 2 
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
```

```
15              // Define a condition.
16              bool condition = 23 < 42;
17
18              // Check whether the condition evaluates to
19              // true.
20              if (condition)
21              {
22                  // TODO gr: Do something ...
23                  //          2008-01-03
24              }
25
26              // Check whether the condition evaluates to
27              // false.
28              if (!condition)
29              {
30                  // TODO gr: Do something ...
31                  //          2008-01-03
32              }
33          }
34      }
35  }
```

Unter Umständen kann es notwendig sein, mehr als zwei Optionen zu prüfen. Dazu kann auf mehrere *if*-Anweisungen zurückgegriffen werden, die ineinander verschachtelt werden.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Define some conditions.
16              bool condition1 = 23 < 42;
17              bool condition2 = 17 < 23;
18
19              // Check whether the first condition evaluates
20              // to true.
21              if (condition1)
22              {
23                  // TODO gr: Do something ...
24                  //          2008-01-03
25              }
26              else
```

```
27              {
28                  // Check whether the second condition
29                  // evaluates to true.
30                  if (condition2)
31                  {
32                      // TODO gr: Do something else ...
33                      //          2008-01-03
34                  }
35                  else
36                  {
37                      // TODO gr: Do something completely
38                      //          else ...
39                      //          2008-01-03
40                  }
41              }
42          }
43      }
44  }
```

Zunächst wird also geprüft, ob die erste Bedingung zutrifft, wenn nein, wird in den entsprechenden *else*-Block verzweigt, in dem wiederum die zweite Bedingung geprüft wird, und so weiter. Der innerste *else*-Block wird dabei nur ausgeführt, wenn alle vorangegangenen Bedingungen fehlgeschlagen sind.

Obwohl dieses Vorgehen zum gewünschten Ziel führt, wird die Darstellung bei einer zunehmenden Anzahl von Ebenen unübersichtlich. Daher gibt es die Möglichkeit, weitere Abfragen mit dem Konstrukt *else if* auf der gleichen Ebene wie das erste *if* zu positionieren. Die Angabe des abschließenden *else* ohne Bedingung ist dabei wiederum optional.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define some conditions.
16             bool condition1 = 23 < 42;
17             bool condition2 = 17 < 23;
18
19             if (condition1)
20             {
21                 // TODO gr: Do something ...
```

```
22                  //          2008-01-03
23              }
24              else if (condition2)
25              {
26                  // TODO gr: Do something else ...
27                  //          2008-01-03
28              }
29              else
30              {
31                  // TODO gr: Do something completely else ...
32                  //          2008-01-03
33              }
34          }
35      }
36  }
```

Werden innerhalb einer Bedingung mehrere Bedingungen angegeben und mit Hilfe von logischen Operatoren wie && oder || verknüpft, werden diese in C# von links nach rechts ausgewertet. Zu beachten ist hierbei, dass C# die Auswertung abbricht, sobald das endgültige Ergebnis des Gesamtausdrucks feststeht. Diese Technik wird als Kurzschlussevaluierung bezeichnet.

Werden beispielsweise zwei Bedingungen mit Hilfe von && verknüpft und ergibt bereits die erste Bedingung *false*, so wird die zweite Bedingung nicht mehr ausgewertet, da der Gesamtausdruck unabhängig von deren Ergebnis in jedem Fall nur noch zu *false* evaluiert werden kann.

Da es häufig Abfragen der Art gibt, dass einer Variablen entweder ein oder ein anderer Wert zugewiesen werden soll, gibt es dafür in C# zwei abkürzende Schreibweisen. Handelt es sich bei der entsprechenden Variablen um eine Variable des Typs *bool*, so ist es kürzer, an Stelle von

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents the application class.
7      /// </summary>
8      public class Program
9      {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             bool condition = 23 < 42;
17
18             // Set result to true if the condition evaluates
19             // to true, otherwise set the result to false.
```

```
20              bool result;
21              if (condition)
22              {
23                  result = true;
24              }
25              else
26              {
27                  result = false;
28              }
29          }
30      }
31  }
```

den verkürzten Ausdruck

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             bool condition = 23 < 42;
17
18             // Set the result to true if the condition
19             // evaluates to true, otherwise set the result
20             // to false.
21             bool result = condition;
22         }
23     }
24 }
```

zu verwenden. Ebenso kann bei Variablen jedes beliebigen anderen Typs der einzige Operator mit drei Operanden verwendet werden, der sogenannte triäre Operator. An Stelle von

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
```

```
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             bool condition = 23 < 42;
17
18             // Set the result to 23 if the condition
19             // evaluates to true, otherwise set the result
20             // to 42.
21             int result;
22             if (condition)
23             {
24                 result = 23;
25             }
26             else
27             {
28                 result = 42;
29             }
30         }
31     }
32 }
```

lässt sich unter Zuhilfenahme des triären Operators

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             bool condition = 23 < 42;
17
18             // Set the result to 23 if the condition
19             // evaluates to true, otherwise set the result
20             // to 42.
21             int result = condition ? 23 : 42;
22         }
```

```
23      }
24  }
```

schreiben. Des weiteren besteht im Zusammenhang mit nullbaren Wertetypen häufig der Wunsch, einen Standardwert zuzuweisen, falls der nullbare Wertetyp dem Literal *null* entspricht. An Stelle der umfangreichen Abfrage

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Define a nullable type.
16              int? nullableType = null;
17
18              // Set the value type to 23 if the nullable type
19              // is null, otherwise set it to the value of the
20              // nullable type.
21              int valueType;
22              if (nullableType == null)
23              {
24                  valueType = 23;
25              }
26              else
27              {
28                  valueType = (int)nullableType;
29              }
30          }
31      }
32  }
```

kann in C# seit der Version 2.0 der Operator ?? verwendet werden, so dass statt dessen

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
```

```
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a nullable type.
16             int? nullableType = null;
17
18             // Set the value type to the value of the
19             // nullable type if it is not equal to null,
20             // otherwise set the value type to 23.
21             int valueType = nullableType ?? 23;
22         }
23     }
24 }
```

geschrieben werden kann. Schließlich gibt es neben *if* noch die Anweisung *switch* zur bedingten Ausführung von Code, die sich insbesondere dann anbietet, wenn für jede Ausführungsalternative der gleiche Ausdruck ausgewertet werden soll und die Ausführung nur vom jeweiligen Ergebnis abhängt.

Die *switch*-Anweisung erwartet die Bedingung ebenfalls innerhalb von runden Klammern, die einzelnen Fälle werden aber über entsprechende *case*-Zweige abgedeckt. Ebenso wie bei *if* gibt es auch bei *switch* einen optionalen Ausführungspfad ohne Bedingung, der ausgeführt wird, falls jeder vorige Option fehlschlägt, und der mit Hilfe des Schlüsselwortes *default* eingeleitet wird.

Alle Blöcke müssen bei *switch* mit dem Schlüsselwort *break* abgeschlossen werden.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Set the condition.
16             int condition = 23;
17
18             // Execute code depending on the condition.
19             switch (condition)
```

```
20              {
21                  case 23:
22                      // TODO gr: Do something ...
23                      //          2008-01-03
24                      break;
25                  case 42:
26                      // TODO gr: Do something else ...
27                      //          2008-01-03
28                      break;
29                  default:
30                      // TODO gr: Do something completely
31                      //          else ...
32                      //          2008-01-03
33                      break;
34              }
35          }
36      }
37  }
```

Die einzige Ausnahme davon ist das sogenannte Durchfallen von einer Alternative zu der darauffolgenden, was genutzt werden kann, falls beide Alternativen den gleichen Ausführungsblock verwenden sollen. Sobald der durchfallende Block allerdings eine einzige Zeile Code enthält, wird von C# ein entsprechender Fehler bei der Übersetzung ausgelöst.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Define a condition.
16             int condition = 23;
17
18             // Execute code depending on the condition.
19             switch (condition)
20             {
21                 case 17:
22                 case 23:
23                     // TODO gr: Do something ...
24                     //          2008-01-03
25                     break;
26                 case 42:
```

```
27                      // TODO gr: Do something else ...
28                      //          2008-01-03
29                      break;
30                  default:
31                      // TODO gr: Do something completely
32                      //          else ...
33                      //          2008-01-03
34                      break;
35              }
36          }
37      }
38  }
```

Prinzipiell kann in C# auch gezielt von einem Ausführungsblock in einen anderen gesprungen werden, um beispielsweise trotz enthaltenem Code in einen weiteren Ausführungsblock durchzufallen. Dies geschieht mittels des Schlüsselwortes *goto*. Da dies in der Praxis aber als schlechter Stil angesehen wird, wird an dieser Stelle nicht näher darauf eingegangen.

18.2 Schleifen

Während durch eine Bedingung wie *if* oder *switch* definiert werden kann, welche Anweisungen unter welchen Umständen ausgeführt werden, können Anweisungen mit Hilfe von Schleifen wiederholt ausgeführt werden, wobei die Anzahl der Wiederholungen entweder vorher festgelegt wird oder sich dynamisch während der Ausführung ergibt.

Die einfachste Schleife in C# ist eine reine Zählschleife, welche die in ihrem Rumpf enthaltenen Anweisungen in einer vorherbestimmten Anzahl an Durchläufen ausführt. Diese Schleife wird mittels des Schlüsselworts *for* implementiert. Innerhalb runder Klammern werden mit Hilfe dreier Ausdrücke der Initialisierungsausdruck, das Abbruchkriterium und der Aktualisierungsausdruck angegeben.

Um beispielsweise eine Anweisung n Mal auszuführen, wird zu Beginn der Schleife eine Variable mit dem Wert 0 initialisiert und anschließend in jedem Durchlauf um eins erhöht, bis die Schleife n Mal durchlaufen wurde. Diese Variable wird auch als Schleifenvariable oder Schleifeninvariante bezeichnet und enthält in jedem Durchlauf den Wert des jeweils aktuellen Durchlaufs.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents the application class.
7      /// </summary>
8      public class Program
```

```
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Set the upper limit for the loop.
16             int n = 23;
17
18             // Initialize the invariant with 0 and execute
19             // the loop as long as i is less than n.
20             for (int i = 0; i < n; i++)
21             {
22                 // Print the square numbers to the console.
23                 Console.WriteLine("The square number of " +
24                     i + " is " + i * i + ".");
25             }
26         }
27     }
28 }
```

Es hat sich in der Praxis eingebürgert, die Invariante mit i zu bezeichnen, obwohl dies den Namenskonventionen für lokale Variablen widerspricht. Sofern Schleifen verschachtelt werden, werden für die Invarianten der inneren Schleifen fortlaufend die Buchstaben ab j verwendet.

Des weiteren ist es guter Stil, die Invariante einer Schleife mit dem Wert 0 und nicht mit 1 zu initialisieren, wobei abhängig vom Kontext auch vollständig andere Startwerte sinnvoll sein können. Ebenso wird die Invariante in den meisten Fällen bei jedem Durchlauf um eins erhöht, jedoch kann auch dies je nach Bedarf beliebig gewählt werden. Unter Umständen sind auch Schleifen denkbar, deren Initialisierungsausdruck die Invariante zunächst auf einen hohen Wert setzt, der dann in jedem Schleifendurchlauf verringert wird – kurz, der Fantasie sind dabei keine Grenzen gesetzt.

Da das Abbruchkriterium vor jedem neuen Durchlauf überprüft wird, kann es vorkommen, dass eine *for*-Schleife überhaupt nicht ausgeführt wird. Dann nämlich, wenn der Initialisierungsausdruck die Invariante auf einen Wert setzt, für den das Abbruchkriterium zu **false** evaluiert wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
```

```
12          /// </summary>
13          public static void Main()
14          {
15              for (int i = 0; i > 1; i++)
16              {
17                  // This loop is never executed, since i is
18                  // initialized with 0, so i > 1 evaluates
19                  // to false.
20              }
21          }
22      }
23  }
```

Der wesentliche Nachteil der *for*-Schleife ist, dass von vornherein feststehen muss, wie viele Durchläufe ausgeführt werden sollen. Falls dies nicht bekannt ist, sondern nur ein Abbruchkriterium feststeht, kann in C# die *while*-Schleife eingesetzt werden. Ihr Prinzip entspricht dem der *for*-Schleife, wobei sich die Angabe innerhalb der runden Klammern auf das Abbruchkriterium beschränken.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Set the upper limit.
16              int upperLimit = 23;
17
18              // Iterate over all square numbers from 0 to the
19              // upper limit.
20              int i = 0;
21              while (i < upperLimit)
22              {
23                  // Print the square number to the console.
24                  Console.WriteLine("The square number of " +
25                      i + " is " + i * i + ".");
26
27                  // Increase i by 1.
28                  i++;
29              }
30          }
31      }
32  }
```

Falls sich in einem Durchlauf nichts an der Gültigkeit des Abbruchkriteriums ändert, wird die Schleife ein weiteres Mal durchlaufen. Um keine Endlosschleife zu erhalten, ist es allerdings wichtig, darauf zu achten, dass sich das Abbruchkriterium zumindest überhaupt ändern könnte.

Wie bei der *for*-Schleife wird auch die *while*-Schleife unter Umständen kein einziges Mal durchlaufen, falls nämlich das Abbruchkriterium von vornherein zu *false* evaluiert wird. Daher werden diese beiden Schleifen auch als abweisende Schleifen bezeichnet.

Sofern eine Schleife in jedem Fall mindestens ein Mal durchlaufen werden soll, gibt es in C# auch eine nichtabweisende Variante der *while*-Schleife, die sich des Schlüsselworts *do* bedient.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Set the upper limit.
            int upperLimit = 23;

            // Iterate over all square numbers from 0 to the
            // upper limit.
            int i = 0;
            do
            {
                // Print the square number to the console.
                Console.WriteLine("The square number of " +
                    i + " is " + i * i + ".");

                // Increase i by 1.
                i++;
            }
            while (i < upperLimit);
        }
    }
}
```

Abgesehen davon, dass die *do*-Schleife das Abbruchkriterium erst nach und nicht vor dem Durchlauf überprüft, ist sie in ihrer sonstigen Arbeitsweise identisch mit der *while*-Schleife.

Beachtenswert bei diesen beiden Schleifen ist, dass der öffnenden Anweisung nie ein Semikolon folgt, das schließende *while* bei der *do*-Schleife allerdings durch ein Semikolon abgeschlossen wird.

18.3 Sprunganweisungen

In manchen Fällen kann es je nach Kontext notwendig sein, die Ausführung einer Schleife mit sofortiger Wirkung abzubrechen. Dies ist in C# mit Hilfe des Schlüsselworts *break* möglich, das bereits bei den einzelnen Zweigen einer *switch*-Anweisung Verwendung fand.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Set the upper limit.
            int upperLimit = 23;

            // Iterate over all square numbers from 0 to the
            // upper limit.
            for (int i = 0; i < upperLimit; i++)
            {
                // Print the square number to the console.
                Console.WriteLine("The square number of " +
                    i + " is " + i * i + ".");

                // Check whether the loop shall still be
                // continued. If not, break.
                if (i > 17)
                {
                    break;
                }
            }
        }
    }
}
```

break bricht die Ausführung der aktuellen Schleife ab, indem es diese verlässt und in die umgebende Struktur springt. Sofern dies beispielsweise bei geschachtelten Schleifen wiederum eine Schleife ist, wird diese allerdings nach wie vor ausgeführt, da *break* nur eine einzelne Ebene verlässt.

Während *break* einen Schleifenablauf vollständig abbricht, kann es unter Umständen nur gewünscht sein, den aktuellen Durchlauf abzubrechen, prinzipiell aber innerhalb der Schleifenausführung zu bleiben, das heißt, direkt mit dem nächsten Durchlauf fortzufahren. Dies geschieht in C# mit Hilfe des Schlüsselwortes *continue*.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Set the upper limit.
16             int upperLimit = 23;
17
18             // Iterate over all square numbers from 0 to the
19             // upper limit.
20             for (int i = 0; i < upperLimit; i++)
21             {
22                 // Check whether the current number is odd.
23                 // If so, skip the current iteration.
24                 if (i {\%} 2 != 0)
25                 {
26                     continue;
27                 }
28
29                 // Print the square number to the console.
30                 Console.WriteLine("The square number of " +
31                     i + " is " + i * i + ".");
32             }
33         }
34     }
35 }
```

18.4 foreach

Zu guter letzt gibt es in C# noch eine weitere Schleife, die allerdings über kein entsprechendes Pendant in MSIL verfügt, sondern die lediglich als komfortable Lösung in C# enthalten ist, vom Compiler während der Übersetzung aber in eine klassische *while*-Schleife umgewandelt wird.

Diese Schleife dient dazu, alle Elemente einer Aufzählung zu durchlaufen, ohne den Aufwand, zunächst die Länge dieser Auflistung ermitteln und eine Schleifeninvariante erzeugen zu müssen. Implementiert wird diese Schleife mit Hilfe des Schlüsselwortes *foreach*.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Define an array of colors.
            string[] colors =
                new string[] { "Red", "Green", "Blue" };

            // Iterate over all colors.
            foreach (string color in colors)
            {
                // Print the current color to the console.
                Console.WriteLine(color);
            }
        }
    }
}
```

Intern wandelt der Compiler diese Schleife in eine *while*-Schleife um, die mit sämtlichen Aufzählungen arbeiten kann, welche die Schnittstelle IEnumerator implementieren. Dies sind nicht nur sämtliche Arrays, sondern auch etliche der in den Namensräumen System.Collections und System.Collections.Generic enthaltenen Typen.

Im Gegensatz zu den anderen Schleifen, bei denen die Invariante als Index für eine Aufzählung dienen kann, liefert die Invariante der *foreach*-Schleife direkt ein Element der Aufzählung. Dabei wird allerdings nicht garantiert, in welcher Reihenfolge diese Elemente zurückgegeben werden. Insbesondere wird also nicht ga-

rantiert, dass zwei *foreach*-Schleifen über eine gemeinsame Aufzählung die darin enthaltenen Elemente in der identischen Reihenfolge durchlaufen.

Gelegentlich steht die Aufzählung, über die mit einer *foreach*-Schleife iteriert werden soll, nicht fest, sondern soll erst zur Laufzeit von einer Methode erzeugt werden. Seit C# 2.0 gibt es dafür das Schlüsselwort *yield*, das genau diese Funktionalität ermöglicht.

Damit eine Methode als Aufzählung für eine *foreach*-Schleife dienen kann, muss sie über die Schnittstelle IEnumerable als Rückgabetyp verfügen. Allerdings gibt sie nicht die gesamte Aufzählung auf einmal zurück, sondern liefert bei jedem Methodenaufruf den nächsten Wert der Aufzählung.

Das Schlüsselwort *yield* bewirkt, dass der nächste Aufruf die Methode an der Stelle fortsetzt, an der sie im vorherigen Durchlauf verlassen wurde. *yield* ermöglicht also eine zeitweise Unterbrechung der Methodenausführung.

C#

```
using System;
using System.Collections;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Iterate over all square numbers.
            foreach (int i in this.GetNextSquareNumber())
            {
                // Print the square number to the console.
                Console.WriteLine(
                    "The next square number is " + i + ".");
            }
        }

        /// <summary>
        /// Gets the next square number.
        /// </summary>
        /// <returns>The next square number.</returns>
        private IEnumerable GetNextSquareNumber()
        {
            // Initialize the square numbers with 0.
            int i = 0;

            // Iterate endlessly over all numbers.
            while (true)
            {
```

```
37                  // Return the current square number to the
38                  // caller.
39                  yield return i * i;
40
41                  // Increase i by 1.
42                  i++;
43              }
44          }
45      }
46  }
```

Kapitel 19
Linq

19.1 Was ist Linq?

Während es in früheren Versionen von C# unter Umständen sehr aufwändig war, einzelne Elemente innerhalb einer Aufzählung zu suchen oder diese zu sortieren, stellt C# dafür seit der Version 3.0 eine eigene Abfragesprache zur Verfügung, die als Language Integrated Query, abgekürzt Linq, bezeichnet wird.

Um beispielsweise aus einem Array, das Elemente des Typs *string* enthält, alle diejenigen zu ermitteln, deren Wert mit einem bestimmten Buchstaben beginnt und diese alphabetisch sortiert auszugeben, wurden zumindest eine Schleife und zahlreiche *if*-Anweisungen benötigt. Seit der Version 3.0 von C# gibt es dafür eine eigene Abfragesprache, deren Syntax sich an der Datenbanksprache SQL orientiert, die aber über vollständige Unterstützung in C# und Visual Studio verfügt. Durch die Integration in C# wird Linq ebenso wie der übrige Code durch den Compiler in MSIL übersetzt, so dass auf Linq basierende Abfragen auf Fehler überprüft werden können. Im Gegensatz zu klassischen Abfragen, die beispielsweise als Zeichenketten innerhalb von C# vorliegen, können Fehler auf diese Art bereits vor der Ausführung erkannt werden.

Außerdem werden in Linq geschriebene Abfragen ebenfalls von der Common Language Runtime ausgeführt und nutzen daher wie C# ebenfalls die Vorteile von verwalteter Ausführung.

19.2 Abfrageoperatoren

Die einfachste Abfrage, die in Linq geschrieben werden kann, ermittelt alle Elemente aus einer Aufzählung, gibt also die Aufzählung selbst zurück, ohne diese zu durchsuchen oder zu sortieren. Um die Fähigkeiten von Linq nutzen zu können, muss der Namensraum System.Linq eingebunden werden, der sich in der Assembly System.Core befindet.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Eine Abfrage wird in Linq mit Hilfe des Schlüsselwortes *form* eingeleitet, dem ein Bezeichner für ein einzelnes Element folgt. Die Wahl dieses Bezeichners ist beliebig und vergleichbar mit der Wahl des Bezeichners innerhalb einer *foreach*-Schleife.

Im Anschluss wird mit Hilfe des Schlüsselwortes *in* angegeben, aus welcher Aufzählung die Elemente stammen, mit dem Schlüsselwort *select* wird schließlich das jeweilige Element als relevant für die Ergebnismenge ausgewählt.

C#

```
 1 using System;
 2 using System.Linq;
 3
 4 namespace GoloRoden.GuideToCSharp
 5 {
 6     /// <summary>
 7     /// Represents the application class.
 8     /// </summary>
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an array of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
19
20             // Get all colors.
21             var result =
22                 from c in colors
23                 select c;
24
25             // Print all colors to the console.
26             foreach (var color in result)
27             {
28                 Console.WriteLine(color);
29             }
30         }
31     }
32 }
```

Obwohl die Ergebnismenge nur aus Elementen des Typs *string* besteht, ist es in der Praxis üblich, den Typ einer in Linq geschriebenen Abfrage mit Hilfe von *var* zu definieren.

Ebenso könnte die Schleifenvariable der *foreach*-Schleife als *string* definiert werden, da jedes einzelne Element diesem Typ entspricht, aber auch dies ist in der Praxis unüblich.

Um die Ergebnismenge zu sortieren, kann das Schlüsselwort *orderby* verwendet werden, wobei nach diesem ein Ausdruck angegeben werden muss, welcher die

Sortierreihenfolge definiert. Es ist also nicht nur möglich, nach dem Element an sich zu sortieren, sondern es kann beispielsweise auch eine Eigenschaft oder ein Delegat angegeben werden, der die Sortierung vornimmt.

C#

```
using System;
using System.Linq;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Define an array of colors.
            string[] colors =
                new string[] { "Red", "Green", "Blue" };

            // Get all colors in alphabetical order.
            var result =
                from c in colors
                orderby c
                select c;

            // Print all colors to the console.
            foreach (var color in result)
            {
                Console.WriteLine(color);
            }
        }
    }
}
```

Alternativ zu der aufsteigenden Sortierung können die Elemente der Ergebnismenge durch die Angabe des zusätzlichen Schlüsselwortes *descending* auch absteigend sortiert werden.

C#

```
using System;
using System.Linq;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
```

```
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an array of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
19
20             // Get all colors in reversed alphabetical order.
21             var result =
22                 from c in colors
23                 orderby c descending
24                 select c;
25
26             // Print all colors to the console.
27             foreach (var color in result)
28             {
29                 Console.WriteLine(color);
30             }
31         }
32     }
33 }
```

Zusätzlich kann die Ergebnismenge mit dem Schlüsselwort *where* durchsucht werden, so dass nur einige Elemente in der Ergebnismenge enthalten sind. Im folgenden Beispiel werden beispielsweise nur die Elemente in die Ergebnismenge aufgenommen, deren Anfangsbuchstabe ein R ist.

C#

```
 1 using System;
 2 using System.Linq;
 3
 4 namespace GoloRoden.GuideToCSharp
 5 {
 6     /// <summary>
 7     /// Represents the application class.
 8     /// </summary>
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an array of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
19
20             // Get all colors whose name starts with an R
```

```
21              // in reversed alphabetical order.
22              var result =
23                  from c in colors
24                  where c.StartsWith("R")
25                  orderby c descending
26                  select c;
27
28              // Print all colors to the console.
29              foreach (var color in result)
30              {
31                  Console.WriteLine(color);
32              }
33          }
34      }
35  }
```

Falls der Typ der Elemente der Ergebnismenge kein einfacher Typ, sondern ein komplexer Typ wie beispielsweise ein Objekt ist, kann es gewünscht sein, nicht das gesamte Element in die Ergebnismenge aufzunehmen, sondern nur eine oder mehrere Eigenschaften.

Sofern nur eine einzelne Eigenschaft als Element in die Ergebnismenge aufgenommen werden soll, genügt es, diese an Stelle des eigentlichen Objekts bei *select* anzugeben. Sollen statt dessen mehrere Eigenschaften verwendet werden, können diese mit Hilfe von Objektinitialisierern in ein neues Objekt von einem anonymen Typ zusammengeführt werden.

C#

```
 1 using System;
 2 using System.Linq;
 3
 4 namespace GoloRoden.GuideToCSharp
 5 {
 6     /// <summary>
 7     /// Represents the application class.
 8     /// </summary>
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an arrays of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
19
20             // Get all colors whose name starts with R in
21             // reversed alphabetical order, and limit the
22             // result set to the name and length of the
23             // colors.
24             var result =
```

```
25                  from c in colors
26                  where c.StartsWith("R")
27                  orderby c descending
28                  select new { Name = c, c.Length };
29
30              // Print all colors to the console.
31              foreach (var color in result)
32              {
33                  Console.WriteLine(
34                      color.Name + " (" + color.Length + ")");
35              }
36          }
37      }
38  }
```

Spätestens an dieser Stelle wird deutlich, warum der Datentyp bei Linq nie spezifisch, sondern in der Regel mit dem Schlüsselwort *var* definiert wird. Ändert sich die Auswahl der in der Ergebnismenge enthaltenen Eigenschaften, so müssen die verwendeten Typen nicht angepasst werden.

Mit Linq ist es jedoch nicht nur möglich, einzelne Elemente in einer Ergebnismenge zusammenzufassen, zusätzlich kann diese Menge ihrerseits gruppiert werden. Dazu dient das Schlüsselwort *group*, das immer in Kombination mit dem Schlüsselwort *by* verwendet werden muss, und das angibt, welche Elemente wie gruppiert werden sollen. Wird *group* innerhalb einer Abfrage verwendet, so kann diese Abfrage kein *select* enthalten.

Eine auf diese Art erzeugte Gruppe von Elementen enthält das Kriterium, mit dessen Hilfe sie erstellt wurde, in der Eigenschaft Key und kann ihrerseits wiederum als Quelle für eine Schleife oder eine weitere Abfrage dienen. Im folgenden Beispiel werden jeweils all jene Elemente zu einer Gruppe zusammengefasst, deren Namen die gleiche Länge haben.

C#

```
 1 using System;
 2 using System.Linq;
 3
 4 namespace GoloRoden.GuideToCSharp
 5 {
 6     /// <summary>
 7     /// Represents the application class.
 8     /// </summary>
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an array of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
```

```
19
20              // Get all colors ordered alphabetically and put
21              // them in groups depending on the length of the
22              // color's name.
23              var result =
24                  from c in colors
25                  orderby c
26                  group c by c.Length;
27
28              // Iterate over all groups.
29              foreach (var group in result)
30              {
31                  // Print the group's key to the console.
32                  Console.WriteLine(group.Key);
33
34                  // Iterate over all colors within the group.
35                  foreach (var color in group)
36                  {
37                      // Print the color to the console.
38                      Console.WriteLine(color);
39                  }
40              }
41          }
42      }
43  }
```

Neben den Möglichkeiten, die sich direkt aus der Verwendung solcher Abfragen ergeben, erweitert Linq sämtliche Aufzählungstypen mit Hilfe von Erweiterungsmethoden um weitere Methoden zur Manipulation der Ergebnismenge.

Soll beispielsweise nur das erste Element einer Ergebnismenge ausgewertet werden, so kann dies mit Hilfe der Methode First abgerufen werden.

C#

```
 1 using System;
 2 using System.Linq;
 3
 4 namespace GoloRoden.GuideToCSharp
 5 {
 6     /// <summary>
 7     /// Represents the application class.
 8     /// </summary>
 9     public class Program
10     {
11         /// <summary>
12         /// Executes the application.
13         /// </summary>
14         public static void Main()
15         {
16             // Define an array of colors.
17             string[] colors =
18                 new string[] { "Red", "Green", "Blue" };
19
```

```
20              // Get all colors ordered alphabetically.
21              var result =
22                  from c in colors
23                  orderby c
24                  select c;
25
26              // Get the first color from the result.
27              string color = result.First();
28          }
29      }
30  }
```

Ebenso kann eine gewisse Anzahl an ersten Elementen abgerufen werden, wozu die Methode Take dient. Im folgenden Beispiel werden immer nur die ersten beiden Elemente der Ergebnismenge zurückgegeben, unabhängig davon, wie viele Elemente tatsächlich enthalten sind.

C#

```
 1  using System;
 2  using System.Linq;
 3
 4  namespace GoloRoden.GuideToCSharp
 5  {
 6      /// <summary>
 7      /// Represents the application class.
 8      /// </summary>
 9      public class Program
10      {
11          /// <summary>
12          /// Executes the application.
13          /// </summary>
14          public static void Main()
15          {
16              // Define an array of colors.
17              string[] colors =
18                  new string[] { "Red", "Green", "Blue" };
19
20              // Get all colors ordered alphabetically.
21              var result =
22                  from c in colors
23                  orderby c
24                  select c;
25
26              // Get the two top colors from the result.
27              var topColors = result.Take(2);
28          }
29      }
30  }
```

19.3 Lambdaausdrücke

Intern werden in Linq geschriebene Abfragen in Aufrufe von Erweiterungsmethoden und Lambdaausdrücke umgewandelt. Dies geschieht ähnlich wie bei der *foreach*-Schleife im Hintergrund durch den Compiler, ohne dass der Entwickler dies bemerkt.

Beispielsweise wird die Abfrage

C#

```
1 // Get all colors whose name starts with an R in reversed
2 // alphabetical order.
3 var result =
4     from c in colors
5     where c.StartsWith("R")
6     orderby c descending
7     select c;
```

von C# in

C#

```
1 // Get all colors whose name starts with an R in reversed
2 // alphabetical order.
3 var result =
4     colors.Where(c => c.StartsWith("R"))
5     .OrderByDescending(c => c).Select(c => c);
```

umgewandelt.

Kapitel 20
Ausnahmen

20.1 Was sind Ausnahmen?

Wird die Frage gestellt, aus welchem Grund Anwendungen entwickelt werden, so geschieht dies zunächst im Wesentlichen zur Bewältigung einer Aufgabe und zur Lösung der mit dieser Aufgabe einhergehenden Problemen. Obwohl dies den initialen Beweggrund darstellt, enthält jede Anwendung zahlreiche weitere Aspekte, die bei der Entwicklung neben der eigentlichen Domäne beachtet werden müssen.

Dazu zählen beispielsweise Aspekte wie Sicherheit, Ausführungsgeschwindigkeit oder Stabilität. Ein wesentlicher Faktor, der sich in direkter Konsequenz auf die Qualität einer jeden Anwendung auswirkt, ist der Umgang mit potenziellen Fehlern, die während der Ausführung der Anwendung auftreten können.

Anwendungen, die auf Basis der Win32-API und COM entwickelt werden, verfügen nicht über ein einheitliches System, wie Fehler ausgelöst und behandelt werden. Einige Methoden der Win32-API verwenden Rückgabewerte, wobei es dem Entwickler obliegt, den Rückgabewert überhaupt auszuwerten und ihn außerdem entsprechend seiner Bedeutung zu interpretieren. Andere Methoden wiederum handhaben die Fehlerbehandlung anders, wobei dies nicht nur von der verwendeten Plattform, sondern zusätzlich noch von der verwendeten Sprache abhängt.

.NET hingegen stellt allen Anwendungen, die für .NET entwickelt werden, ein einheitliches System zur Fehlerbehandlung zur Verfügung. Dieses basiert auf sogenannten Ausnahmen, wobei eine Ausnahme einen konkreten Fehlerfall darstellt. Ein wesentlicher Unterschied zwischen Ausnahmen und den klassischen Rückgabewerten liegt darin, wie sie behandelt werden.

Während es früher Aufgabe des Entwicklers war, auf die Behandlung zu achten, brechen Ausnahmen die Ausführung der Anwendung ab. Damit dies jedoch nicht bei jeder Ausnahme geschieht, bietet C# entsprechende Möglichkeiten, auf Ausnahmen zu reagieren, so dass die Ausführung nach der Fehlerbehandlung fortgesetzt werden kann – erfolgt jedoch keine Fehlerbehandlung, so wird die Ausführung der Anwendung abgebrochen. Es ist also nicht mehr möglich, Fehler zu ignorieren.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Ausnahmen können in .NET so wohl von der Common Language Runtime ausgelöst werden, wenn eine Anwendung beispielsweise versucht, auf eine nicht vorhandene Ressource zuzugreifen, sie können aber auch vom Entwickler gezielt eingesetzt werden, um Fehlersituationen innerhalb der Anwendung zu kennzeichnen.

Damit der fehlerbehandelnde Code auf eine Ausnahme möglichst geeignet reagieren kann, enthalten Ausnahmen neben einer ausführlichen, detaillierten Fehlermeldung auch den sogenannten Aufrufstapel, mit dessen Hilfe sich nachverfolgen lässt, an welcher Stelle in der Ausführung sich die Anwendung gerade befindet. Dabei enthält der Aufrufstapel nicht nur Informationen zu der Klasse, Methode und Zeile, welche die Ausnahme ausgelöst hat, sondern auch zur Aufrufhierarchie.

Des weiteren enthält eine Ausnahme unter Umständen noch weitere, sogenannte innere Ausnahmen, wenn beispielsweise während der Fehlerbehandlung ein weiterer Fehler aufgetreten ist, allerdings Informationen zu beiden Fehlern an die nächste Fehlerbehandlung weitergereicht werden sollen.

20.2 Ausnahmen behandeln

Prinzipiell werden Ausnahmen immer dort behandelt, wo sie auftreten. Das heißt, tritt eine Ausnahme innerhalb einer Methode auf, obliegt es dieser Methode, sich um die Fehlerbehandlung zu kümmern. Geschieht dies nicht, so wird die Ausnahme an die aufrufende Methode weitergereicht, die sich ihrerseits nun um die Fehlerbehandlung kümmern kann.

Geschieht auch dies nicht, wird die Ausnahme wieder eine Ebene nach oben gereicht, bis sich entweder eine Methode findet, welche die Ausnahme behandelt, oder die oberste Ebene, also die Main-Methode, erreicht ist. Wird die Ausnahme auch dort nicht behandelt, wird die Ausführung der Anwendung abgebrochen und .NET gibt die Fehlermeldung der Ausnahme an den Benutzer aus.

Um eine Ausnahme abzufangen, bietet C# die beiden Schlüsselwörter *try* und *catch*. Beide verfügen über einen Rumpf, der durch geschweifte Klammern eingeschlossen wird. Während *try* die Anweisungen umschließt, die potenziell eine Ausnahme auslösen könnten, stellt *catch* den fehlerbehandelnden Code zur Verfügung.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
```

```
13          public static void Main()
14          {
15              try
16              {
17                  // Define two operands.
18                  int operand1 = 23;
19                  int operand2 = 0;
20
21                  // Cause an exception.
22                  int result = operand1 / operand2;
23
24                  // Print the result to the console.
25                  Console.WriteLine(
26                      "The result is " + result + ".");
27              }
28              catch
29              {
30                  // Catch any exceptions.
31                  Console.WriteLine("Division by zero!");
32              }
33          }
34      }
35  }
```

Im vorangegangenen Beispiel löst die Zeile, in der versucht wird, den einen Operanden durch den anderen zu teilen, eine Ausnahme aus, da die Division durch Null mathematisch nicht definiert ist. Die Ausführung innerhalb des *try*-Blocks wird daraufhin abgebrochen, weshalb die Ausgabe des Ergebnisses nicht erfolgt. Statt dessen verzweigt die Ausführung in den *catch*-Block, der eine entsprechende Fehlermeldung ausgibt.

Ein solcher *catch*-Block reagiert allerdings nicht nur auf die aufgetretene DivideByZeroException, sondern auf sämtliche Ausnahmen. Unter Umständen ist dieses Verhalten allerdings nicht gewünscht, da nur gezielt einige Ausnahmen behandelt werden sollen. Dazu ist es möglich, den Typ der zu behandelnden Ausnahme als Parameter anzugeben.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
```

```
15              try
16              {
17                  // Define two operands.
18                  int operand1 = 23;
19                  int operand2 = 0;
20
21                  // Cause an exception.
22                  int result = operand1 / operand2;
23
24                  // Print the result to the console.
25                  Console.WriteLine(
26                      "The result is " + result + ".");
27              }
28              catch (DivideByZeroException)
29              {
30                  // Catch a DivideByZeroException.
31                  Console.WriteLine("Division by zero!");
32              }
33          }
34      }
35  }
```

Derzeit ist es in C# allerdings nicht möglich, mehrere Typen anzugeben. Sollen also mehrere Ausnahmen behandelt werden, müssen mehrere *catch*-Blöcke verwendet werden.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              try
16              {
17                  // Define two operands.
18                  int operand1 = 23;
19                  int operand2 = 0;
20
21                  // Cause an exception.
22                  int result = operand1 / operand2;
23
24                  // Print the result to the console.
25                  Console.WriteLine(
26                      "The result is " + result + ".");
```

```
27              }
28              catch (DivideByZeroException)
29              {
30                  // Catch a DivideByZeroException.
31                  Console.WriteLine("Division by zero!");
32              }
33              catch (OverflowException)
34              {
35                  // Catch an OverflowException.
36                  Console.WriteLine("Result too large!");
37              }
38          }
39      }
40  }
```

Die einzige Möglichkeit, diese Einschränkung zu umgehen, ist, eine gemeinsame Basisklasse als Typ anzugeben, sofern eine solche existiert. Prinzipiell leiten alle Ausnahmen von der Klasse System.Exception ab, manche verfügen allerdings über eine andere Basisklasse, die ihrerseits erst von System.Exception ableitet. Ein typisierter *catch*-Block behandelt also nicht nur die Ausnahmen, die dem angegebenen Typ entsprechen, sondern auch all jene, die von diesem Typ abgeleitet sind.

Generell gilt allerdings, dass Ausnahmen so lokal und so spezifisch wie möglich behandelt werden sollten.

Sofern mehrere *catch*-Blöcke vorhanden sind, muss deren Reihenfolge beachtet werden. Da C# immer den frühesten passenden *catch*-Block mit der Fehlerbehandlung betraut, ist es wichtig, Blöcke für spezifische Ausnahmen vor solchen für allgemeinere Ausnahmen zu positionieren.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             try
16             {
17                 // Define two operands.
18                 int operand1 = 23;
19                 int operand2 = 0;
20
21                 // Cause an exception.
22                 int result = operand1 / operand2;
```

```
23
24                  // Print the result to the console.
25                  Console.WriteLine(
26                      "The result is " + result + ".");
27              }
28              catch
29              {
30                  // This block is executed on every exception
31                  // since it catches any exception.
32              }
33              catch(DivideByZeroException)
34              {
35                  // This block is executed never.
36              }
37          }
38      }
39  }
```

Eine Fähigkeit von Ausnahmen wurde noch nicht vorgestellt: Der Zugriff auf die in einer Ausnahme enthaltenen Informationen wie Fehlermeldung, Aufrufstapel und innere Ausnahmen. Dazu ist es nötig, eine Ausnahme mit einem Variablennamen zu kennzeichnen, so dass darauf innerhalb des *catch*-Blocks zugegriffen werden kann. Es hat sich in der Praxis eingebürgert, Ausnahmen mit der Abkürzung ex zu benennen, obwohl dies nicht den Namenskonventionen für lokale Variablen entspricht, weshalb diese Bezeichnung in den folgenden Beispiele nicht verwendet wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             try
16             {
17                 // Define two operands.
18                 int operand1 = 23;
19                 int operand2 = 0;
20
21                 // Cause an exception.
22                 int result = operand1 / operand2;
23
24                 // Print the result to the console.
25                 Console.WriteLine(
```

```
26                      "The result is " + result + ".");
27              }
28              catch (DivideByZeroException exception)
29              {
30                  // Catch a DivideByZeroException.
31                  Console.WriteLine(exception.Message);
32              }
33          }
34      }
35  }
```

Auch ein Weiterreichen und somit ein erneutes Auslösen einer Ausnahme innerhalb eines *catch*-Blocks ist möglich, was in C# mit Hilfe des Schlüsselwortes *throw* geschieht. Es wird kein weiterer Parameter benötigt, da *throw* immer die Ausnahme weiterreicht, in deren fehlerbehandelndem Block sich der entsprechende Aufruf befindet.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              try
16              {
17                  // Define two operands.
18                  int operand1 = 23;
19                  int operand2 = 0;
20
21                  // Cause an exception.
22                  int result = operand1 / operand2;
23
24                  // Print the result to the console.
25                  Console.WriteLine(
26                      "The result is " + result + ".");
27              }
28              catch (DivideByZeroException exception)
29              {
30                  // Catch the DivideByZeroException.
31                  Console.WriteLine(exception.Message);
32
33                  // Rethrow the exception.
34                  throw;
```

```
35              }
36          }
37      }
38  }
```

Dennoch kann eine Ausnahme als Parameter angegeben werden, wobei dabei allerdings der Aufrufstapel verloren geht, weshalb dies in der Praxis als schlechter Stil angesehen wird.

In einigen Fällen kann es vorkommen, dass Code im Anschluss an einen *try-catch*-Block ausgeführt werden muss, unabhängig davon, ob der *try*-Block vollständig erfolgreich durchlaufen wurde oder nicht, wenn also eine Ausnahme ausgelöst wurde. Solcher Code könnte beispielsweise dazu dienen, eine geöffnete Verbindung zu einer Datenbank zu schließen oder sonstige Ressourcen wieder freizugeben. Im einfachsten Fall genügt es, solchen Code hinter dem *catch*-Block anzugeben.

C#

```
 1 try
 2 {
 3     // TODO gr: Do something.
 4     //          2008-01-02
 5 }
 6 catch
 7 {
 8     // TODO gr: Handle eventually thrown exceptions.
 9     //          2008-01-02
10 }
11
12 // TODO gr: Clean up.
13 //          2008-01-02
```

Führt jedoch mindestens einer der beiden Blöcke ein *return* aus und verlässt die aktuelle Methode damit, oder reicht der *catch*-Block die Ausnahme an eine höhergelegene Methode weiter, wird der entsprechende Code nicht mehr ausgeführt.

Eine denkbare Lösung wäre, den entsprechenden Code in beiden Blöcken einzufügen, doch dies verschlechtert die Wartbarkeit und erhöht die Unübersichtlichkeit. Statt dessen stellt C# das Schlüsselwort *finally* zur Verfügung, das einen weiteren Block nach *try* und *catch* einleitet, dessen Inhalt in jedem Fall ausgeführt wird – sogar dann, wenn durch einen der beiden Blöcke ein *return* oder ein *throw* ausgeführt wird.

C#

```
 1 Try
 2 {
 3     // TODO gr: Do something.
 4     //          2008-01-02
 5
 6     return;
 7     // Return to the caller.
 8 }
```

```
 9 catch
10 {
11     // TODO gr: Handle eventually thrown exceptions.
12     //          2008-01-02
13
14     // Rethrow the exception.
15     throw;
16 }
17 finally
18 {
19     // TODO gr: Clean up.
20     //          2008-01-02
21 }
```

20.3 Benutzerdefinierte Ausnahmen

Wie zu Anfang bereits erwähnt ist es dem Entwickler möglich, eigene Ausnahmen zu definieren, um Fehlerzustände innerhalb der Anwendung zu kennzeichnen. Prinzipiell ist eine solche benutzerdefinierte Ausnahme nichts anderes, als eine direkt oder indirekt von System.Exception abgeleitete Klasse.

Um systembedingte und benutzerdefinierte Ausnahmen unterscheiden zu können, ist es in der Praxis üblich, eigene Ausnahmeklassen nicht von System.Exception, sondern von der Klasse System.ApplicationException abzuleiten, die ihrerseits wiederum von System.Exception ableitet.

Ausgelöst wird eine benutzerdefinierte Ausnahme mit Hilfe des bereits bekannten Schlüsselwortes *throw*, wobei diesem als Parameter eine neue Instanz der entsprechenden Ausnahme übergeben wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a custom defined exception.
 7     /// </summary>
 8     public class MyException : ApplicationException
 9     {
10     }
11
12     /// <summary>
13     /// Represents the application class.
14     /// </summary>
15     public class Program
16     {
17         /// <summary>
18         /// Executes the application.
```

```
19          /// </summary>
20          public static void Main()
21          {
22              // Throw a custom defined exception.
23              throw new MyException();
24          }
25      }
26  }
```

Es wird in der Praxis als guter Stil angesehen, die Standardkonstruktoren der Basisklasse System.ApplicationException zu überschreiben, um auch benutzerdefinierte Ausnahmen durch Angabe der entsprechenden Parameter mit einer Fehlermeldung und inneren Ausnahmen ausstatten zu können.

20.4 Leistung und Ressourcenbedarf

Im Zusammenhang mit Ausnahmen liest und hört man häufig, dass diese nicht eingesetzt werden sollten, da sie sehr leistungshungrig seien. Aus dieser Aussage ergibt sich direkt die Frage, wann Ausnahmen überhaupt eingesetzt werden sollten.

Prinzipiell ergibt sich die Antwort auf diese Frage bereits aus dem Begriff einer Ausnahme: Sie stellen Ausnahmesituationen dar. Das heißt, Ausnahmen sind explizit nicht dazu gedacht, bedenkenlos an den verschiedensten Stellen innerhalb einer Anwendung eingesetzt zu werden. Sofern es möglich ist, einen Fehler im Vorfeld abzufangen, sollte dies dem Einsatz einer Ausnahme vorgezogen werden.

Beispielsweise würde man in dem Beispiel, das die DivideByZeroException abfängt, in der Praxis keine Ausnahme einsetzen, sondern im Vorfeld mit Hilfe einer *if*-Abfrage prüfen, ob durch 0 geteilt werden soll. Insbesondere, wenn solche Berechnungen innerhalb von Schleifen auftreten, kann dadurch die Leistung der Anwendung durchaus gesteigert werden.

Dies liegt daran, dass für jede Ausnahme, die ausgelöst wird, der Aufrufstapel ermittelt werden muss, was bei einer entsprechend tiefen Verschachtelung von Methodenaufrufen unter Umständen aufwändig sein kann.

Obwohl Ausnahmen also nicht wahlfrei eingesetzt werden sollten, gibt es dennoch Fälle, in denen ihr Einsatz nicht verzichtbar ist. Dann nämlich, wenn Fehler nicht erwartbar sind und auf Ausnahmesituationen reagiert werden muss. In einem solchen Fall ist es in der Regel allerdings ohnehin nötig, den Benutzer zu informieren und ihn das weitere Vorgehen bestimmen zu lassen, weshalb es in einer solchen Situation nicht darauf ankommt, ob eine Ausnahme schnell oder langsam erzeugt wird – die Anwendung gelangt auf beide Arten zum Stillstand.

Zusammengefasst lässt sich also sagen, dass Ausnahmen entgegen ihrem Ruf durchaus eingesetzt werden können, dass dies allerdings gezielt und mit Bedacht geschehen sollte. Insbesondere sollten Fehlersituationen bereits im Vorfeld vermieden werden, sofern dies möglich ist.

Kapitel 21
Attribute

21.1 Was sind Attribute?

Wie bereits im Rahmen der Fehlerbehandlung erwähnt, gibt es Aspekte in Anwendungen, die über die reine fachliche Domäne hinausgehen. Als Beispiele waren dort unter anderem Sicherheit, Ausführungsgeschwindigkeit und Stabilität genannt. Auch die Fehlerbehandlung zählt zu diesen nicht-fachlichen Aspekten.

Neben Aspekten, die per Code definiert werden, bietet C# auch die Möglichkeit, Aspekte deklarativ umzusetzen, das heißt, ohne dass Code zu ihrer Umsetzung geschrieben werden müsste. Statt dessen werden die entsprechenden Stellen innerhalb der Anwendung mit sogenannten Attributen markiert, die Einfluss auf die Semantik des markierten Codes haben.

Beispielsweise gibt es für Enumerationen ein Attribut, das bewirkt, dass die interne Abbildung der Enumeration auf Ganzzahlen dem Schema der Zweierpotenzen folgt, statt die Zahlen lediglich fortlaufend zuzuordnen. Dieses Attribut ist beispielsweise dann äußerst nützlich, wenn die einzelnen Werte einer Enumeration binär verknüpft werden sollen.

Um ein Attribut in C# zu verwenden, genügt es, das entsprechende Attribut vor dem zu markierenden Abschnitt innerhalb eckiger Klammern anzugeben. Das Attribut, um die einer Enumeration zugeordneten Zahlen als Zweierpotenzen zu organisieren, heißt FlagsAttribute und befindet sich im Namensraum System.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Contains colors.
 7     /// </summary>
 8     [Flags]
 9     enum Colors
10     {
```

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

```
11          /// <summary>
12          /// Represents the red color.
13          /// </summary>
14          Red,    // = 1
15
16          /// <summary>
17          /// Represents the green color.
18          /// </summary>
19          Green,  // = 2
20
21          /// <summary>
22          /// Represents the blue color.
23          /// </summary>
24          Blue    // = 4
25      }
26  }
```

Wie das Beispiel zeigt, entfällt bei der Angabe eines Attributs das Suffix Attribute, obwohl der interne Bezeichner der Klasse FlagsAttribute lautet. Attribute ermöglichen prinzipiell also, die Semantik von Code auf deklarativem Wege zu verändern.

Die meisten Attribute ermöglichen außerdem, sie mit Hilfe von Parametern an den jeweiligen Kontext anzupassen. Prinzipiell werden Parameter zu Attributen ähnlich denen zu einer Methode angegeben, innerhalb runder Klammern. Allerdings werden bei Attributen zwei Typen von Parametern unterschieden: Positions- und Namensparameter.

Während Positionsparameter eine feste Reihenfolge besitzen, in der sie angegeben werden müssen, ist diese bei Namensparametern frei wählbar. Allerdings muss diesen ein Name vorangestellt werden, damit C# den Parameter entsprechend zuordnen kann. Die meisten Attribute folgen dem Schema, dass Positionsparameter zwingende, Namensparameter allerdings nur optionale Parameter darstellen. Sofern Namensparameter angegeben werden, muss dies nach den Positionsparametern erfolgen.

Ein Beispiel für Positionsparameter bietet das Attribut ObsoleteAttribute, das genutzt werden kann, um Methoden oder Typen zu kennzeichnen, die aus Kompatibilitätsgründen noch enthalten sind, allerdings nicht mehr verwendet werden sollten. Es gibt dieses Attribut in drei Ausführungen: Ohne Parameter, mit einem und mit zwei Parametern. Der erste Parameter definiert eine Fehlermeldung, die C# ausgeben soll, wenn die Methode oder der Typ verwendet wird, der zweite Parameter legt mit Hilfe eines logischen Wertes fest, ob der Compiler eine Warnung oder einen Fehler erzeugen soll.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents a foo class.
7      /// </summary>
```

```
 8     public class Foo
 9     {
10         /// <summary>
11         /// Does nothing.
12         /// </summary>
13         [Obsolete]
14         public void Bar()
15         {
16         }
17
18         /// <summary>
19         /// Does nothing.
20         /// </summary>
21         /// <param name="param0">A foo parameter.</param>
22         [Obsolete("Use method X instead.")]
23         public void Bar(int param0)
24         {
25         }
26
27         /// <summary>
28         /// Does nothing.
29         /// </summary>
30         /// <param name="param0">A foo parameter.</param>
31         /// <param name="param1">Another foo parameter.
32         /// </param>
33         [Obsolete("Use method X instead.", true)]
34         public void Bar(int param0, int param1)
35         {
36         }
37     }
38 }
```

21.2 Benutzerdefinierte Attribute

Außer den vordefinierten Attributen bietet C# auch die Möglichkeit, eigene Attribute zu definieren. Dies geschieht, indem eine eigene Klasse definiert wird, die von der Basisklasse Attribute im Namensraum System ableitet und deren Name auf das Suffix Attribute endet.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the author attribute.
 7     /// </summary>
 8     public class AuthorAttribute : Attribute
 9     {
```

```
10      }
11  }
```

Um dieses Attribut mit Parametern zu versehen, werden zum einen Felder benötigt, welche die entsprechenden Werte aufnehmen. Außerdem muss das Attribut für Positionsparameter mindestens mit einem Konstruktor versehen werden, für Namensparameter muss es entsprechende Eigenschaften geben.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the author attribute.
 7     /// </summary>
 8     public class AuthorAttribute : Attribute
 9     {
10         /// <summary>
11         /// Contains the name.
12         /// </summary>
13         private string _name;
14
15         /// <summary>
16         /// Contains the email address.
17         /// </summary>
18         private string _eMail;
19
20         /// <summary>
21         /// Gets or sets the name.
22         /// </summary>
23         /// <value>The name.</value>
24         public string Name
25         {
26             get
27             {
28                 return this._name;
29             }
30
31             set
32             {
33                 this._name = value;
34             }
35         }
36
37         /// <summary>
38         /// Gets or sets the email.
39         /// </summary>
40         /// <value>The email.</value>
41         public string EMail
42         {
43             get
```

```
44              {
45                  return this._eMail;
46              }
47
48              set
49              {
50                  this._eMail = value;
51              }
52          }
53
54          /// <summary>
55          /// Initializes a new instance of the
56          /// AuthorAttribute type.
57          /// </summary>
58          /// <param name="name">The name.</param>
59          public AuthorAttribute(string name)
60          {
61              // Set the values.
62              this._name = name;
63          }
64      }
65  }
```

In diesem Beispiel ist es auf Grund des Konstruktors notwendig, den Namen des Autors anzugeben, die E-Mail-Adresse ist allerdings optional. Methoden und Typen können, sofern sie mit diesem Attribut markiert werden, mit der Angabe versehen werden, wer sie entwickelt hat und für sie zuständig ist, was beispielsweise in Teams nützlich zu wissen sein kann.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     [Author("Golo Roden", EMail = "webmaster@goloroden.de")]
 9     public class Foo
10     {
11     }
12 }
```

21.3 Ziele von Attributen

Attribute selbst können wiederum mit Attributen versehen werden, was in C# unter anderem dafür genutzt wird, die potenziellen Ziele für Attribute vorzugeben.

Ein Ziel ist ein Element innerhalb des Codes, auf welches das Attribut angewendet werden kann, wie beispielsweise eine Methode, ein Parameter oder eine Klasse.

Ziele werden in C# mit Hilfe des Attributes AttributeUsageAttribute definiert, das als Parameter eine bitweise-oder-verknüpfte Liste von Zielen erwartet. Um die Verwendung des Attributs AuthorAttribute beispielsweise auf Methoden und Klassen einzuschränken, werden dem AttributeUsageAttribut die entsprechenden Ziele übergeben.

C#

```
using System;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the author attribute.
    /// </summary>
    [AttributeUsage(AttributeTargets.Method $\vert $
        AttributeTargets.Class)]
    public class AuthorAttribute : Attribute
    {
        /// <summary>
        /// Contains the name.
        /// </summary>
        private string _name;

        /// <summary>
        /// Contains the email.
        /// </summary>
        private string _eMail;

        /// <summary>
        /// Gets or sets the name.
        /// </summary>
        /// <value>The name.</value>
        public string Name
        {
            get
            {
                return this._name;
            }

            set
            {
                this._name = value;
            }
        }

        /// <summary>
        /// Gets or sets the email.
        /// </summary>
        /// <value>The email.</value>
        public string EMail
```

```
44          {
45              get
46              {
47                  return this._eMail;
48              }
49
50              set
51              {
52                  this._eMail = value;
53              }
54          }
55
56          /// <summary>
57          /// Initializes a new instance of the
58          /// AuthorAttribute type.
59          /// </summary>
60          /// <param name="name">The name.</param>
61          public AuthorAttribute(string name)
62          {
63              this._name = name;
64          }
65      }
66  }
```

Kapitel 22
Speicherverwaltung

22.1 Speicherverbrauch

Wird eine auf .NET basierende Anwendung ausgeführt, so wird nicht nur sie in den Speicher geladen, sondern auch die Common Language Runtime und die Klassenbibliothek von .NET. Aus diesem Grund verbraucht eine Anwendung, die auf .NET basiert, zunächst deutlich mehr Speicher als eine vergleichbare Anwendung, die beispielsweise ausschließlich auf der Win32-API aufbaut.

Seit .NET 2.0 werden die Systemkomponenten allerdings nur ein einziges Mal geladen und anschließend allen derzeit im Speicher befindlichen Anwendungen zur Verfügung gestellt, so dass der hohe Speicherbedarf bei zahlreichen gleichzeitig laufenden Anwendungen relativiert wird. Obwohl diese Maßnahme den Speicherbedarf von Anwendungen für .NET bereits deutlich gesenkt hat, scheinen sie doch übermäßig viel Speicher zu verbrauchen.

Verlässt man sich auf die Angaben, die beispielsweise der Taskmanager von Windows anzeigt, wird allerdings ein Detail des Speichermanagements von .NET übersehen: .NET reserviert für jede gestartete Anwendung zunächst zu viel freien Speicher, so dass nicht während der Ausführung der Anwendung aufwändig neuer Speicher angefordert werden muss. Der Anwendung steht also in jedem Fall genügend Speicher zur Verfügung, was der Ausführungsgeschwindigkeit zugute kommt.

Wird allerdings der Speicher im System knapp, da in der Zwischenzeit weitere Anwendungen gestartet wurden, oder da der Speicherbedarf anderer gleichzeitig ausgeführter Anwendungen gestiegen ist, gibt .NET Teile des zwar reservierten, aber ungenutzen Speichers frei. Insofern liegt der Speicherbedarf einer auf .NET basierenden Anwendung deutlich niedriger, als man zunächst annehmen könnte.

22.2 Freigabe von Ressourcen

Die aus diesem Verhalten resultierende Frage ist, warum .NET den Speicher auf diese Art verwaltet. Um diese Frage beantworten zu können, muss man wissen, was intern geschieht, wenn Typen instanziiert werden.

Golo Roden, *Auf der Fährte von C#*
DOI: 10.1007/978-3-540-27889-4, © Springer 2008

Bisher wurde zwischen Werte- und Verweistypen unterschieden, die entweder direkt oder indirekt im Speicher verwaltet werden. Ein weiterer Unterschied zwischen diesen Arten von Typen besteht darin, wo im Speicher Instanzen dieser Typen abgelegt werden. Während Wertetypen im sogenannten Stack abgelegt werden, werden Verweistypen auf dem sogenannten Managed Heap gespeichert, und nur ein Verweis auf diese Speicherstelle wird im Stack abgelegt.

Auffällig ist, dass Objekte in C# zwar mit Hilfe des Operators *new* erzeugt werden können, dass sie aber – beispielsweise im Gegensatz zu C++ – nicht wieder freigegeben werden müssen. Dies liegt daran, dass C# die Bereinigung des Speichers um nicht mehr benötigte Objekte eigenständig mit einer entsprechenden Komponente durchführt, die als Garbage Collection oder Garbage Collector bezeichnet wird.

Da es notwendig sein kann, vor dem Freigeben des Speichers, der durch ein Objekt belegt ist, einige Aufräumarbeiten auszuführen, gibt es dafür eine eigene Methode, die als Finalisierer bezeichnet wird und deren Basisimplementierung sich als Finalize in *object* befindet. Innerhalb dieser Methode können beispielsweise Ressourcen freigegeben werden, die nicht unter der Verwaltung von .NET stehen, wie unter anderem COM-Objekte oder Win32-Handles. Allerdings muss darauf geachtet werden, in jedem Fall den Finalisierer der Basisklasse aufzurufen.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Finalizes this instance.
12         /// </summary>
13         protected override void Finalize()
14         {
15             // TODO gr: Clean up any managed and unmanaged
16             //          resources.
17             //          2008-01-01
18
19             // Call the base finalizer.
20             base.Finalize();
21         }
22     }
23 }
```

Da es durchaus geschehen kann, dass der händische Aufruf des Finalisierers in der Basisklasse vergessen wird, bietet C# die Möglichkeit, analog zu einem Konstruktor eine Methode als Destruktor zu implementieren, die diesen Aufruf implizit durchführt. Ein Destruktor folgt dem gleichen Namensschema wie der Konstruk-

tor, allerdings wird ihm eine Tilde als Präfix vorangestellt. Außerdem verfügt ein Destruktor nicht über einen Zugriffsmodifizierer. An Stelle von

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Finalizes this instance.
12         /// </summary>
13         protected override void Finalize()
14         {
15             // TODO gr: Clean up any managed and unmanaged
16             //          resources.
17             //          2008-01-02
18
19             // Call the base finalizer.
20             base.Finalize();
21         }
22     }
23 }
```

kann in C# also auch

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         // Finalizes this instance.
11         ~Foo()
12         {
13             // TODO gr: Clean up any managed and unmanaged
14             //          resources.
15             //          2008-01-02
16         }
17     }
18 }
```

verwendet werden. Obwohl beide Varianten semantisch gleichwertig sind, sollte in der Praxis immer die zweite Variante verwendet werden.

Der einzige Nachteil an Destruktoren in C# ist, dass ihr Ausführungszeitpunkt nicht deterministisch ist. Sie werden dann ausgeführt, wenn die Garbage Collection den Speicher aufräumt und nicht mehr benötigte Objekte entfernt. Da die Ausführung der Garbage Collection nach einem internen Algorithmus von .NET gesteuert wird, kann man sich nicht darauf verlassen, dass ein Objekt zu einem bestimmten Zeitpunkt aufgeräumt und damit sein Finalisierer ausgeführt wird.

Die Garbage Collection kann ein Objekt jedoch nur dann freigeben, wenn sein Finalisierer ausgeführt wurde, weshalb Objekte, die über einen Finalisierer verfügen, länger im Speicher verbleiben als solche, die keinen Finalisierer enthalten. Diese Verzögerung dauert bis zur nächsten Ausführung der Garbage Collection, weshalb nur solche Klassen einen Finalisierer implementieren sollten, die nicht verwaltete Ressourcen wieder freigeben müssen.

Sollen nicht verwaltete Ressourcen zu einem vom Entwickler bestimmten Zeitpunkt oder auch verwaltete Ressourcen freigegeben werden, stellt .NET die Schnittstelle IDisposable zur Verfügung. Eine Klasse, deren Freigabeprozesse gezielt gesteuert werden sollen, muss diese Schnittstelle und die damit einhergehende Methode Dispose implementieren.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo : IDisposable
 9     {
10         /// <summary>
11         /// Disposes this instance.
12         /// </summary>
13         public void Dispose()
14         {
15             // TODO gr: Clean up any unmanaged resources.
16             //          2008-01-02
17
18             // TODO gr: Clean up any managed resources.
19             //          2008-01-02
20         }
21     }
22 }
```

Nun kann die Methode Dispose aufgerufen werden, um die entsprechenden Ressourcen freizugeben. Allerdings kann dieser Aufruf nun wiederum vergessen werden, weshalb der Finalisierer ebenfalls Dispose aufrufen sollte.

C#

```
 1 using System;
 2
```

```
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo : IDisposable
 9     {
10         /// <summary>
11         /// Finalizes this instance.
12         /// </summary>
13         ~Foo()
14         {
15             // Dispose this instance.
16             this.Dispose();
17         }
18 
19         /// <summary>
20         /// Disposes this instance.
21         /// </summary>
22         public void Dispose()
23         {
24             // TODO gr: Clean up any unmanaged resources.
25             //          2008-01-02
26 
27             // TODO gr: Clean up any managed resources.
28             //          2008-01-02
29         }
30     }
31 }
```

Doch auch diese Variante enthält einen Fehler. Wird Dispose vom Entwickler aufgerufen, so wird der Finalisierer dennoch von der Garbage Collection ausgeführt, die ihrerseits Dispose ein zweites Mal aufruft. Das heißt, es wird versucht, Ressourcen freizugeben, die längst nicht mehr belegt sind. Um dies zu verhindern, muss die Dispose-Methode den Finalisierer in der Garbage Collection abmelden, so dass dieser nicht mehr ausgeführt wird.

C#

```
 1 using System;
 2 
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo : IDisposable
 9     {
10         /// <summary>
11         /// Finalizes this instance.
12         /// </summary>
13         ~Foo()
14         {
```

```
15              // Dispose this instance.
16              this.Dispose();
17          }
18
19          /// <summary>
20          /// Disposes this instance.
21          /// </summary>
22          public void Dispose()
23          {
24              // TODO gr: Clean up any unmanaged resources.
25              //          2008-01-02
26
27              // TODO gr: Clean up any managed resources.
28              //          2008-01-02
29
30              // Suppress execution of the finalizer for this
31              // object.
32              GC.SuppressFinalize(this);
33          }
34      }
35  }
```

Da die Garbage Collection alle verwalteten Objekte in einer beliebigen Reihenfolge aufräumt, kann es beim automatischen Aufruf von Dispose durch die Garbage Collection vorkommen, dass einige der verwalteten Ressourcen, die freigegeben werden sollen, bereits nicht mehr existieren. Um dies zu verhindern, wird eine neue Variable eingeführt, mit der überprüft werden kann, ob Dispose vom Entwickler oder von der GarbageCollection aufgerufen wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo : IDisposable
 9     {
10         /// <summary>
11         /// Finalizes this instance.
12         /// </summary>
13         ~Foo()
14         {
15             // Dispose this instance.
16             this.Dispose(false);
17         }
18
19         /// <summary>
20         /// Disposes this instance.
21         /// </summary>
```

```
22          /// <param name="isDisposeByUser"><c>true</c> whether
23          /// disposing is called by the user; <c>false</c>
24          /// otherwise.</param>
25          private void Dispose(bool isDisposeByUser)
26          {
27              // If the disposing is called by the user,
28              // managed resources may be cleaned up, too.
29              if (isDisposeByUser)
30              {
31                  // TODO gr: Clean up any managed resources.
32                  //          2008-01-02
33              }
34
35              // TODO gr: Clean up any unmanaged resources.
36              //          2008-01-02
37
38              // Suppress execution of the finalizer for
39              // this object.
40              GC.SuppressFinalize(this);
41          }
42
43          /// <summary>
44          /// Disposes this instance.
45          /// </summary>
46          public void Dispose()
47          {
48              // Dispose this instance.
49              this.Dispose(true);
50          }
51      }
52  }
```

Es bietet sich an, eine weitere Variable einzuführen, die festlegt, ob Dispose bereits ausgeführt wurde oder nicht, um zu verhindern, dass eine Methode noch nach dem Aufruf von Dispose ausgeführt werden soll. Geschieht dies, kann eine Ausnahme vom Typ ObjectDisposedException ausgelöst werden, der als Parameter der Name des aktuellen Objekts übergeben werden muss.

C#

```
1  using System;
2
3  namespace GoloRoden.GuideToCSharp
4  {
5      /// <summary>
6      /// Represents a foo class.
7      /// </summary>
8      public class Foo : IDisposable
9      {
10         /// <summary>
11         /// Contains, whether this instance has been
12         /// disposed yet.
```

```
13          /// </summary>
14          private bool _isDisposed;
15
16          /// <summary>
17          /// Finalizes this instance.
18          /// </summary>
19          ~Foo()
20          {
21              // Dispose this instance.
22              this.Dispose(false);
23          }
24
25          /// <summary>
26          /// Disposes this instance.
27          /// </summary>
28          /// <param name="isDisposeByUser"><c>true</c> whether
29          /// disposing is called by the user; <c>false</c>
30          /// otherwise.</param>
31          private void Dispose(bool isDisposeByUser)
32          {
33              // If the disposing is called by the user,
34              // managed resources may be cleaned up, too.
35              if (isDisposeByUser)
36              {
37                  // TODO gr: Clean up any managed resources.
38                  //          2008-01-02
39              }
40
41              // TODO gr: Clean up any unmanaged resources.
42              //          2008-01-02
43
44              // Suppress execution of the finalizer for
45              // this object.
46              GC.SuppressFinalize(this);
47
48              // Define this instance as disposed.
49              this._isDisposed = true;
50          }
51
52          /// <summary>
53          /// Dispose this instance.
54          /// </summary>
55          public void Dispose()
56          {
57              // If this instance has been disposed, throw an
58              // exception.
59              if (this._isDisposed)
60              {
61                  throw new ObjectDisposedException(
62                      this.ToString());
63              }
64
65              // Dispose this instance.
66              this.Dispose(true);
```

```
67          }
68      }
69  }
```

Prinzipiell kann eine solche Klasse wie jede andere Klasse verwendet werden, mit dem Unterschied, dass ihre Dispose-Methode aufgerufen werden sollte, sobald die Arbeit mit ihr erledigt ist.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
13          public static void Main()
14          {
15              // Create an instance of the Foo class.
16              Foo foo = new Foo();
17
18              // TODO gr: Use the object.
19              //          2008-01-02
20
21              // Dispose the object.
22              foo.Dispose();
23          }
24      }
25  }
```

Damit dieser Aufruf nicht vergessen wird, bietet C# eine abkürzende Schreibweise mit Hilfe des Schlüsselwortes *using*.

C#

```
 1  using System;
 2
 3  namespace GoloRoden.GuideToCSharp
 4  {
 5      /// <summary>
 6      /// Represents the application class.
 7      /// </summary>
 8      public class Program
 9      {
10          /// <summary>
11          /// Executes the application.
12          /// </summary>
```

```
13          public static void Main()
14          {
15              // Create an instance of the Foo class and
16              // dispose it implicitly.
17              using (Foo foo = new Foo())
18              {
19                  // TODO gr: Use the object.
20                  //          2008-01-02
21              }
22          }
23      }
24  }
```

22.3 Verhalten von Zeichenketten

Neben der Art, wie .NET Speicher verwaltet, gibt es einige weitere Themen, über die ein wenig Hintergrundwissen nicht schadet. Eines dieser Themen ist die Verwaltung von Strings. Strings nehmen in .NET eine Sonderstellung ein, da sie im Speicher nicht veränderbar sind. Wird ein String verändert, wird im Hintergrund eine veränderte Kopie erzeugt, was wiederum Speicher und Zeit kostet.

Aus diesem Grund ist es nicht empfehlenswert, Strings mit Hilfe des Operators + zu verketten. Bei dem Ausdruck

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents the application class.
 7     /// </summary>
 8     public class Program
 9     {
10         /// <summary>
11         /// Executes the application.
12         /// </summary>
13         public static void Main()
14         {
15             // Concatenate some strings.
16             string result =
17                 "Hallo" + " " + "Welt" + "!";
18         }
19     }
20 }
```

werden intern sieben Strings erzeugt – zunächst jeder Teilstring einzeln, dann die Kombination aus den ersten beiden, dann die Kombination aus dieser Kombination und dem dritten, und abschließend die Kombination aller Strings.

Bei einigen wenigen Strings, die miteinander verkettet werden, ist dies noch akzeptabel, ist die Anzahl aber hoch oder geschieht eine solche Verkettung innerhalb einer Schleife, so wird dadurch der Speicherbedarf unnötig in die Höhe getrieben.

Als Alternative gibt es die Klasse StringBuilder aus dem Namensraum System.Text, die einen großen Speicherbereich reserviert, in dem einzelne Strings hintereinander platziert und anschließend auf Anforderung in einen einzigen String zusammengefügt werden.

C#

```
using System;
using System.Text;

namespace GoloRoden.GuideToCSharp
{
    /// <summary>
    /// Represents the application class.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Executes the application.
        /// </summary>
        public static void Main()
        {
            // Create a string builder instance.
            StringBuilder stringBuilder =
                new StringBuilder();

            // Append some strings.
            stringBuilder.Append("Hallo");
            stringBuilder.Append(" ");
            stringBuilder.Append("Welt");
            stringBuilder.Append("!");

            // Get the string from the string builder.
            string result = stringBuilder.ToString();
        }
    }
}
```

Obwohl das Verketten von Strings mit Hilfe der StringBuilder-Klasse deutlich schneller und speicherschonender funktioniert als auf dem klassischen Weg, muss bei ihrem Einsatz bedacht werden, dass auch hier zunächst eine Instanz erzeugt wird und Speicher reserviert werden muss, was ebenfalls Zeit kostet. Je nach Kontext gilt es also abzuwägen, auf welche Art Strings verkettet werden.

22.4 Verspätete Initialisierung

Im Zusammenhang mit statischen Konstruktoren gibt es in C# noch einen wesentlichen Aspekt zu beachten. Zunächst könnte man vermuten, die Ausführung der Klasse

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Contains a bar field.
12         /// </summary>
13         private static int _bar = 23;
14     }
15 }
```

würde analog zur Ausführung der Klasse

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Contains a bar field.
12         /// </summary>
13         private static int _bar;
14
15         /// <summary>
16         /// Initializes the Foo type.
17         /// </summary>
18         static Foo()
19         {
20             // Set the class's fields.
21             _bar = 23;
22         }
23     }
24 }
```

stattfinden. Es gibt allerdings einen Unterschied, der sich darin bemerkbar macht, wann die Zuweisung des Wertes an die Variable stattfindet. Während der Wert in der ersten Variante irgendwann zwischen dem Start der Anwendung und dem ersten Zugriff auf den Typ stattfindet, geschieht dies bei der zweiten Variante auf jeden Fall erst beim Zugriff auf den Typ.

Es wäre sogar ausreichend, einen vollständig leeren statischen Konstruktur bereitzustellen, der Effekt wäre der gleiche: Sobald ein statischer Konstruktor vorhanden ist, wird ein Typ erst initialisiert, wenn er tatsächlich verwendet wird.

C#

```
 1 using System;
 2
 3 namespace GoloRoden.GuideToCSharp
 4 {
 5     /// <summary>
 6     /// Represents a foo class.
 7     /// </summary>
 8     public class Foo
 9     {
10         /// <summary>
11         /// Contains a bar field.
12         /// </summary>
13         private static int _bar = 23;
14
15         /// <summary>
16         /// Initializes the Foo type.
17         /// </summary>
18         static Foo()
19         {
20         }
21     }
22 }
```

Dies liegt daran, dass der Compiler jeden Typ mit dem internen Flag beforefieldinit kennzeichnet, der nicht über einen statischen Konstruktor verfügt. Dieses Flag bewirkt, dass der Typ irgendwann vor, spätestens aber beim ersten Zugriff initialisiert wird.

Ausnutzen lässt sich dieses Verhalten, wenn ein Typ nicht in jedem Fall in einer Anwendung benötigt wird, seine Erzeugung aber relativ aufwändig ist, weil beispielsweise auf zahlreiche externe Ressourcen zugegriffen werden muss. In einem solchen Fall kann die Initialisierung durch das Hinzufügen eines statischen Konstruktors verzögert werden, bis der Typ tatsächlich benötigt wird.

Sachverzeichnis