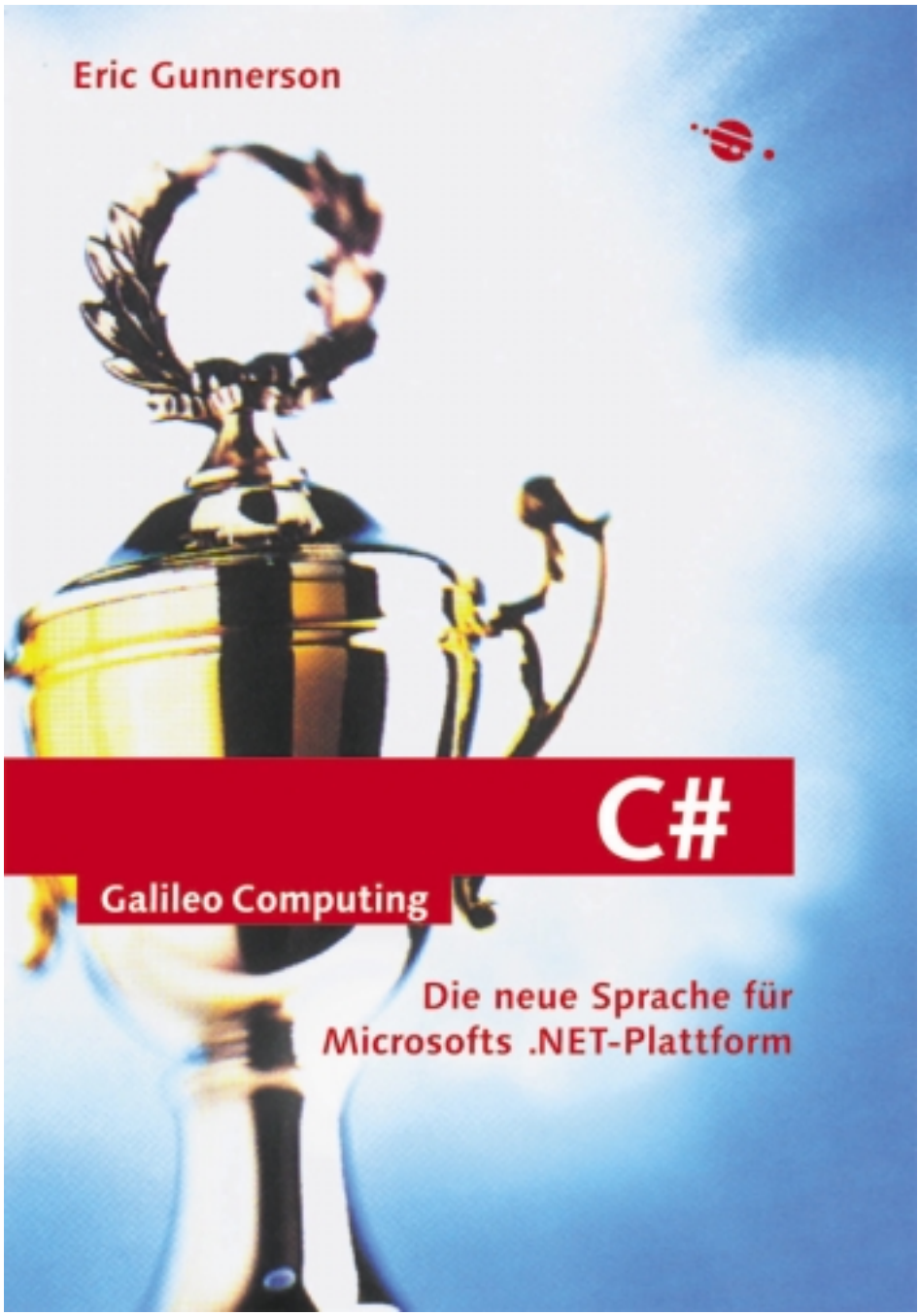
Galileo Computing



http://www.galileocomputing.de/

Liebe Leserin, lieber Leser,

wir freuen uns, dass Sie sich für ein Buch der Reihe Galileo Computing entschieden haben.

Galileo Computing bietet Titel zu allen wichtigen Sprachen, Tools und Techniken der Programmierung. Die Bücher der Reihe zeigen, wie es wirklich geht und warum – immer mit Blick auf die praktische Anwendung. Galileo Computing ist Fachliteratur für Experten von Experten. Kompakt in der Darstellung und benutzerfreundlich gestaltet. Wo wir Übersetzungen in unser Programm aufnehmen, garantieren wir deren herausragenden Informationswert. Hohe Qualität der Übertragung ist selbstverständlich.

Jedes unserer Bücher will Sie überzeugen. Damit uns das immer wieder neu gelingt, sind wir auf Ihre Rückmeldung angewiesen. Bitte teilen Sie uns Ihre Meinung zu diesem Buch mit. Ihre kritischen und freundlichen Anregungen, Ihre Wünsche und Ideen werden uns weiterhelfen.

Wir freuen uns auf den Dialog mit Ihnen.

Ihre Judith Stevens

Galileo Press
Gartenstr. 24
53229 Bonn

judith.stevens@galileo-press.de
www.galileocomputing.de

Eric Gunnerson

C#

Die neue Sprache für
Microsofts .NET-Plattform

Galileo Computing

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
der Deutschen Bibliothek erhältlich

ISBN 3-89842-107-4

1. Auflage 2000

Die amerikanische Originalausgabe trägt den
Titel: A Programmer's Introduction to C#
(ISBN 1-893115-86-0, Apess™)

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch **Eppur se muove** (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Übersetzung Simone Matull, Lemoine International GmbH, Köln et al., **Lektorat** Judith Stevens **Korrektorat** Lisa Alexin, Claudia Falk, Holger Schmidt, Bonn **Einbandgestaltung** Barbara Thoben, Köln **Titelfoto** zefa visual media gmbh **Herstellung** Iris Warkus **Satz** reemers publishing services gmbh, Krefeld **Druck und Bindung** Media-Print, Paderborn

Inhalt

Für Tony Jongejan, der mich an die Programmierung herangeführt hat und der seiner Zeit voraus ist

Danksagung

Obwohl das Schreiben eines Buches häufig ein einsames Unterfangen darstellt, kommt dennoch kein Autor ohne Hilfe aus.

Ich möchte all denen danken, die mich bei diesem Buch unterstützt haben, einschließlich aller Teammitglieder, die mir meine unzähligen Fragen beantwortet und meine unfertigen Entwürfe gelesen haben. Darüber hinaus möchte ich meinen Managern und Microsoft dafür danken, dass ich an so einem einmaligen Projekt arbeiten und darüber auch noch ein Buch schreiben durfte.

Danke auch an das Apress-Team für das mir entgegengebrachte Vertrauen und die unendliche Geduld.

Bedanken möchte ich mich auch bei allen Künstlern, zu deren Musik man Bücher schreiben kann – die CDs wurden natürlich alle rechtmäßig erstanden –, vor allem jedoch der Gruppe Rush für all ihre Songs.

Abschließend möchte ich denen danken, die mir zu Hause Unterstützung gaben: meiner Frau Kim und meiner Tochter Samantha, die sich nicht darüber beklagten, dass ich viel arbeitete – selbst dann nicht, wenn ich dies im Urlaub tat; und ein besonderer Dank an meine Katze dafür, dass sie mir beim Schreiben die Arme heruntergedrückt hat.

Eric Gunnerson, im September 2000

Über dieses Buch

C# ist eines der aufregendsten Projekte, an denen ich je mitgearbeitet habe. Es gibt unzählige Sprachen mit unterschiedlichen Stärken und Schwächen, aber nur selten wird eine neue Sprache entwickelt, die sich optimal in Hardware, Software und spezifische Programmieransätze einfügt. Ich glaube, dass C# solch eine Sprache ist. Natürlich stellt die Auswahl der Programmiersprache oft eine Glaubensfrage (»religious issue«) dar[1].

Ich habe dieses Buch als eine Art Einführung in die Sprache C# gestaltet, da ich denke, dass dies die beste und interessanteste Möglichkeit zum Erlernen einer Sprache darstellt. Unglücklicherweise können solche Einführungen manchmal langweilig und lang werden, besonders dann, wenn einem die Materie bekannt ist. Der Schwerpunkt wird vielleicht auf Dinge gelegt, die Sie nicht interessieren, während andere Aspekte nicht berücksichtigt werden. Daher ist es schön, wenn man den langweiligen Part umgehen und direkt zum interessanten Teil übergehen kann. Hierzu kann ich Ihnen zwei Ansätze anbieten:

Wenn Sie sofort einsteigen möchten, lesen Sie Kapitel 3, Schnelleinstieg in C#. Dieses Kapitel bietet einen kurzen Überblick über C# und bietet genügend Informationen, um direkt mit der Codierung zu beginnen.

Kapitel 34, C# im Vergleich zu anderen Sprachen, enthält sprachspezifische Vergleiche mit C++, Visual Basic und Java und richtet sich an Programmierer, die hauptsächlich mit einer bestimmten Sprache arbeiten bzw. an alle, die gerne Vergleiche anstellen.

Es ist nun Anfang August 2000 und die Beta-Version von Visual Studio, die C# enthalten wird, ist fast fertig. Die Sprachsyntax steht mehr oder weniger, aber einige Ecken und Kanten müssen sicherlich noch abgeschliffen werden. In Kapitel 35, C# und die Zukunft, finden Sie einige Informationen dazu, was zukünftige Versionen von C# bereithalten sollen.

Wenn Sie Kommentare zu diesem Buch haben, können Sie mich unter gunnerso@halcyon.com erreichen. Der gesamte Quellcode kann von der Apress-Website unter http://www.apress.com heruntergeladen werden.

1. Eine gute Definition von »religious issue« finden Sie in der Jargondatei unter http://www.jargonfile.org.

Vorwort

Bei der Entwicklung einer neuen Programmiersprache stellt sich zunächst immer eine Frage: Warum? Bei der Entwicklung von C# hatten wir verschiedene Ziele im Auge:

- **Die Entwicklung der ersten komponentenorientierten Sprache für die C/C++-Familie**
  Die Softwareentwicklung stützt sich immer weniger auf das Entwickeln monolithischer Anwendungen als auf das Entwerfen von Komponenten, die sich in die verschiedenen Ausführungsumgebungen einfügen; beispielsweise ein Steuerelement in einem Browser oder ein Geschäftsobjekt, das in ASP+ ausgeführt wird. Der Schlüssel zu solchen Komponenten sind Eigenschaften, Methoden und Ereignisse sowie die Attribute, mit denen beschreibende Informationen zu den Komponenten bereitgestellt werden. All diese Konzepte werden in C# berücksichtigt und machen C# zu einer sehr natürlichen Sprache, in der Komponenten erstellt und verwendet werden können.
- **Die Entwicklung einer Sprache, bei der alle Elemente wirkliche Objekte sind**
  Durch die innovative Umsetzung von Konzepten wie dem Boxing und Unboxing schließt C# die Lücke zwischen einfachen Typen und Klassen und ermöglicht den Einsatz jeglicher Daten als Objekte. Darüber hinaus wird mit C# das Konzept von Wertetypen eingeführt, mit dem der Benutzer kleine schlanke Objekte implementieren kann, die keine Heapzuweisung erfordern.
- **Die Möglichkeit zur Entwicklung robuster und langlebiger Software**
  C# schließt eine Speicherbereinigung, eine strukturierte Ausnahmebehandlung und die Typensicherheit ein. Durch die Anwendung dieser Konzepte werden ganze Kategorien von Fehlern beseitigt, die häufig in C++-Programmen auftreten.
- **Die Vereinfachung von C++ unter Ausnutzung des vorhandenen Potenzials der Programmierer**
  C# weist eine große Ähnlichkeit zu C++ auf, wodurch sich C++-Programmierer in C# sehr schnell wohl fühlen können. Des Weiteren bietet C# umfangreiche Interoperabilität mit COM und DLLs, d.h., bereits vorhandener Code kann problemlos integriert werden.

Wir haben hart daran gearbeitet, diese Ziele zu erreichen. Ein Großteil dieser Arbeit fand im C#-Entwurfsteam statt, das sich über einen Zeitraum von zwei Jahren regelmäßig traf. Als Kopf des Teams für die Qualitätssicherung war Eric einer der

wichtigsten Mitarbeiter und ist daher besonders geeignet, nicht nur die Funktionsweise von C# zu erläutern, sondern auch deren Hintergründe zu beleuchten. Dies wird Ihnen im Verlauf der Lektüre dieses Buches klar werden.

Ich hoffe, dass Sie an der Verwendung von C# genauso viel Spaß haben wie jeder Einzelne im C#-Entwurfsteam daran, C# zu entwickeln.

Anders Hejlsberg

Distinguished Engineer
Microsoft Corporation

1 Einleitung

1.1 Wozu eine weitere Sprache?

Sie fragen sich bestimmt auch, »Warum *noch eine* Sprache? Warum nicht weiterhin C++ nutzen?« (oder VB oder Java oder welche Sprache auch immer Sie bevorzugen). Zumindest werden Sie sich das gefragt haben, bevor Sie das Buch kauften.

Programmiersprachen können auf gewisse Weise mit Elektrowerkzeugen verglichen werden. Jedes Werkzeug ist für einen bestimmten Zweck besonders geeignet. Ich *könnte* beispielsweise ein Brett mit Hilfe einer Fräse kürzen, es wäre jedoch sehr viel einfacher, eine Säge zu benutzen. Genauso könnte ich mit Hilfe einer Programmiersprache wie LISP ein Computerspiel mit aufwendiger Grafikanimation schreiben, mit C++ wäre es aber wahrscheinlich einfacher.

C# (ausgesprochen »C sharp«) ist die systemeigene Sprache für die .NET Common Language Runtime. C# wurde entworfen, um sich nahtlos in die .NET-Laufzeitumgebung einzufügen. Sie können (und sollten dies gelegentlich auch) Code entweder in Visual C++ oder Visual Basic schreiben, in den meisten Fällen ist C# jedoch geeigneter. Da die Common Language Runtime einen Kernpunkt für viele Funktionen von C# darstellt, wird diese Laufzeitumgebung in Kapitel 2, Die .NET-Laufzeitumgebung, näher vorgestellt – vor allem diejenigen Elemente, die für die Sprache C# von besonderer Bedeutung sind.

1.2 Die Entwicklungsziele bei C#

Die erste Veröffentlichung von C++ sorgte für einige Aufregung. Hier wurde eine Programmiersprache zum Erstellen objektorientierter Software angeboten, bei der C-Programmierer ihr bereits vorhandenes Wissen und ihre Software weiter nutzen und ausbauen konnten. C++ ist nicht so objektorientiert wie beispielsweise Eiffel, stellte aber dennoch genügend objektorientierte Funktionen bereit, um große Vorteile zu bieten.

Mit C# verhält es sich ähnlich. In Kombination mit der .NET Common Language Runtime kann diese Sprache für die Entwicklung komponentenorientierter Software eingesetzt werden, ohne dass die Programmierer ihr Potenzial im Hinblick auf C-, C++- oder COM-Code ungenutzt lassen müssen.

C# wurde für die Entwicklung robuster, langlebiger Komponenten konzipiert, mit denen Situationen des wirklichen Lebens bewältigt werden können.

1.2.1 Komponentensoftware

Die .NET Common Language Runtime ist eine komponentenbasierte Umgebung, daher ist es wohl keine Überraschung, dass C# die Komponentenerstellung vereinfacht. Es ist eine Programmiersprache, die sich auf Komponenten konzentriert, d.h., alle Objekte werden als Komponenten geschrieben, und die Komponenten bilden den Mittelpunkt aller Geschehnisse.

Die komponentenbezogenen Konzepte, beispielsweise Eigenschaften, Methoden und Ereignisse, stellen die Kernelemente der Sprache und der zugrunde liegenden Laufzeitumgebung dar. Deklarative Informationen (die Attribute) können auf Komponenten angewendet werden, um anderen Bestandteilen des Systems Entwurfszeit- und Laufzeitinformationen zu einer Komponente zu liefern. Die Dokumentation kann innerhalb der Komponente geschrieben und in XML exportiert werden.

C#-Objekte erfordern weder Header- oder IDL-Dateien noch Typbibliotheken. Die in C# erstellten Komponenten sind vollständig selbstbeschreibend und können ohne Registrierung eingesetzt werden.

C# wird bei der Komponentenerstellung von der .NET-Laufzeitumgebung und dem .NET-Framework unterstützt, die zusammen ein einheitliches Typensystem bereitstellen, bei dem jedes Element als Objekt behandelt werden kann, ohne dass – wie bei rein objektorientierten Systemen wie z.B. Smalltalk – Leistungseinbußen hingenommen werden müssen.

1.2.2 Robuste und langlebige Software

In der komponentenbasierten Welt ist die Entwicklung von robuster und langlebiger Software ein sehr wichtiger Aspekt. Webserver können monatelang ohne einen außerplanmäßigen Neustart ausgeführt werden, und jeder weiß, dass nicht geplante Neustarts nichts Wünschenswertes sind.

Mit der Speicherbereinigung wird dem Programmierer die Last der Speicherverwaltung abgenommen[1]. Dem Problem der Versionssteuerung für Komponenten wird durch die Definition einer entsprechenden Semantik und die Möglichkeit zur Trennung von Schnittstelle und Implementierung begegnet. Numerische Operationen können auf einen möglichen Überlauf geprüft werden und Arrays unterstützen die Bereichsprüfung.

1. Die Speicherverwaltung von C++ ist nicht wirklich schwierig, in Bezug auf die Komponenten kann es jedoch einige knifflige Situationen geben. Das Hauptproblem bei der Speicherverwaltung liegt darin, dass Zeit und Mühe investiert werden müssen. Mit der Speicherbereinigung ist es nicht länger erforderlich, Zeit auf die Codierung und Prüfung zu verwenden, um Speicherlecks zu vermeiden. Auf diese Weise kann sich der Programmierer auf die Programmlogik konzentrieren.

C# bietet eine einfache, sichere und intuitive Umgebung. Die Fehlerbehandlung wird hierbei ebenfalls berücksichtigt: Eine Ausnahmebehandlung ist in der gesamten Umgebung verfügbar. Die Sprache ist typensicher und bietet dem Programmierer Schutz vor nicht initialisierten Variablen, unsicheren Typumwandlungen und weiteren häufig auftretenden Programmierfehlern.

1.2.3 Software für das wirkliche Leben

Softwareentwicklung ist keine einfache Sache. Die Software kann in den seltensten Fällen von Grund auf neu entwickelt werden. Worauf es ankommt, sind eine annehmbare Leistung, die Integration mit vorhandenem Code sowie ein gutes Kosten-Nutzen-Verhältnis. Eine gut entworfene Umgebung nützt nichts, wenn sie im wirklichen Leben nicht standhalten kann.

C# weist die Vorteile einer eleganten und einheitlichen Umgebung auf und bietet dennoch Zugriff auf weniger »anständige« Funktionen (beispielsweise Zeiger), wenn diese benötigt werden.

Mit C# gehen die Investitionen in vorhandenen Code nicht einfach verloren. Vorhandene COM-Objekte können wie .NET-Objekte eingesetzt werden[2]. Mit der .NET Common Language Runtime erscheinen dem vorhandenen COM-basierten Code die Objekte in der Laufzeitumgebung wie COM-Objekte. Systemeigener C-Code in DLL-Dateien kann über den C#-Code aufgerufen werden[3].

C# bietet Lowlevelzugriff, falls erforderlich. Kleine schlanke Objekte können einem Stack zugewiesen werden und dennoch Bestandteil der einheitlichen Umgebung sein. Lowlevelzugriff wird über den so genannten **unsafe**-Modus gewährt, bei dem der Einsatz von Zeigern möglich ist, wenn die Leistung eine besonders große Rolle spielt oder Zeiger für vorhandene DLLs benötigt werden.

C# baut auf C++ auf, daher wird sich jeder C++-Programmierer schnell in dieser Sprache wohl fühlen. C# ist schnell erlernbar und führt zu höherer Produktivität ohne Leistungseinbußen.

Darüber hinaus setzt C# auf die Stärken der .NET Common Language Runtime, die umfangreiche Bibliotheksunterstützung für allgemeine Programmieraufgaben und anwendungsspezifische Aufgaben bietet. Die .NET-Laufzeitumgebung, die Frameworks und die verschiedenen Programmiersprachen werden in der Visual Studio-Umgebung zu einem Universalpaket für den .NET-Programmierer zusammengefasst.

2. Normalerweise. In einigen Fällen ist dies in der Praxis etwas schwieriger umzusetzen.
3. Visual C++ wurde mit so genannten »verwalteten Erweiterungen« ausgestattet, die das Erstellen von .NET-Komponenten ermöglichen. Weitere Informationen zu diesen Erweiterungen finden Sie in der Microsoft-Website.

2 Grundlagen der objektorientierten Programmierung

In diesem Kapitel erhalten Sie eine Einführung in die objektorientierte Programmierung. Diejenigen von Ihnen, die bereits mit der objektorientierten Programmierung vertraut sind, können diesen Abschnitt überspringen.

Beim objektorientierten Entwurf gibt es verschiedenste Ansätze, was durch die Anzahl der zu diesem Thema veröffentlichten Bücher belegt wird. Die folgende Einleitung geht von einem recht pragmatischen Ansatz aus und beschäftigt sich weniger mit dem Design, auch wenn genau diese entwurfsbezogenen Ansätze für Anfänger recht nützlich sein können.

2.1 Was ist ein Objekt?

Ein Objekt ist eine Sammlung zueinander in Beziehung stehender Informationen und Funktionen. Ein Objekt kann etwas sein, das über ein entsprechendes Äquivalent in der tatsächlichen Welt verfügt (z. B. ein **Mitarbeiter**-Objekt), etwas, das virtuelle Bedeutung hat (beispielsweise ein Fenster auf dem Bildschirm), oder es kann einfach ein abstraktes Element innerhalb eines Programms darstellen (etwa eine Liste der zu erledigenden Aufgaben).

Ein Objekt setzt sich aus den Daten zusammen, die das Objekt selbst und die Operationen beschreiben, die für das Objekt ausgeführt werden können. Die in einem **Mitarbeiter**-Objekt gespeicherten Informationen beispielsweise können Informationen zur Person (Name, Adresse), arbeitsbezogene Informationen (Position, Gehalt) usw. enthalten. Zu den ausgeführten Operationen gehört vielleicht das Erstellen einer Gehaltsabrechnung oder die Beförderung des Mitarbeiters.

Im objektorientierten Entwurf besteht der erste Schritt darin, die Bedeutung der Objekte zu definieren. Bei der Verwendung von Objekten, die auch im wirklichen Leben vorkommen, ist dies einfach, wenn Sie jedoch in der virtuellen Welt arbeiten, verschwimmen die Grenzen. Hier zeigt sich die Kunst eines guten Designs, und dies ist auch der Grund dafür, weshalb eine gute Architektur gefordert ist.

2.2 Vererbung

Die Vererbung ist ein fundamentales Leistungsmerkmal eines objektorientierten Systems. Sie stellt die Fähigkeit dar, Daten und Funktionen von einem übergeordneten Objekt an ein untergeordnetes weiterzugeben, also zu vererben. Statt Objekte neu zu entwickeln, kann neuer Code auf der Arbeit anderer Programmierer

basieren. Es werden lediglich einige neue Funktionen hinzugefügt. Das übergeordnete Objekt, auf dem der neue Code beruht, wird als *Basisklasse* bezeichnet, das untergeordnete Objekt als *abgeleitete Klasse*.

Der Vererbung kommt bei der Erläuterung des objektorientierten Designs große Bedeutung zu, tatsächlich verwendet wird die Vererbung jedoch seltener. Hierfür gibt es verschiedene Gründe.

Zunächst ist die Vererbung ein Beispiel für die im objektorientierten Design angeführte »Ist-Ein(e)«-Beziehung (engl. »Is-A«). Wenn ein System über ein **Tier**-Objekt und ein **Katze**-Objekt verfügt, kann das **Katze**-Objekt vom **Tier**-Objekt erben, denn eine **Katze** »Ist-Ein(e)« **Tier**. Bei der Vererbung ist die Basisklasse stets allgemeiner gefasst als die abgeleitete Klasse. Die **Katze**-Klasse würde die **Essen**-Funktion der **Tier**-Klasse erben und über eine erweiterte **Schlafen**-Funktion verfügen. Im wirklichen Leben sind derartige Beziehungen jedoch weniger gebräuchlich.

Als Zweites muss zur Verwendung der Vererbung die Basisklasse mit dem Hintergedanken der Vererbung entworfen werden. Dies ist aus verschiedenen Gründen wichtig. Wenn die Objekte keine geeignete Struktur aufweisen, kann die Vererbung nicht richtig funktionieren. Noch wichtiger: Ein Design, das die Vererbung verwendet, macht deutlich, dass der Autor der Klasse damit einverstanden ist, dass andere Klassen von dieser Klasse erben. Wenn eine neue Klasse von einer bestehenden abgeleitet wird, bei der dies nicht der Fall ist, kann dies zu einer Änderung der Basisklasse und damit zur Zerstörung der abgeleiteten Klasse führen.

Einige weniger erfahrene Programmierer gehen fälschlicherweise davon aus, dass die Vererbung in der objektorientierten Programmierung breite Anwendung finden *sollte* und setzen sie daher zu häufig ein. Die Vererbung sollte nur dann verwendet werden, wenn die sich ergebenden Vorteile tatsächlich genutzt werden[1]. Siehe hierzu auch den Abschnitt »Polymorphismus und virtuelle Funktionen«.

In der .NET Common Language Runtime werden alle Objekte von einer Basisklasse namens **object** abgeleitet, und hierbei wird nur die Einfachvererbung unterstützt (d.h., ein Objekt kann nur von einer Basisklasse abgeleitet werden). Auf diese Weise wird die Verwendung einiger gebräuchlicher Wendungen verhindert, die in Mehrfachvererbungssystemen wie C++ genutzt werden. Gleichzeitig wird jedoch der Missbrauch der Mehrfachvererbung verhindert und eine Vereinfachung erreicht. In den meisten Fällen stellt dies einen guten Tausch dar. Die .NET-Laufzeitumgebung ermöglicht eine Mehrfachvererbung in Form von Schnittstellen ohne Implementierung. Die Schnittstellen werden in Kapitel 10, Schnittstellen, behandelt.

1. Vielleicht sollte eine Studie wie »Mehrfachvererbung als schädlich eingestuft« veröffentlicht werden. Naja, wahrscheinlich gibt es sie schon irgendwo.

2.3 Das Prinzip des Containments

Wenn also Vererbung nicht die richtige Wahl ist, was dann?

Die Antwort lautet: Containment, auch Aggregation oder Enthaltenseinbeziehung genannt. Hierbei wird ein Objekt nicht als ein Beispiel eines anderen Objekts betrachtet, sondern als eine Instanz eines Objekts, die sich im Objekt befindet. Statt also über eine Klasse zu verfügen, die wie eine Zeichenfolge aussieht, enthält die Klasse eine Zeichenfolge (oder ein Array oder eine Hashtabelle).

Üblicherweise sollte beim Design der Ansatz des Containments gewählt werden, auf die Vererbung sollte nur zurückgegriffen werden, wenn dies erforderlich ist (z. B. wenn tatsächlich eine »Ist-Ein(e)«-Beziehung vorliegt).

2.4 Polymorphismus und virtuelle Funktionen

Vor einiger Zeit habe ich ein Musiksystem geschrieben, und ich wollte hierbei sowohl WinAmp als auch den Windows Media Player für die Wiedergabe unterstützen. Gleichzeitig sollte nicht jeder Codebestandteil wissen müssen, welches Programm verwendet wird. Ich definierte daher eine abstrakte Klasse, d. h. eine Klasse, mit der die Funktionen beschrieben werden, die eine abgeleitete Klasse implementieren muss. Auf diese Weise werden gelegentlich Funktionen bereitgestellt, die für beide Klassen nützlich sind.

In diesem Fall hieß die abstrakte Klasse **MusicServer** und verfügte über Funktionen wie **Play()**, **NextSong()**, **Pause()** usw. Jede dieser Funktionen wurde als abstrakt deklariert, damit sie durch die Playerklasse selbst implementiert würde.

Abstrakte Funktionen sind automatisch virtuelle Funktionen, die dem Programmierer die Verwendung des Polymorphismus zur Codevereinfachung ermöglichen. Ist eine virtuelle Funktion vorhanden, kann der Programmierer einen Verweis auf die abstrakte statt auf die abgeleitete Klasse erstellen, und der Compiler schreibt Code zum Aufrufen der geeigneten Funktionsversion zur Laufzeit.

Ich möchte dies durch ein Beispiel verdeutlichen. Das Musiksystem unterstützt für die Wiedergabe sowohl WinAmp als auch den Windows Media Player. Der nachstehende Code gibt einen kurzen Überblick über das Aussehen der Klassen:

```
using System;
public abstract class MusicServer
{
   public abstract void Play();
}
public class WinAmpServer: MusicServer
{
```

```
    public override void Play()
    {
        Console.WriteLine("WinAmpServer.Play()");
    }
}
public class MediaServer: MusicServer
{
    public override void Play()
    {
        Console.WriteLine("MediaServer.Play()");
    }
}
class Test
{
    public static void CallPlay(MusicServer ms)
    {
        ms.Play();
    }
    public static void Main()
    {
        MusicServer ms = new WinAmpServer();
        CallPlay(ms);
        ms = new MediaServer();
        CallPlay(ms);
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
WinAmpServer.Play()
MediaServer.Play()
```

Polymorphismus und virtuelle Funktionen werden in der .NET-Laufzeitumgebung an vielen Stellen eingesetzt. Das Basisobjekt **object** beispielsweise verfügt über die virtuelle Funktion **ToString()**, die zum Konvertieren eines Objekts in eine Zeichenfolgendarstellung des Objekts verwendet wird. Wenn Sie die Funktion **ToString()** für ein Objekt aufrufen, das nicht über eine eigene Version von **ToString()** verfügt, wird die Version von **ToString()** aufgerufen, die Teil der Klasse **object** ist[2], wodurch einfach der Name der Klasse zurückgegeben wird. Wenn Sie die **ToString()**-Funktion überladen (eine eigene Version schreiben), wird statt des-

2. Falls eine Basisklasse des aktuellen Objekts vorhanden ist und diese **ToString()** definiert, wird diese Version aufgerufen.

sen diese Version aufgerufen, und Sie können eine sinnvollere Operation ausführen. So könnten Sie beispielsweise den Namen des Mitarbeiters ausschreiben, der im **Mitarbeiter**-Objekt enthalten ist. Im Musiksystem bedeutete dies, die Funktionen **Play()**, **Pause()**, **NextSong()** usw. zu überladen.

2.5 Kapselung und Sichtbarkeit

Beim Entwurf von Objekten muss der Programmierer entscheiden, in welchem Ausmaß das Objekt für den Benutzer sichtbar ist bzw. dem Benutzer verborgen bleibt. Details, die für den Benutzer nicht sichtbar sind, bezeichnet man als in der Klasse gekapselt.

Im Allgemeinen besteht das Ziel beim Objektentwurf darin, die Klasse in größtmöglichem Umfang zu kapseln. Die Gründe hierfür lauten:

1. Der Benutzer kann die als privat deklarierten Elemente im Objekt nicht ändern, d.h. das Risiko, dass der Benutzer diese Details im Code ändert oder von diesen abhängig ist, wird verringert. Ist der Benutzer von diesen Details abhängig, können Objektänderungen zur Beschädigung des Benutzercodes führen.
2. Änderungen, die an den öffentlichen Bestandteilen eines Objekts vorgenommen werden, müssen eine weitere Kompatibilität mit der vorherigen Version sicherstellen. Je mehr Einsicht der Benutzer erhält, desto weniger Codeelemente können geändert werden, ohne den Benutzercode zu zerstören.
3. Größere Schnittstellen erhöhen die Komplexität des gesamten Systems. Auf private Felder kann nur von einer Klasse aus, auf öffentliche Felder kann über eine beliebige Instanz der Klasse zugegriffen werden. Der erforderliche Aufwand für die Fehlersuche steigt mit der Anzahl der öffentlichen Felder.

Dieses Thema wird in Kapitel 5, 101-Klassen, weiter ausgeführt.

3 Die .NET-Laufzeitumgebung

In der Vergangenheit war es schwierig, Module zu schreiben, die aus mehreren Sprachen aufgerufen werden konnten. Code, der in Visual Basic geschrieben wurde, kann nicht aus Visual C++ aufgerufen werden. Visual C++-Code kann unter Umständen aus Visual Basic aufgerufen werden, aber dies ist nicht einfach. Visual C++ verwendet die C- und C++-Laufzeitumgebungen, die ein besonderes Verhalten aufweisen, und Visual Basic verwendet eine eigene Ausführungsengine, die ebenfalls über ein eigenes und anderes Verhalten verfügt.

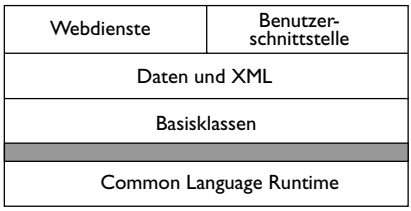
Daher wurde COM entwickelt und hat sich als nützliche Methode zum Schreiben von komponentenbasierter Software herausgestellt. Unglücklicherweise ist es nicht leicht, COM in der Visual C++-Umgebung einzusetzen, außerdem wird COM von Visual Basic nicht vollständig unterstützt. Aus diesem Grund wurde COM häufig zum Schreiben von COM-Komponenten eingesetzt, zur Entwicklung von systemeigenen Anwendungen dagegen weniger. Wenn also ein Programmierer brauchbaren C++-Code schrieb und ein anderer Programmierer dies in Visual Basic tat, gab es nicht wirklich die Möglichkeit zur Zusammenarbeit.

Darüber hinaus bestanden Schwierigkeiten bei den Bibliotheksprovidern, denn es gab nicht einen, der in allen Umgebungen eingesetzt werden konnte. Wenn die Bibliothek auf Visual Basic ausgerichtet war, war die Verwendung in Visual Basic einfach, der Zugriff von C++ war jedoch möglicherweise eingeschränkt oder führte zu einer inakzeptablen Leistungseinbuße. Oder eine Bibliothek wurde für C++-Benutzer geschrieben und bot diesen gute Leistungsergebnisse und Lowlevelzugriff, für Visual Basic-Programmierer war sie jedoch nicht zugänglich.

Einige Bibliotheken wurden für beide Benutzertypen geschrieben, aber dies bedeutete üblicherweise auch, Kompromisse einzugehen. Beim Versenden einer E-Mail auf einem Windows-System hat der Programmierer die Auswahl zwischen CDO (Collaboration Data Objects), einer COM-basierten Schnittstelle, die von beiden Sprachen aus aufgerufen werden kann, jedoch nicht sämtliche Funktionen bereitstellt[1], und systemeigenen MAPI-Funktionen (sowohl in den C- als auch in den C++-Versionen), mit denen auf alle Funktionen zugegriffen werden kann.

Die .NET-Laufzeitumgebung wurde entwickelt, um hier Abhilfe zu schaffen. Es gibt eine Möglichkeit, Code (Metadaten), eine Laufzeitumgebung und eine Bibliothek (die Common Language Runtime und Frameworks) zu beschreiben. Das folgende Diagramm zeigt den Aufbau der .NET-Laufzeitumgebung.

1. Dies rührt wahrscheinlich daher, dass es schwierig ist, das interne Lowleveldesign in etwas zu übersetzen, das von einer Automatisierungsschnittstelle aufgerufen werden kann.



Die Common Language Runtime stellt die grundlegenden Ausführungsdienste zur Verfügung. Die darüber angeordneten Basisklassen stellen maßgebliche Datentypen, Auflistungsklassen und allgemeine Klassen bereit. Oberhalb der Basisklassen befinden sich die Klassen zur Handhabung von Daten und XML. Ganz oben in der Architektur befinden sich die Klassen zur Offenlegung der Webdienste[2] sowie die Klassen zur Handhabung der Benutzerschnittstelle. Eine Anwendung kann von allen diesen Ebenen aus aufgerufen werden sowie die Klassen sämtlicher Ebenen verwenden.

Zum Verständnis der Funktionsweise von C# sind Kenntnisse über die .NET-Laufzeitumgebung und deren Frameworks erforderlich. Die folgenden Abschnitte enthalten einen Überblick über dieses Thema, detailliertere Informationen finden Sie in Kapitel 31, C# im Detail.

3.1 Die Ausführungsumgebung

Dieser Abschnitt war ursprünglich mit »Die Ausführungsengine« betitelt, aber die .NET-Laufzeitumgebung ist viel mehr als nur eine Engine. Die Umgebung bietet ein einfacheres Programmiermodell, Schutz und Sicherheit, Unterstützung für leistungsstarke Tools sowie Methoden zum Bereitstellen, Packen usw.

3.1.1 Ein einfacheres Programmiermodell

Alle Dienste werden über ein gemeinsames Modell bereitgestellt, auf das von allen .NET-Sprachen aus gleichberechtigt zugegriffen werden kann. Alle Dienste können in einer beliebigen .NET-Sprache geschrieben werden[3]. Die Umgebung ist in hohem Maße spracherkennend und ermöglicht daher eine Sprachauswahl. Dies macht eine Codewiederverwendung einfacher, sowohl für den Programmierer als auch für den Bibliotheksprovider.

2. Eine Möglichkeit zur Offenlegung einer programmatischen Schnittstelle über einen Webserver.
3. Einige Sprachen sind möglicherweise nicht mit den systemeigenen Plattformfunktionen kompatibel.

Die Umgebung unterstützt außerdem die Verwendung von vorhandenem C#-Code, entweder über das Aufrufen von Funktionen in DLLs oder dadurch, dass COM-Komponenten wie .NET-Laufzeitkomponenten eingesetzt werden. .NET-Laufzeitkomponenten können auch in Situationen verwendet werden, in denen COM-Komponenten benötigt werden.

Im Gegensatz zu den verschiedenen Fehlerbehandlungsmethoden vorhandener Bibliotheken werden in der .NET-Laufzeitumgebung alle Fehler über Ausnahmen ermittelt. Es besteht also keine Veranlassung, zwischen Fehlercodes, HRESULTs und Ausnahmen zu wechseln.

Abschließend enthält die Umgebung BCLs (Base Class Libraries), über die Funktionen bereitgestellt werden, die traditionell in Laufzeitbibliotheken vorhanden sind, plus einige neue Funktionen. Zu den BCL-Funktionen zählen:

- Auflistungsklassen, z. B. Warteschlangen, Arrays, Stacks und Hashtabellen
- Klassen für den Datenbankzugriff
- E/A-Klassen
- **WinForms**-Klassen zum Erstellen der Benutzerschnittstelle
- Netzwerkklassen

Außerhalb der Laufzeitumgebung für die Basisklassen sind viele weitere Komponenten zur Behandlung der Benutzerschnittstelle sowie zur Durchführung anderer aufwendiger Operationen vorhanden.

3.1.2 Sicherheit

Die .NET-Laufzeitumgebung bietet Schutz und Sicherheit. Es handelt sich um eine verwaltete Umgebung, d. h. die Laufzeitumgebung verwaltet den Speicher für den Programmierer. Der Programmierer muss weder Speicher zuweisen noch diese Speicherzuweisungen wieder aufheben, all dies wird durch die Speicherbereinigung erledigt. Durch diese wird der Programmierer nicht nur entlastet, in einer Serverumgebung kann so auch die Anzahl der Speicherlecks erheblich reduziert werden. Dies vereinfacht die Entwicklung von Systemen, bei denen eine hohe Verfügbarkeit gefordert ist.

Zusätzlich bietet die .NET-Laufzeitumgebung eine Umgebungsprüfung. Zur Laufzeit wird über die Umgebung geprüft, ob der Ausführungscode typensicher ist. Auf diese Weise werden Fehler wie beispielsweise das Übergeben eines falschen Typs an eine Funktion, das Überschreiten zugewiesener Bereiche bei einem Lesevorgang oder das Ausführen von Code an ungeeigneten Standorten ermittelt.

Das Sicherheitssystem interagiert mit dem Prüfcode, um sicherzustellen, dass über den Code nur die Ausführung erfolgt, die diesem zugedacht ist. Die Sicherheitsanforderungen für einen bestimmten Codeabschnitt können genau festgelegt werden. So kann beispielsweise angegeben werden, dass ein Codeabschnitt in eine Arbeitsdatei schreiben muss; diese Anforderung wird während der Ausführung überprüft.

3.1.3 Unterstützung für leistungsfähige Tools

Microsoft stellt vier .NET-Sprachen bereit: Visual Basic, Visual C++ mit verwalteten Erweiterungen, C# und JScript. Andere Firmen arbeiten an Compilern für weitere Sprachen, hierbei reicht das Spektrum von COBOL bis Perl.

Die Fehlersuche wurde in der .NET-Laufzeitumgebung erheblich verbessert. Das gemeinsame Ausführungsmodell vereinfacht eine sprachübergreifende Fehlersuche, die Fehlerermittlung kann sich mühelos über Code erstrecken, der in verschiedenen Sprachen geschrieben wurde, unterschiedliche Prozesse ausführt oder auf verschiedenen Computern läuft.

Des Weiteren sind alle .NET-Programmieraufgaben in der Visual Studio-Umgebung zusammengefasst, d.h., Entwurf, Entwicklung, Fehlersuche und Bereitstellung von Anwendungen werden von hier aus realisiert.

3.1.4 Bereitstellen, Packen und weitere Funktionen

Die .NET-Laufzeitumgebung bietet auch auf diesen Gebieten Hilfe. Die Bereitstellung von Anwendungen wurde vereinfacht, in einigen Fällen fällt der traditionelle Installationsschritt vollständig weg. Da die Pakete in einem allgemeinen Format bereitgestellt werden, kann ein einzelnes Paket in einer beliebigen Umgebung ausgeführt werden, die .NET unterstützt. Darüber hinaus trennt die Umgebung die Anwendungskomponenten, sodass eine Anwendung nur mit den Komponenten läuft, mit denen sie geliefert wurde und nicht mit anderen Versionen, die zum Lieferumfang anderer Anwendungen gehören.

3.2 Metadaten

Metadaten sind das, was die .NET-Laufzeitumgebung zusammenhält. Metadaten stellen das Äquivalent zu den Typbibliotheken in der COM-Umgebung dar, bieten jedoch umfassendere Informationen.

Für jedes Objekt der .NET-Laufzeitumgebung werden in den Metadaten alle Objektinformationen aufgezeichnet, die zur Verwendung des Objekts erforderlich sind. Hierzu zählen:

- Der Name des Objekts
- Die Namen aller Felder des Objekts sowie deren Typen
- Die Namen aller Mitgliedsfunktionen, einschließlich Parametertypen und -namen

Auf diese Weise verfügt die .NET-Laufzeitumgebung über ausreichende Informationen zur Erstellung der Objekte, zum Aufrufen der Mitgliedsfunktionen oder für den Zugriff auf die Objektdaten, und der Compiler kann mit diesen Informationen ermitteln, welche Objekte verfügbar sind und welche sich in Verwendung befinden.

Diese Vereinheitlichung kommt sowohl dem Entwickler als auch dem Codebenutzer zugute: Der Codeentwickler kann auf einfache Weise Code erstellen, der in allen .NET-kompatiblen Sprachen verwendet werden kann, der Benutzer des Codes kann Objekte verwenden, die von anderen Programmierern erstellt wurden, unabhängig von der Sprache, in der die Objekte implementiert sind.

Zusätzlich ermöglichen Metadaten anderen Tools den Zugriff auf detaillierte Codeinformationen. Die Visual Studio-Shell verwendet diese Informationen im Objektbrowser und für Funktionen wie beispielsweise IntelliSense.

Darüber hinaus können – in einem Prozess, der als Reflektion bezeichnet wird – mit Hilfe des Laufzeitcodes die Metadaten abgefragt werden, um zu ermitteln, welche Objekte verfügbar und welche Funktionen und Felder für die Klasse vorhanden sind. Dies ähnelt der Verwendung von **IDispatch** in der COM-Umgebung, es wird jedoch ein einfacheres Modell angewendet. Natürlich ist ein derartiger Zugriff nicht stark typisiert, daher erfolgt eine Referenzierung der Metadaten bei der Mehrzahl der Softwarekomponenten eher zur Kompilierungs- als zur Laufzeit, aber für Anwendungen wie Skriptsprachen stellt dies eine sehr nützliche Funktion dar.

Der Endbenutzer kann außerdem über die Reflektion das Aussehen von Objekten ermitteln, nach Attributen suchen oder Methoden ausführen, deren Namen bis zur Laufzeit nicht bekannt sind.

3.3 Assemblierung

In der Vergangenheit konnte ein fertiges Softwarepaket über verschiedene Mechanismen bereitgestellt werden, beispielsweise als ausführbare Datei, DLL und LIB-Datei oder als DLL mit COM-Objekt und einer Typbibliothek.

In der .NET-Laufzeitumgebung wird als Packmethode die *Assemblierung* verwendet. Wird der Code über einen der .NET-Compiler kompiliert, wird er in eine Zwischenform konvertiert, die als »IL« bezeichnet wird. Die Assemblierung enthält

alle IL-, Metadaten- und weitere für die Ausführung erforderliche Dateien in einem einzigen Paket. Jede Assemblierung enthält eine Festlegung, in der die Dateien aufgeführt werden, die in der Assemblierung enthalten sind. Über die Festlegung wird gesteuert, welche Typen und Ressourcen außerhalb der Assemblierung offen gelegt werden und mit dem Verweis auf Typen und Ressourcen den Dateien zugeordnet werden, die die Typen und Ressourcen enthalten. In der Festlegung werden außerdem die Assemblierungen aufgelistet, von denen eine Assemblierung abhängt.

Assemblierungen sind unabhängig, d.h., sie enthalten genügend Informationen, um selbstbeschreibend zu sein.

Die Definition einer Assemblierung kann in einer einzigen Datei enthalten oder über mehrere Dateien verteilt sein. Die Verwendung mehrerer Dateien ermöglicht das Herunterladen von Abschnitten einer Assemblierung nach Bedarf.

3.4 Sprachinteroperabilität

Eines der Ziele der .NET-Laufzeitumgebung ist die Spracherkennung, die Möglichkeit, Code in beliebiger Sprache zu schreiben und zu verwenden. Es ist nicht nur möglich, in Visual Basic geschriebene Klassen aus C# oder C++ (oder einer anderen .NET-Sprache) aufzurufen, sondern eine Klasse, die in Visual Basic geschrieben wurde, kann auch als Basisklasse für eine C#-Klasse eingesetzt werden. Diese Klasse könnte dann wiederum von einer C++-Klasse eingesetzt werden.

Mit anderen Worten, es spielt keine Rolle, in welcher Sprache eine Klasse erstellt wurde. Dazu ist zu sagen, dass häufig gar nicht ermittelbar ist, in welcher Sprache eine Klasse geschrieben wurde.

In der Praxis stößt man jedoch auf ein paar Hindernisse. Einige Sprachen verfügen über Datentypen ohne Vorzeichen, die nicht von anderen Sprachen unterstützt werden, einige Sprachen unterstützen die Operatorüberladung. Die Erhaltung der Ausdrucksvielfalt dieser sehr funktionellen Sprachen und die gleichzeitige Gewährleistung der Klasseninteroperabilität mit anderen Sprachen stellt eine Herausforderung dar.

Die .NET-Laufzeitumgebung bietet ausreichende Unterstützung für Sprachen mit vielfältigen Funktionen[4], daher wird der Code, der in diesen Sprachen geschrieben wurde, nicht durch die einfacheren Sprachen in seiner Funktionalität eingeschränkt.

4. Dies trifft für das verwaltete C++ nicht ganz zu, denn im Vergleich zu C++ müssen einige Einschränkungen hingenommen werden.

Damit Klassen von der .NET-Laufzeitumgebung allgemein verwendbar sind, müssen die Klassen den so genannten *Common Language Specifications* (CLS) entsprechen, über die beschrieben wird, welche Funktionen in einer öffentlichen Schnittstelle der Klasse sichtbar sind (alle Funktionen können klassenintern verwendet werden). Die CLS verbieten beispielsweise das Offenlegen von Datentypen ohne Vorzeichen, da diese nicht von allen Sprachen verwendet werden können. Weitere Informationen zu den CLS finden Sie im .NET-SDK im Abschnitt zur sprachübergreifenden Interoperabilität.

Ein Benutzer, der C#-Code schreibt, kann angeben, dass dieser CLS-konform sein sollte. Der Compiler markiert in diesem Fall alle nicht konformen Bereiche. Weitere Informationen zu den spezifischen Einschränkungen für C#-Code hinsichtlich der CLS finden Sie im Abschnitt »CLS-Kompatibilität« in Kapitel 31, C# im Detail.

3.5 Attribute

Zur Umwandlung einer Klasse in eine Komponente sind häufig weitere Informationen erforderlich, beispielsweise Informationen dazu, wie eine Klasse auf Festplatte gespeichert werden kann oder wie Transaktionen gehandhabt werden sollten. Der traditionelle Ansatz besteht darin, die Informationen in eine separate Datei zu schreiben und zum Erstellen einer Komponente die Datei mit dem Quellcode zu kombinieren.

Das Problem bei diesem Ansatz liegt darin, dass Informationen häufig dupliziert werden. Dieses Vorgehen ist mühselig und fehleranfällig, außerdem verfügen Sie erst dann über die vollständige Komponente, wenn Sie beide Dateien besitzen[5].

Die .NET-Laufzeitumgebung unterstützt benutzerdefinierte Attribute (diese werden in C# einfach als *Attribute* bezeichnet), durch die beschreibende Informationen zusammen mit einem Objekt in den Metadaten gespeichert werden können. Die Daten können dann zu einem späteren Zeitpunkt abgerufen werden. Attribute bieten einen allgemeinen Mechanismus hierfür und werden in der Laufzeitumgebung häufig zum Speichern von Informationen bei der Änderung der Klassenverwendung eingesetzt. Attribute sind vollständig erweiterbar, wodurch der Programmierer Attribute definieren und verwenden kann.

5. Jeder, der jemals versucht hat, eine COM-Programmierung ohne Typbibliothek durchzuführen, kennt dieses Problem.

4 Schnelleinstieg in C#

Das vorliegende Kapitel enthält einen kurzen Überblick über C#. Hierbei werden grundlegende Programmierkenntnisse vorausgesetzt, d.h., es wird nicht jedes Detail ausführlich besprochen. Sollten Ihnen die Erläuterungen hier nicht ausreichen, finden Sie detailliertere Informationen unter den jeweiligen Themen in späteren Kapiteln.

4.1 Hello, Universe

Als ein Anhänger von SETI[1] dachte ich, dass ein »Hello, Universe«-Programm angemessener sei als das übliche »Hello, World«-Programm.

```
using System;
class Hello
{
   public static void Main(string[] args)
   {
      Console.WriteLine("Hello, Universe");

         // Befehlszeilenargumente durchlaufen
         // und ausgeben
      for (int arg = 0; arg < args.Length; arg++)
         Console.WriteLine("Arg {0}: {1}", arg, args[arg]);
   }
}
```

Wie bereits besprochen, weist die .NET-Laufzeitumgebung einen einheitlichen Namespace für alle Programminformationen (oder Metadaten) auf. Die **using System**-Klausel ist eine Methode zum Referenzieren der Klassen, die sich im **System**-Namespace befinden, daher können diese verwendet werden, ohne dass der Typenname mit **System** eingeleitet werden muss. Der **System**-Namespace enthält viele nützliche Klassen, beispielsweise die Klasse **Console**, die zur Kommunikation mit der Konsole (oder DOS-Box oder Befehlszeile für diejenigen, die noch nie eine Konsole gesehen haben) eingesetzt wird.

Da in C# keine globalen Funktionen vorhanden sind, wird im Beispiel eine **Hello**-Klasse mit statischer **Main**-Funktion deklariert, die als Startpunkt der Ausführung dient. **Main** kann ohne Parameter oder mit einem Zeichenfolgenarray deklariert

1. Search for Extraterrestrial Intelligence (Suche nach außerirdischer Intelligenz). Weitere Informationen finden Sie unter http://www.teamseti.org.

werden. Da es sich um die Startfunktion handelt, muss die Funktion statisch sein, d.h., sie ist nicht mit einer Objektinstanz verknüpft.

Die erste Zeile der Funktion ruft die **WriteLine**-Funktion der **Console**-Klasse auf, die ein »Hello, Universe« an die Konsole übermittelt. Der Schleifenoperator **For** durchläuft die übergebenen Parameter und schreibt eine Zeile für jeden Parameter in der Befehlszeile.

4.2 Namespaces und Using-Klausel

Die Namespaces in der .NET-Laufzeitumgebung dienen der Anordnung von Klassen und anderen Typen in einer einzigen hierarchischen Struktur. Die ordnungsgemäße Verwendung der Namespaces vereinfacht den Einsatz von Klassen und verhindert Kollisionen mit Klassen, die von anderen Programmierern stammen.

Namespaces kann man sich als Möglichkeit zur Festlegung sehr langer Namen für Klassen und andere Typen vorstellen, ohne dass immer ein vollständiger Name eingegeben werden muss.

Namespaces werden mit Hilfe der **namespace**-Anweisung definiert. Für Strukturen mit mehreren Ebenen können Namespaces verschachtelt werden:

```
namespace Outer
{
   namespace Inner
   {
      class MyClass
      {
         public static void Function() {}
      }
   }
}
```

Dieses Beispiel erfordert sehr viele Eingaben sowie Einschübe, eine Vereinfachung lautet:

```
namespace Outer.Inner
{
   class MyClass
   {
      public static void Function() {}
   }
}
```

Jede Quelldatei kann beliebig viele Namespaces definieren.

Wie bereits im Abschnitt »Hello, Universe« erwähnt, werden die Metadaten für Typen über **using** in das aktuelle Programm importiert, sodass die Typen leichter referenziert werden können. **Using** ist lediglich ein Shortcut, der den Eingabeaufwand verringert, der zur Referenzierung von Elementen anfällt, wie die nachfolgende Tabelle zeigt.

Using-Klausel	Quellzeile
<keine>	System.Console.WriteLine(»Hello«);
Using System	Console.WriteLine(»Hello«);

Kollisionen zwischen gleichnamigen Typen oder Namespaces können immer durch den vollqualifizierten Typennamen beseitigt werden. Dies kann ein sehr langer Name sein, wenn es sich um eine stark verschachtelte Klasse handelt, daher hier eine Variante der **Using**-Klausel, die das Definieren eines Alias für eine Klasse ermöglicht:

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

Damit der Code lesbarer wird, werden in den verwendeten Beispielen selten Namespaces verwendet, im tatsächlichen Code sollten sie jedoch eingesetzt werden.

4.3 Namespace und Assemblierung

Ein Objekt kann nur dann innerhalb einer C#-Quelldatei verwendet werden, wenn es durch den C#-Compiler ermittelt werden kann. Standardmäßig öffnet der Compiler nur die Assemblierung **mscorlib.dll**, die die Hauptfunktionen der Common Language Runtime enthält.

Zur Referenzierung von Objekten, die sich in anderen Assemblierungen befinden, muss der Name der assemblierten Datei an den Compiler übergeben werden. Dies kann über die Befehlszeile und die Option **/r:<Assemblierung>** oder innerhalb der Visual Studio-IDE durch Hinzufügen eines Verweises auf das C#-Projekt geschehen.

Üblicherweise besteht eine Beziehung zwischen dem Namespace, in dem sich das Objekt befindet, und dem Namen der Assemblierung, in der das Objekt gespeichert ist. Die Typen im Namespace **System.Net** beispielsweise befinden sich in der Assemblierung **System.Net.dll**. Assemblierungstypen basieren üblicherweise auf den Verwendungsmustern des Objekts in dieser Assemblierung; ein sehr häufig oder selten verwendeter Typ in einem Namespace kann in einer eigenen Assemblierung vorliegen.

Der exakte Name der Assemblierung, in der ein Objekt enthalten ist, kann der Objektdokumentation entnommen werden.

4.4 Grundlegende Datentypen

C# unterstützt die üblichen Datentypensätze. Für jeden von C# unterstützten Datentyp ist ein entsprechender .NET Common Language Runtime-Typ vorhanden. Der Datentyp **int** in C# wird beispielsweise dem Typ **System.Int32** in der Laufzeitumgebung zugeordnet. **System.Int32** kann fast überall dort verwendet werden, wo **int** zum Einsatz kommt. Dieses Vorgehen wird jedoch nicht empfohlen, da es die Lesbarkeit des Codes herabsetzt.

Die grundlegenden Datentypen werden in der nachstehenden Tabelle beschrieben. Die Laufzeittypen befinden sich allesamt im **System**-Namespace der .NET Common Language Runtime.

Typ	Byte	Laufzeittyp	Beschreibung
byte	1	Byte	**Byte**-Wert ohne Vorzeichen
sbyte	1	SByte	**Byte**-Wert mit Vorzeichen
short	2	Int16	**Short**-Wert mit Vorzeichen
ushort	2	UInt16	**Short**-Wert ohne Vorzeichen
int	4	Int32	**Integer**-Wert mit Vorzeichen
uint	4	UInt32	**Integer**-Wert ohne Vorzeichen
long	8	Int64	Langer **integer**-Wert mit Vorzeichen
ulong	8	UInt64	Langer **integer**-Wert ohne Vorzeichen
float	4	Single	Gleitkommazahl
double	8	Double	Gleitkommazahl mit doppelter Genauigkeit
decimal	8	Decimal	Zahl mit fester Genauigkeit
string	String	Unicode-Zeichenfolge	
char	Char	Unicode-Zeichen	
bool	Boolean	Boolescher Wert	

Der Unterschied zwischen den grundlegenden (integrierten) Datentypen von C# ist eher formell, da benutzerdefinierte Typen auf die gleiche Weise verwendet werden können wie die integrierten Typen. Tatsächlich besteht der einzige wirkliche Unterschied zwischen den integrierten und den benutzerdefinierten Datentypen darin, dass es möglich ist, für die integrierten Datentypen literale Werte zu schreiben.

Datentypen werden in Werte- und Verweistypen unterteilt. Wertetypen werden entweder dem Stack zugeordnet oder sind strukturintern zugewiesen. Verweistypen sind Heaps zugeordnet.

Sowohl die Verweis- als auch die Wertetypen werden von der Basisklasse **object** abgeleitet. In Fällen, in denen ein Wertetyp als **object** fungieren muss, wird ein Wrapper, der den Wertetyp als Verweisobjekt erscheinen lässt, dem Heap zugeordnet; der Wert des Wertetyps wird kopiert. Dieser Vorgang wird als *Boxing* bezeichnet, der umgekehrte Vorgang als *Unboxing*. Durch das Boxing und Unboxing kann ein *beliebiger* Typ als **object** behandelt werden. Dies macht folgenden Code möglich:

```
using System;
class Hello
{
   public static void Main(string[] args)
   {
      Console.WriteLine("Value is: {0}", 3.ToString());
   }
}
```

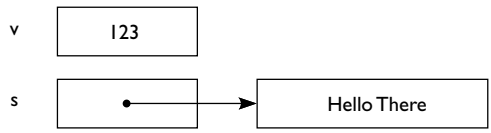
Hier wurde für den **integer**-Wert 3 ein Boxing durchgeführt; die Funktion **Int32.ToString()** wird für den geboxten Wert aufgerufen.

C#-Arrays können entweder in mehrdimensionaler oder unregelmäßiger (jagged) Form deklariert werden. Erweiterte Datenstrukturen wie Stacks und Hashtabellen sind im Namespace **System.Collection** enthalten.

4.5 Klassen, Strukturen und Schnittstellen

In C# wird das Schlüsselwort **class** zum Deklarieren eines (dem Heap zugeordneten) Verweistyps verwendet, das Schlüsselwort **struct** dient der Deklaration von Wertetypen. Strukturen (Schlüsselwort **struct)** werden für schlanke kleine Objekte verwendet, die wie integrierte Typen fungieren müssen, in allen anderen Fällen werden Klassen eingesetzt. Der **int**-Typ ist beispielsweise ein Wertetyp, der Typ **string** ist ein Verweistyp. Das folgende Diagramm zeigt die Funktionsweise dieser Typen:

```
int v = 123;
string s = "Hello There";
```



C# und die .NET-Laufzeitumgebung bieten keine Unterstützung für die Mehrfachvererbung bei Klassen, unterstützen jedoch die mehrfache Implementierung von Schnittstellen.

4.6 Anweisungen

Die C#-Anweisungen ähneln denen von C++ sehr stark. Die Unterschiede bestehen in einigen Änderungen, durch die weniger Fehler auftreten, sowie in einigen neuen Anweisungen. Die Anweisung **foreach** wird zum Durchlaufen von Arrays und Auflistungen eingesetzt, die Anweisung **lock** wird für das gegenseitige Ausschließen in Threadingszenarien verwendet. Die Anweisungen **checked** und **unchecked** dienen der Überlaufsteuerung in arithmetischen Operationen und Konvertierungen.

4.7 Enum

Aufzählungsbezeichner (**enum**) werden zum einfachen und typensicheren Deklarieren bezogener Konstanten verwendet – beispielsweise zum Auflisten der Farben, die ein Steuerelement annehmen kann. Beispiel:

```
enum Colors
{
   red,
   green,
   blue
}
```

Diese Aufzählungsbezeichner werden in Kapitel 20, Aufzählungsbezeichner, näher erläutert.

4.8 Zuweisungen und Ereignisse

Zuweisungen (**delegate**) sind eine typensichere, objektorientierte Implementierung von Funktionszeigern und werden in vielen Situationen eingesetzt, in denen eine Komponente die Komponente aufrufen muss, die sie verwendet. Zuweisungen werden weitestgehend als Grundlage für Ereignisse eingesetzt, die das einfache Registrieren von Zuweisungen für ein Ereignis ermöglichen. Zuweisungen werden in Kapitel 22 ausführlich behandelt.

In den .NET-Frameworks finden Zuweisungen und Ereignisse breite Anwendung.

4.9 Eigenschaften und Indizierer

C# unterstützt Eigenschaften und Indizierer, die zur Trennung der Schnittstelle eines Objekts von der Objektimplementierung nützlich sind. Statt einem Benutzer direkten Zugriff auf ein Feld oder Array zu gewähren, ermöglichen Eigenschaften oder Indizierer einem angegebenen Anweisungsblock diesen Zugriff, während die Verwendung von Feld oder Array weiterhin möglich ist. Hier ein einfaches Beispiel:

```
using System;
class Circle
{
   public int X
   {
      get
      {
         return(x);
      }
      set
      {
         x = value;
         // Objekt hier zeichnen.
      }
   }
   int x;
}
class Test
{
   public static void Main()
   {
      Circle c = new Circle();
      c.X = 35;
```

```
    }
}
```

In diesem Beispiel wird die Zugriffsroutine **get** oder **set** aufgerufen, wenn auf die Eigenschaft **X** verwiesen wird.

4.10 Attribute

In C# und den .NET-Frameworks verwendet der Codeentwickler Attribute dazu, erklärende Informationen an Codeabschnitte zu übermitteln, die an diesen Informationen interessiert sind. So kann angegeben werden, welche Felder eines Objekts serialisiert werden sollten, welcher Transaktionskontext bei der Objektausführung zu verwenden ist, für welche Felder ein Marshaling in systemeigene Funktionen durchgeführt werden sollte oder wie eine Klasse in einem Klassenbrowser angezeigt wird.

Attribute werden von eckigen Klammern umschlossen. Ein typisches Attribut kann folgendermaßen aussehen:

```
[CodeReview("12/31/1999", Comment="Gut gemacht")]
```

Attributinformationen werden zur Laufzeit über einen Vorgang abgerufen, der als Reflektion bezeichnet wird. Neue Attribute können leicht geschrieben und auf Codeelemente angewendet werden (beispielsweise auf Klassen, Mitglieder oder Parameter) und per Reflektion abgerufen werden.

35 C# im Vergleich zu anderen Sprachen

In diesem Kapitel soll C# mit anderen Sprachen verglichen werden. C#, C++ und Java gehen auf dieselben Wurzeln zurück und ähneln sich daher stärker als viele andere Sprachen. Visual Basic ist C# nicht so ähnlich wie die zuvor genannten Sprachen, weist jedoch immer noch einige übereinstimmende syntaktische Elemente auf.

In einem gesonderten Abschnitt dieses Kapitels werden die .NET-Versionen von Visual C++ und Visual Basic besprochen, da diese sich von ihren jeweiligen Vorgängerversionen ebenfalls unterscheiden.

35.1 Unterschiede zwischen C# und C/C++

Der C#-Code wird C- und C++-Programmierern vertraut vorkommen, dennoch gibt es einige wesentliche und verschiedene weniger bedeutende Unterschiede. Im Folgenden erhalten Sie einen Überblick über diese Unterschiede. Einen detaillierteren Vergleich finden Sie im entsprechenden Microsoft-Whitepaper, C# for the C++ Programmer.

35.1.1 Eine verwaltete Umgebung

C# wird in der .NET-Laufzeitumgebung ausgeführt. Dies bedeutet nicht nur, dass viele Vorgänge nicht der Steuerung des Programmierers unterliegen, gleichzeitig wird auch ein brandneuer Satz Frameworks bereitgestellt. Alles in allem ergeben sich hierdurch verschiedene Änderungen.

- Das Löschen von Objekten erfolgt über die Speicherbereinigung und wird ausgeführt, wenn das Objekt nicht länger benötigt wird. Destruktoren (= Finalisierungsroutinen) können zur Durchführung von Bereinigungsaufgaben eingesetzt werden, dies jedoch nicht im gleichen Umfang wie bei C++-Destruktoren.
- In C# gibt es keine Zeiger. Nun, es gibt Zeiger im **unsafe**-Modus, diese werden jedoch selten eingesetzt. Statt dessen werden Verweise verwendet, die denen von C++ ähneln, jedoch nicht alle der C++-Einschränkungen aufweisen.
- Quellen werden in Assemblierungen kompiliert, die sowohl den kompilierten Code (ausgedrückt in der .NET-Zwischensprache IL) als auch Metadaten zur Beschreibung des kompilierten Codes enthalten. Alle .NET-Sprachen fragen über die Metadaten die gleichen Informationen ab, wie sie in den C++-**.h**-Dateien enthalten sind; die **include**-Dateien fallen weg.
- Das Aufrufen von systemeigenem Code ist etwas zeitaufwendiger.

- Es ist keine C/C++-Laufzeitbibliothek vorhanden. Die hiermit ausgeführten Operationen, beispielsweise die Zeichenfolgenbearbeitung, E/A-Operationen und andere Routinen, können mit dem .NET-Laufzeitsystem ausgeführt werden und befinden sich in dem Namespace, dessen Name mit **System** beginnt.
- Anstelle einer Fehlerrückgabe wird die Ausnahmebehandlung verwendet.

35.1.2 .NET-Objekte

C#-Objekte gehen allesamt auf die Basisklasse **object** zurück, daher ist nur eine Einfachvererbung von Klassen möglich, obwohl das mehrfache Implementieren von Schnittstellen zulässig ist.

Kleine, schlanke Objekte, beispielsweise Datentypen, können als Strukturen deklariert werden (so genannte Wertetypen), d. h., sie werden nicht dem Heap, sondern einem Stack zugeordnet.

C#-Strukturen und andere Wertetypen (einschließlich der integrierten Datentypen) können in Situationen eingesetzt werden, in denen ein Objektboxing erforderlich ist. Bei diesem Vorgang werden die Werte in einen dem Heap zugeordneten Wrapper kopiert, der mit den dem Heap zugeordneten Objekten kompatibel ist (auch Verweisobjekte genannt). Dies vereinheitlicht das Typensystem und ermöglicht die Verwendung von Variablen als Objekte. Gleichzeitig entsteht kein Overhead, wenn eine Vereinheitlichung nicht erforderlich ist.

C# unterstützt Eigenschaften und Indizierer zum Trennen des Benutzermodells eines Objekts von der Objektimplementierung und unterstützt Zuweisungen und Ereignisse zum Kapseln der Funktionalität von Zeigern und Rückrufen.

C# stellt das **params**-Schlüsselwort bereit, um eine den **vararg**-Funktionen ähnliche Unterstützung zu bieten.

35.1.3 C#-Anweisungen

C#-Anweisungen ähneln den C++-Anweisungen in vielerlei Hinsicht. Es gibt zwei merkliche Unterschiede:

- Das **new**-Schlüsselwort bedeutet »Abrufen einer neuen Kopie von«. Das Objekt ist dem Heap zugeordnet, wenn es sich um einen Verweistyp handelt, und ist dem Stack oder intern zugeordnet, wenn es sich um einen Wertetyp handelt.
- Alle Anweisungen, mit denen eine boolesche Bedingung geprüft wird, erfordern jetzt eine Variable vom Typ **bool**. Es findet keine automatische Konvertierung von **int** in **bool** statt, daher ist **if (i)** unzulässig.

- **Switch**-Anweisungen verhindern Fehlschläge, um die Fehlerzahl zu verringern. **Switch** kann auch für Zeichenfolgenwerte verwendet werden.
- Für das Durchlaufen von Objekten und Auflistungen kann **foreach** eingesetzt werden.
- Über **checked** und **unchecked** werden arithmetische Operationen und Konvertierungen auf einen Überlauf geprüft.
- Feste Zuordnungen erfordern, dass Objekte vor der Verwendung über einen festen Wert verfügen.

35.1.4 Attribute

Attribute dienen der Weitergabe beschreibender Daten vom Programmierer an anderen Code. Bei diesem Code kann es sich um die Laufzeitumgebung, einen Designer, ein Tool zur Codeanalyse oder um ein benutzerdefiniertes Tool handeln. Attributinformationen werden über einen Vorgang abgerufen, der als Reflektion bezeichnet wird.

Attribute werden von eckigen Klammern umschlossen und können für Klassen, Mitglieder, Parameter und weitere Codeelemente gesetzt werden. Beispiel:

```
[CodeReview("1/1/199", Comment="Rockin'")]
class Test
{
}
```

35.1.5 Versionssteuerung

C# ermöglicht gegenüber C++ eine verbesserte Versionssteuerung. Da das Laufzeitsystem den Mitgliedsentwurf handhabt, stellt die binäre Kompatibilität kein Problem dar. Die Laufzeit bietet, falls gewünscht, die Möglichkeit zur Verwendung nebeneinander existierender Komponentenversionen sowie eine geeignete Semantik zur Versionssteuerung für Frameworks, C# ermöglicht dem Programmierer die Angabe des beabsichtigten Versionszwecks.

35.1.6 Codeorganisation

C# verwendet keine Headerdateien, sämtlicher Code wird intern geschrieben, und während eine Präprozessorunterstützung für bedingten Code vorhanden ist, werden Makros nicht unterstützt. Die Einschränkungen ermöglichen dem Compiler eine schnellere Analyse des C#-Codes und ermöglichen es darüber hinaus einer Entwicklungsumgebung, den C#-Code besser zu verstehen.

Zusätzlich liegen im C#-Code weder Reihenfolgenabhängigkeiten noch Vorwärtsdeklarationen vor. Die Reihenfolge der Klassen in den Quelldateien ist unerheblich, die Klassen können nach Wunsch umgestellt werden.

35.1.7 Fehlende C++-Funktionen

Die folgenden C++-Funktionen stehen in C# nicht zur Verfügung:

- Mehrfachvererbung
- **Const**-Mitgliedsfunktionen oder -Parameter. **Const**-Felder werden unterstützt.
- Globale Variablen
- Typedef
- Konvertierung durch Erstellungsroutine
- Standardargumente für Funktionsparameter

35.2 Unterschiede zwischen C# und Java

C# und Java gehen auf gleiche Wurzeln zurück, daher ist es nicht überraschend, dass zwischen den beiden Sprachen Ähnlichkeiten vorhanden sind. Dennoch gibt es auch einige Unterschiede. Der größte Unterschied besteht darin, dass sich C# oberhalb der.NET-Frameworks und der .NET-Laufzeit befindet und Java den Frameworks und der Laufzeit von Java übergeordnet ist.

35.2.1 Datentypen

C# verfügt über primitivere Datentypen als Java. In der folgenden Tabelle werden die Java-Typen und deren C#-Äquivalente zusammengefasst:

C#-Typ	Java-Typ	Kommentar
sbyte	byte	C#-**byte** hat kein Vorzeichen
short	short	
int	int	
long	long	
bool	Boolean	
float	float	
double	double	
char	char	
string	string	

C#-Typ	Java-Typ	Kommentar
object	**object**	
byte	**Byte**-Wert ohne Vorzeichen	
ushort	**Short**-Wert ohne Vorzeichen	
uint	**Integer**-Wert ohne Vorzeichen	
ulong	**Long**-Wert ohne Vorzeichen	
decimal	Finanzdaten-/Währungstyp	

In Java werden die primitiven und die objektbasierten Datentypen voneinander getrennt. Damit die primitiven Typen an der objektbasierten Welt teilhaben können (beispielsweise in einer Auflistung), müssen sie in einer Instanz einer Wrapperklasse platziert werden, und die Wrapperklasse wird anschließend in der Auflistung platziert.

C# geht dieses Problem anders an. In C# sind die primitiven Typen (wie bei Java) dem Stack zugeordnet, sie werden jedoch auch als von der ultimativen Basisklasse **object** abgeleitet betrachtet. Dies bedeutet, dass für die primitiven Typen Mitgliedsfunktionen definiert und aufgerufen werden können. Mit anderen Worten, der folgende Code ist zulässig:

```
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(5.ToString());
   }
}
```

Die Konstante **5** weist den Typ **int** auf, das **ToString()**-Mitglied ist für den Typ **int** definiert, daher kann der Compiler einen Aufruf generieren und den **int** an die Mitgliedsfunktion übergeben, als handele es sich um ein Objekt.

Dies funktioniert prima, wenn der Compiler die Handhabung eines primitiven Typs kennt, jedoch nicht, wenn ein primitiver Typ mit den dem Heap zugeordneten Objekten einer Auflistung zusammenarbeiten muss. Immer dann, wenn ein primitiver Typ verwendet wird und ein Parameter vom Typ **object** erforderlich ist, führt der Compiler ein automatisches Boxing des primitiven Typs in einen dem Heap zugeordneten Wrapper durch. Hier ein Beispiel zum Boxing:

```
using System;
class Test
{
   public static void Main()
   {
      int v = 55;
      object o = v;       // Boxing von v in o
      Console.WriteLine("Value is: {0}", o);
      int v2 = (int) o;   // Unboxing zurück in int
   }
}
```

In diesem Codeabschnitt wird der **integer**-Wert in ein **object** geboxt und der Mitgliedsfunktion **Console.WriteLine()** als Objektparameter übergeben. Das Deklarieren der Objektvariable erfolgt nur zur Verdeutlichung; im tatsächlichen Code würde **v** direkt übergeben, das Boxing würde auf Aufruferseite erfolgen. Der geboxte **int**-Wert kann durch eine **cast**-Operation extrahiert werden.

35.2.2 Erweitern des Typensystems

Die primitiven C#-Typen (mit Ausnahme von **string** und **object**) werden auch als Wertetypen bezeichnet, da Variablen dieser Typen tatsächliche Werte enthalten. Andere Typen werden Verweistypen genannt, da die zugehörigen Variablen Verweise enthalten.

In C# kann ein Programmierer das Typensystem durch Implementierung eines benutzerdefinierten Wertetyps erweitern. Diese Typen werden mit Hilfe des Schlüsselwortes **struct** implementiert und verhalten sich ähnlich wie die integrierten Wertetypen. Sie sind dem Stack zugeordnet, können Mitgliedsfunktionen aufweisen, und bei Bedarf wird ein Boxing oder Unboxing durchgeführt. Tatsächlich sind alle primitiven C#-Typen als Wertetypen implementiert, der einzige syntaktische Unterschied zwischen den integrierten Typen und den benutzerdefinierten Typen besteht darin, dass die integrierten Typen als Konstanten geschrieben werden können.

Damit sich die benutzerdefinierten Typen natürlich verhalten, können C#-Strukturen arithmetische Operatoren überladen, um numerische Operationen und Konvertierungen auszuführen, d.h., zwischen Strukturen und anderen Typen können implizite und explizite Konvertierungen erfolgen. C# unterstützt des Weiteren auch das Überladen von Klassen.

Ein **struct** wird mit Hilfe der gleichen Syntax geschrieben wie **class**, abgesehen davon, dass eine Struktur (neben der impliziten Basisklasse **object**) keine Basisklasse besitzen kann. Schnittstellen können jedoch implementiert werden.

35.2.3 Klassen

C#-Klassen ähneln den Java-Klassen sehr stark. Im Hinblick auf Konstanten, Basisklassen und Erstellungsroutinen, statische Konstruktoren, virtuelle Funktionen, Ausblenden und Versionssteuerung, Mitgliedszugriff, **ref**- und **out**-Parameter sowie das Identifizieren von Typen bestehen jedoch Unterschiede.

Konstanten

Java verwendet zum Deklarieren einer Klassenkonstante **static final**. In C# wird **static final** durch **const** ersetzt. Zusätzlich fügt C# das Schlüsselwort **readonly** hinzu, das verwendet wird, wenn die Konstantenwerte zur Kompilierungszeit nicht ermittelt werden können. **Readonly**-Felder können nur über eine Initialisierungsroutine oder eine Klassenerstellungsroutine gesetzt werden.

Basisklassen und Erstellungsroutinen

C# verwendet sowohl für das Definieren der Basisklasse und die Schnittstellen einer Klasse als auch für das Aufrufen weiterer Erstellungsroutinen die C++-Syntax. Eine C#-Klasse kann folgendermaßen aussehen:

```
class MyObject: Control, IFormattable
{
   public Control(int value)
   {
      this.value = value;
   }
      public Control() : base(value)
   {
   }
   int value;
}
```

Statische Erstellungsroutinen

Anstelle eines statischen Initialisierungsblocks verwendet C# statische Erstellungsroutinen, die mit Hilfe des Schlüsselworts **static** geschrieben werden, das einer parameterlosen Erstellungsroutine vorangestellt wird.

Virtuelle Funktionen, Ausblenden und Versionssteuerung

In C# sind standardmäßig alle Funktionen nicht virtuell, um eine virtuelle Funktion zu erhalten, muss explizit **virtual** angegeben werden. Aufgrund dieser Tatsache gibt es keine finalen Methoden in C#, auch wenn das Äquivalent zu einer finalen Klasse mit Hilfe von **sealed** erreicht werden kann.

C# bietet eine bessere Versionsunterstützung als Java, woraus sich einige kleinere Änderungen ergeben. Die Methodenüberladung erfolgt statt nach Signatur nach Name, d.h., das Hinzufügen von Klassen zu einer Basisklasse wirkt sich nicht auf das Programmverhalten aus. Sehen Sie sich folgendes Beispiel an:

```
public class B
{
}
public class D: B
{
   public void Process(object o) {}
}
class Test
{
   public static void Main()
   {
      D d = new D();
      d.Process(15);   // Aufruf durchführen
   }
}
```

Wenn der Provider der Basisklasse eine Verarbeitungsfunktion mit größerer Übereinstimmung hinzufügt, ändert sich das Verhalten:

```
public class B
{
   public void Process(int v) {}
}
public class D: B
{
   public void Process(object o) {}
}
class Test
{
   public static void Main()
   {
```

```
        D d = new D();
        d.Process(15);   // Aufruf durchführen
    }
}
```

In Java führt dies zum Aufruf der Klassenimplementierung, was wahrscheinlich falsch ist. In C# setzt das Programm seine Arbeit fort.

Zur Handhabung des ähnlichen Falles für virtuelle Funktionen muss in C# die Versionssemantik explizit angegeben werden. Wenn es sich bei **Process()** um eine virtuelle Funktion der abgeleiteten Klasse handelt, würde Java annehmen, dass es sich bei jeder Basisklassenfunktion mit übereinstimmender Signatur um eine Basis für die virtuelle Funktion handelt, was in den meisten Fällen nicht stimmt.

In C# werden virtuelle Funktionen nur außer Kraft gesetzt, wenn das Schlüsselwort **override** verwendet wird. Weitere Informationen hierzu finden Sie in Kapitel 11, Versionssteuerung mit **new** und **override**.

Mitgliedszugriff

Neben den Zugriffsbezeichnern **public**, **private** und **protected** steht in C# der Zugriffsbezeichner **internal** zur Verfügung. Auf Mitglieder mit dem Zugriff **internal** kann von Klassen desselben Projekts aus, jedoch nicht von außerhalb des Projekts zugegriffen werden.

Ref- und Out-Parameter

In Java werden Parameter immer nach Wert übergeben. C# ermöglicht mit dem Schlüsselwort **ref** das Übergeben von Parametern als Verweis. Dies ermöglicht der Mitgliedsfunktion, den Wert des Parameters zu ändern.

C# bietet darüber hinaus die Möglichkeit, Parameter mit Hilfe des **out**-Schlüsselwortes zu definieren, das genau wie **ref** funktioniert, abgesehen davon, dass die als Parameter übergebene Variable den Wert nicht vor dem Aufruf kennen muss.

Identifizieren von Typen

Java verwendet die Methode **GetClass()**, um ein **Class**-Objekt mit Informationen zum aufgerufenen Objekt zurückzugeben. Das **Type**-Objekt ist das .NET-Äquivalent zum **Class**-Objekt und kann auf verschiedene Weise abgerufen werden:

- Durch Aufrufen der **GetType()**-Methode für eine Objektinstanz
- Durch Verwenden des **typeof**-Operators für den Typennamen
- Durch Ermitteln des Typs nach Name mit der Klasse **System.Reflection**

35.2.4 Schnittstellen

Während Java-Schnittstellen Konstanten besitzen können, ist dies in C# nicht möglich. Bei der Implementierung von Schnittstellen stellt C# eine explizite Schnittstellenimplementierung bereit. Dies ermöglicht einer Klasse die Implementierung zweier Schnittstellen unterschiedlicher Quelle und gleichem Mitgliedsnamen. Die explizite Schnittstellenimplementierung kann auch dazu verwendet werden, Schnittstellenimplementierungen vor dem Benutzer zu verbergen. Weitere Informationen zu diesem Thema finden Sie in Kapitel 10, Schnittstellen.

35.2.5 Eigenschaften und Indizierer

In Java-Programmen werden Eigenschaften häufig durch das Deklarieren von **get**- und **set**-Methoden eingesetzt. In C# erscheint eine Eigenschaft dem Benutzer einer Klasse als Feld, bietet jedoch **get**- und **set**-Zugriffsroutinen zum Durchführen von Lese- oder Schreiboperationen.

Ein Indizierer ähnelt einer Eigenschaft, statt jedoch wie ein Feld auszusehen, wird der Indizierer dem Benutzer als Array angezeigt. Wie die Eigenschaften besitzen Indizierer **get**- und **set**-Zugriffsroutinen; im Gegensatz zu den Eigenscahften kann ein Indizierer jedoch für verschiedene Typen überladen werden. Dies ermöglicht das Indizieren von Datenbankzeilen nach Spaltenzahl und Spaltenname sowie das Indizieren von Hashtabellen nach Hashschlüssel.

35.2.6 Zuweisungen und Ereignisse

Wenn ein Objekt in Java einen Rückruf benötigt, wird mit einer Schnittstelle angegeben, wie das Objekt gebildet wird, und eine Methode innerhalb der Schnittstelle wird für den Rückruf aufgerufen. Bei C#-Schnittstellen kann ein ähnlicher Ansatz verwendet werden.

C# stellt Zuweisungen bereit, die man sich wie typensichere Funktionszeiger vorstellen kann. Eine Klasse kann eine Zuweisung für eine Funktion der Klasse erstellen, anschließend kann die Zuweisung an eine Funktion übergeben werden, die diese Zuweisung akzeptiert. Anschließend kann die Funktion die Zuweisung aufrufen.

C# baut auf Zuweisungen mit Ereignissen auf, die von den .NET-Frameworks verwendet werden. Ereignisse implementieren das Veröffentlichen-Abonnieren-Prinzip; wenn ein Objekt (beispielsweise ein Steuerelement) ein Klickereignis unterstützt, kann eine beliebige Anzahl weiterer Klassen eine Zuweisung registrieren, die bei Auslösen des Ereignisses aufgerufen werden soll.

35.2.7 Attribute

Attribute dienen der Weitergabe beschreibender Daten vom Programmierer an anderen Code. Bei diesem Code kann es sich um die Laufzeitumgebung, einen Designer, ein Tool zur Codeanalyse oder um ein benutzerdefiniertes Tool handeln. Attributinformationen werden über einen Vorgang abgerufen, der als Reflektion bezeichnet wird.

Attribute werden von eckigen Klammern umschlossen und können für Klassen, Mitglieder, Parameter und weitere Codeelemente gesetzt werden. Beispiel:

```
[CodeReview("1/1/199", Comment="Rockin'")]
class Test
{
}
```

35.2.8 Anweisungen

Die C#-Anweisungen werden dem Java-Programmierer bekannt vorkommen, es gibt jedoch einige neue Anweisungen und ein paar Unterschiede bei den vorhandenen Anweisungen zu beachten.

import kontra using

Die **import**-Anweisung wird in Java dazu eingesetzt, ein Paket zu ermitteln und die Typen in die aktuelle Datei zu importieren.

In C# wird diese Operation aufgeteilt. Die Assemblierungen, von denen ein Codeabschnitt abhängt, müssen explizit angegeben werden, entweder über die Befehlszeile mit Hilfe der Option **/r** oder in der Visual Studio-IDE. Die grundlegenden Systemfunktionen (aktuell die in **mscorlib.dll** enthaltenen Funktionen) sind die einzigen, die automatisch vom Compiler importiert werden.

Nachdem eine Assemblierung referenziert wurde, können die enthaltenen Typen verwendet werden, müssen jedoch unter Verwendung des vollqualifizierten Namens angegeben werden. Die reguläre Ausdrucksklasse heißt beispielsweise **System.Text.RegularExpressions.Regex**. Dieser Klassenname kann direkt verwendet werden, oder es werden mit Hilfe der **using**-Anweisung die Typen eines Namespaces in den Namespace oberster Ebene importiert. Mit der folgenden **using**-Klausel

```
using System.Text.RegularExpressions;
```

kann die Klasse einfach durch Einsatz von **Regex** angegeben werden. Es gibt außerdem eine Variante der **using**-Anweisung, die zur Vermeidung von Namenskollisionen das Angeben eines Typenalias ermöglicht.

Überlauf

Java kann weder bei Konvertierungen noch bei mathematischen Fehlern Überläufe ermitteln.

In C# kann eine solche Ermittlung durch die **checked**- und **unchecked**-Anweisungen und Operatoren gesteuert werden. Konvertierungen und mathematische Operationen, die in einem als **checked** deklarierten Kontext erfolgen, erzeugen Ausnahmen, wenn die Operation zu einem Überlauf oder Fehlern führt; die Ausführung einer solchen Operation in einem als **unchecked** deklarierten Kontext führt nie zu einer Fehlerausgabe. Der Standardkontext wird durch das Compilerflag **/checked** gesteuert.

Unsicherer Code

Der so genannte unsichere Code in C# ermöglicht die Verwendung von Zeigervariablen und wird eingesetzt, wenn die Leistung extrem wichtig ist oder eine Integration mit vorhandener Software benötigt wird, beispielsweise mit COM-Objekten oder systemeigenem Code in DLLs. Die **fixed**-Anweisung wird dazu verwendet, ein Objekt »festzunageln«, damit es bei einer Speicherbereinigung nicht verschoben wird.

Da von der Laufzeit nicht geprüft werden kann, ob der unsichere Code gefahrlos ausgeführt werden kann, kann eine Ausführung nur erfolgen, wenn dem Code vom Laufzeitsystem vertraut wird. Dies verhindert eine Ausführung in Downloadszenarien.

Zeichenfolgen

Das C#-Zeichenfolgenobjekt kann für den Zugriff auf bestimmte Zeichen indiziert werden. Beim Zeichenfolgenvergleich werden nicht die Zeichenfolgenverweise, sondern die Zeichenfolgenwerte miteinander verglichen.

Zeichenfolgenliterale weisen in C# ebenfalls Unterschiede auf; C# unterstützt Escapezeichen innerhalb von Zeichenfolgen, die zum Einfügen von Sonderzeichen verwendet werden. Die Zeichenfolge **\t** wird beispielsweise in ein Tabulatorzeichen übersetzt.

Dokumentation

Die XML-Dokumentation in C# ähnelt Javadoc, C# gibt jedoch nicht die Struktur der Dokumentation vor, und der Compiler prüft die Richtigkeit der Dokumentation und erzeugt eindeutige Bezeichner für Verknüpfungen.

Weitere Unterschiede

Es gibt einige weitere Unterschiede:

- Der **>>>**-Operator ist in C# nicht vorhanden, da der **>>**-Operator ein verschiedenes Verhalten für Typen mit und ohne Vorzeichen aufweist.
- Anstelle von **instanceof** wird der **is**-Operator verwendet.
- Es ist keine benannte **break**-Anweisung vorhanden, diese wird durch **goto** ersetzt.
- Die **switch**-Anweisung verbietet das »Durchfallen« von Code, **switch** kann für Zeichenfolgenvariablen eingesetzt werden.
- Es ist nur eine Arraydeklarationssyntax verfügar: **int[] arr**.
- C# ermöglicht bei Verwendung des **params**-Schlüsselwortes eine variable Anzahl Parameter.

35.3 Unterschiede zwischen C# und Visual Basic 6

C# und Visual Basic 6 sind relativ unterschiedliche Sprachen. C# ist eine objektorientierte Sprache, Visual Basic 6 bietet nur beschränkte objektorientierte Funktionen. VB7 weist zusätzliche objektorientierte Funktionen auf, daher kann eine Lektüre der VB7-Dokumentation sehr aufschlussreich sein.

35.4 Codeaussehen

In Visual Basic enden Anweisungsblöcke auf eine Art von **END**-Anweisung, und es dürfen sich nicht mehrere Anweisungen in einer Zeile befinden. In C# werden Blöcke mit geschweiften Klammern ({}) gekennzeichnet, und die Position der Zeilenumbrüche spielt keine Rolle, da das Ende einer Anweisung mit einem Semikolon gekennzeichnet wird. Obwohl der nachfolgende Code vielleicht keinen guten Stil darstellt und hässlich zu lesen ist, ist er in C# möglich:

```
for (int j = 0; j < 10; j++) {if (j == 5) Func(j); else return;}
```

Diese Zeile trägt die gleiche Bedeutung wie der folgende Codeabschnitt:

```
for (int j = 0; j < 10; j++)
{
   if (j == 5)
      Func(j);
   else
      return;
}
```

Auf diese Weise wird der Programmierer zwar weniger eingeschränkt, es sind jedoch auch Abkommen bezüglich des Stils erforderlich.

35.4.1 Datentypen und Variablen

Obwohl viele der in VB und C# verwendeten Datentypen übereinstimmen, gibt es einige wichtige Unterschiede, ein ähnlicher Name kann beispielsweise einen anderen Datentyp bezeichnen.

Der wichtigste Unterschied besteht darin, dass C# bei Variablendeklaration und -verwendung strikter ist. Alle Variablen müssen vor der Verwendung deklariert werden, und sie müssen mit einem bestimmten Typ deklariert werden – es ist kein **Variant**-Typ vorhanden, der einen beliebigen Typ enthalten kann[1].

Variablendeklarationen erfolgen einfach durch Einfügen des Typnamens vor der Variablen; es gibt keine **dim**-Anweisung.

Konvertierungen

Konvertierungen zwischen Typen werden in C# ebenfalls strenger gehandhabt als in Visual Basic. C# kennt zwei Arten der Konvertierung, die implizite und die explizite Konvertierung. Implizite Konvertierungen sind diejenigen, bei denen kein Datenverlust auftritt – d. h., der Quellwert passt stets in die Zielvariable. Beispiel:

```
int    v = 55;
long x = v;
```

Das Zuweisen von **v** zu **x** ist möglich, da **int**-Variablen immer in **long**-Variablen passen.

Explizite Konvertierungen dagegen sind Konvertierungen, bei denen ein Datenverlust auftreten kann bzw. die fehlschlagen können. Aufgrund dieser Tatsache muss die Konvertierung mit Hilfe einer Typumwandlung explizit angegeben werden:

```
long    x = 55;
int v = (int) x;
```

Obwohl die Konvertierung in diesem Fall sicher ist, kann **long** Zahlen enthalten, die zu groß sind, um in einen **int**-Wert zu passen, daher ist eine Typumwandlung erforderlich.

1. Der Typ **object** kann einen beliebigen Typ enthalten, es ist jedoch bekannt, welcher Typ enthalten ist.

Wenn das Ermitteln eines Überlaufs während der Konvertierung von Bedeutung ist, kann mit der **checked**-Anweisung die Überlaufermittlung aktiviert werden. Weitere Informationen hierzu finden Sie in Kapitel 15, Konvertierungen.

Datentypenunterschiede

In Visual Basic lauten die ganzzahligen Datentypen **Integer** und **Long**. In C# werden diese durch die Typen **short** und **int** ersetzt. Es ist ebenfalls ein **long**-Typ vorhanden, bei diesem handelt es sich jedoch um einen 64-Bit-Typ (8-Byte). Dies sollten Sie im Hinterkopf behalten, denn wenn Sie in C# dort **Long** einsetzen, wo Sie in Visual Basic **long** verwenden, werden die Programme sehr viel größer und langsamer. **Byte** dagegen kann fast mit **byte** gleichgesetzt werden.

C# verfügt des Weiteren über die Datentypen **ushort**, **uint** und **ulong** (ohne Vorzeichen) sowie den Typ **sbyte**, einen **byte**-Wert mit Vorzeichen. Diese sind in bestimmten Situationen nützlich, funktionieren jedoch nicht in allen weiteren .NET-Sprachen und sollten daher sparsam eingesetzt werden.

Die Gleitkommatypen **Single** und **Double** werden in **float** und **double** umbenannt, und der **Boolean**-Typ wird zu **bool**.

Zeichenfolgen

Viele der integrierten Funktionen von Visual Basic sind für C#-Zeichenfolgentypen nicht verfügbar. Es gibt Funktionen für das Suchen von Zeichenfolgen, das Extrahieren von Teilzeichenfolgen und das Durchführen weiterer Operationen. Siehe hierzu die Dokumentation des **System.String**-Typs.

Die Zeichenfolgenverkettung erfolgt nicht über den **&**-Operator, sondern über **+**.

Arrays

In C# erhält das erste Element eines Arrays immer den Index 0, und es gibt keine Möglichkeit, höhere oder niedrigere Grenzen festzulegen oder ein **redim** für Arrays auszuführen. Über **ArrayList** im Namespace **System.Collection** wird jedoch eine Dimensionierung bereitgestellt, zusammen mit weiteren nützlichen Auflistungsklassen.

35.4.2 Operatoren und Ausdrücke

Die C#-Operatoren weisen gegenüber Visual Basic einige wenige Unterschiede auf, daher müssen Sie sich mit diesen besonders vertraut machen.

VB-Operator	C#-Äquivalent
^	Keiner. Siehe **Math.Pow()**
Mod	%
&	+
=	==
<>	!=
Like	Keiner. **System.Text.RegularExpressions.Regex** erledigt einige dieser Aufgaben, ist jedoch komplexer.
Is	Keiner. Der C#-Operator **is** trägt eine andere Bedeutung.
And	&&
Or	||
Xor	^
Eqv	Keiner. **A Eqv B** entspricht **!(A ^ B)**.
Imp	Keiner.

35.4.3 Klassen, Typen, Funktionen und Schnittstellen

Da C# eine objektorientierte Sprache ist[2], stellen Klassen die hauptsächliche Organisationseinheit dar; Code oder Variablen werden nicht in globalen Bereichen verwaltet, sondern immer mit einer spezifischen Klasse verknüpft. Dies führt zu Code, der recht anders als der Visual Basic-Code strukturiert und organisiert wird. Dennoch gibt es weiterhin Gemeinsamkeiten. Eigenschaften können weiterhin verwendet werden, auch wenn sie eine andere Syntax aufweisen und keine Standardeigenschaften vorhanden sind.

Funktionen

In C# müssen Funktionsparameter einen deklarierten Typ aufweisen, und anstelle von **ByVal** wird mit Hilfe von **ref** angegeben, dass der Wert einer übergebenen Variable bearbeitet werden kann. Die Funktion **ParamArray** kann durch Verwenden des **params**-Schlüsselwortes ausgeführt werden.

35.4.4 Steuerung und Programmfluss

C# und Visual Basic verfügen über ähnliche Steuerungsstrukturen, die verwendete Syntax unterscheidet sich jedoch leicht.

2. Siehe hierzu Kapitel 1, Grundlagen der objektorientierten Programmierung.

If Then

In C# gibt es keine **Then**-Anweisung; nach der Bedingung folgt die Anweisung oder der Anweisungsblock, die/der bei erfüllter Bedingung ausgeführt werden soll. Im Anschluss an Anweisung oder Anweisungsblock kann eine optionale **else**-Anweisung vorliegen.

Der folgende Visual Basic-Code

```
If size < 60 Then
   value = 50
Else
   value = 55
   order = 12
End If
```

kann umgeschrieben werden zu

```
if (size < 60)
   value = 50;
else
{
   value = 55;
   order = 12;
}
```

In C# gibt es keine **ElseIf**-Anweisung.

For

Die Syntax von **for**-Schleifen ist in C# anders, das Konzept bleibt jedoch dasselbe, abgesehen davon, dass die am Ende einer Schleife durchgeführte Operation in C# explizit angegeben werden muss. Mit anderen Worten, der folgende Visual Basic-Code

```
For i = 1 To 100
   ' Weiterer Code hier
}
kann umgeschrieben werden zu
for (int i = 0; i < 10; i++)
{
   // Weiterer Code hier
}
```

For Each

C# unterstützt die **For Each**-Syntax über die **foreach**-Anweisung, die für Arrays, Auflistungsklassen und weitere Klassen eingesetzt werden kann, die eine geeignete Schnittstelle offen legen.

Do Loop

C# weist zwei Schleifenkonstruktionen aus, die das **Do Loop** ersetzen. Die **while**-Anweisung wird zum Durchlaufen einer Schleife verwendet, während eine Bedingung erfüllt ist, **do while** arbeitet auf die gleiche Weise, abgesehen davon, dass auch ein Schleifendurchlauf vollzogen wird, wenn die Bedingung nicht erfüllt ist. Der folgende Visual Basic-Code

```
I = 1
fact = 1
Do While I <= n
   fact = fact * I
   I = I + 1
Loop
```

kann folgendermaßen umgeschrieben werden:

```
int I = 1;
int fact = 1;
while (I <= n)
{
   fact = fact * I;
   I++;
}
```

Eine Schleife kann mit Hilfe der **break**-Anweisung verlassen oder im nächsten Durchlauf mit der **continue**-Anweisung fortgesetzt werden.

35.4.5 Select Case

Die **switch**-Anweisung in C# führt die gleiche Aufgabe aus wie **Select Case**. Dieser VB-Code

```
Select Case x
   Case 1
      Func1
   Case 2
      Func2
   Case 3
```

```
      Func2
   Case Else
      Func3
End Select
```

kann folgendermaßen umgeschrieben werden:

```
switch (x)
{
   case 1:
      Func1();
      break;
   case 2:
   case 3:
      Func2();
      break;
   default:
      Func3();
      break;
}
```

35.4.6 On Error

In C# gibt es keine **On Error**-Anweisung. Fehlerbedingungen in .NET werden über Ausnahmen gehandhabt. Weitere Informationen zu diesem Thema finden Sie in Kapitel 4, Ausnahmebehandlung.

35.4.7 Fehlende Anweisungen

In C# sind weder **With**, **Choose** noch ein Äquivalent zu **Switch** verfügbar. Des Weiteren kann auch nicht auf die **CallByName**-Funktion zurückgegriffen werden, auch wenn diese Operation über die Reflektion ausgeführt werden kann.

35.5 Weitere .NET-Sprachen

Visual C++ und Visual Basic wurden beide zur Verwendung in der .NET-Welt erweitert.

In der Visual C++-Welt wurde der Sprache ein Satz »verwalteter Erweiterungen« hinzugefügt, um den Programmierern das Erzeugen und Verwenden von Komponenten für die Common Language Runtime zu ermöglichen. Das Visual C++-Modell stattet den Programmierer mit umfangreicheren Steuerungsmöglichkeiten aus als das C#-Modell, da sowohl verwaltete (Speicherbereinigung) als auch nicht verwaltete Objekte (**new** und **delete**) geschrieben werden können.

Eine .NET-Komponente wird durch das Verwenden von Schlüsselworten zum Bearbeiten der Bedeutung vorhandener C++-Konstruktionen erstellt. Wenn beispielsweise das Schlüsselwort **__gc** einer Klassendefinition vorangestellt wird, ermöglicht dieses Vorgehen die Erstellung einer verwalteten Klasse und verbietet der Klasse die Verwendung von Konstruktionen, die in der .NET-Umgebung nicht ausgedrückt werden können (beispielsweise die Mehrfachvererbung). Von den verwalteten Erweiterungen aus können auch die .NET-Systemklassen verwendet werden.

Visual Basic hat ebenfalls erhebliche Verbesserungen erfahren. Es werden nun objektorientierte Konzepte wie Vererbung, Kapselung und Überladung unterstützt, damit eine nahtlose Integration in die .NET-Umgebung gewährleistet ist.

36 C# und die Zukunft

Wie bereits zu Beginn dieses Buches erwähnt, ist C# eine Sprache, die sich noch in Entwicklung befindet, daher fällt es schwer, Spekulationen über die Zukunft anzustellen – es sei denn, Microsoft hat eine offizielle Meinung zum jeweiligen Thema.

Eine Funktion, an der Microsoft arbeitet, ist eine generische Version der Vorlagen. Wären innerhalb der Sprache generische Vorlagen vorhanden, könnten stark typisierte Auflistungsklassen geschrieben werden, beispielsweise ein Stack, der anstelle von beliebigen Objekten nur einen bestimmten Typ enthält.

Eine solche Stackklasse könnte mit dem **int**-Typ eingesetzt werden, sodass der Stack nur **int**-Werte enthält. Dies hätte zwei große Vorteile:

- Wenn der Programmierer versucht, einen **float**-Wert von einem Stack mit **int**-Werten abzurufen, geben die aktuellen Auflistungen diesen Fehler zur Laufzeit aus. Generische Vorlagen würden einen Fehlerbericht zur Kompilierungszeit ermöglichen.
- In den derzeit verwendeten Auflistungen muss für alle Wertetypen ein Boxing durchgeführt werden, **int**-Werte werden daher nicht als Werteobjekte, sondern als Verweisobjekte gespeichert. Das Hinzufügen und Entfernen dieser Objekte erzeugt Overhead, der bei einer generischen Unterstützung wegfiele.

Wir hoffen, Sie haben interessante Anregungen für Ihre Projekte erhalten.

Die weiteren Kapitel finden Sie im gleichnamigen Buch ...

C#

Die neue Sprache für Microsofts .NET-Plattform

Eric Gunnerson

Galileo Computing
408 S., 2001, geb.
79,90 DM, 40,85 Euro
ISBN 3-934358-107-4