C# und das .NET-Framework

Sebastian Krysmanski
Seminar ViCCC
11. Dezember 2007 (Update: 27. Februar 2008)

Zusammenfassung — Dieser Artikel versucht fortgeschrittenen Java-Programmierern, den Unterschied zwischen Java und C# aufzuzeigen. Dabei wird auch kurz auf das .NET Framework eingegangen.

I. Einleitung

Wer eine Programmiersprache kann, kann alle. Diese Weisheit gilt auch für die Programmiersprache C# - zumindest wenn man bereits einige Erfahrung in Java (oder C++) gesammelt hat. Doch wo genau liegen die Unterschiede zwischen Java und C#? Worauf muss man achten? Welche Stolperfallen gibt es, wenn man Java-Code einfach in einen C#-Compiler hineinsteckt? Und was hat es mit diesem ganzen Thema „.NET" auf sich? Alle diese Fragen finden in diesem Artikel (hoffentlich) eine Antwort.

II. Was ist .NET?

Wenn man in der Windows-Welt zu Hause ist, dann hat man evtl. schon von .NET gehört. Hier und da benötigt eine Anwendung das .NET-Framework 1.1 oder 2.0. Und in Microsofts neuster Windows-Kreation (mit Namen „Vista") ist bereits die Version 3.0 integriert. Doch die meisten Benutzer wissen sicherlich nicht einmal, dass ein solches Framework bei ihnen installiert ist. Deshalb stellt sich die Frage: Was ist dieses .NET eigentlich?

.NET ist nicht nur eine Ansammlung von Klassen – auch wenn die Klassenbibliothek[1] aus mehreren hundert Klassen besteht. .NET ist auch keine Programmiersprache, wie etwa Java. Nein, .NET ist ein Programmierkonzept. Bei diesem Konzept geht es darum, eine gemeinsame „Schnittstelle" für verschiedene Programmiersprachen zur Verfügung zu stellen, so dass Module bzw. Klassen zwischen den einzelnen Programmiersprachen ohne Probleme ausgetauscht werden können.

A. Ein geschichtlicher Abriss

Dieses Konzept wurde der Öffentlichkeit im Juni 2000 zum ersten Mal von Bill Gates vorgestellt. Im selben Jahr wurde dann auch die Programmiersprache C# zur Standardisierung eingereicht, so dass der Veröffentlichung von .NET 1.0 (zusammen mit der Entwicklungsumgebung „Visual Studio .NET 2002") im Januar 2002 nichts mehr im Wege stand (siehe **Tabelle 1**).

Datum	Ereignis
Juni 2000	Bill Gates stellt erstmals die .NET-„Vision" vor.
Oktober 2000	C# wird zur Standardisierung eingereicht
Januar 2002	.NET 1.0 und Visual Studio .NET 2002 werden veröffentlicht
April 2003	.NET 1.1 und Visual Studio 2003 werden veröffentlicht
November 2005	.NET 2.0 und Visual Studio 2005 werden veröffentlicht
November 2006	.NET 3.0 wird veröffentlicht
November 2007	.NET 3.5 und Visual Studio 2008 werden veröffentlicht

Tabelle 1 Geschichtlicher Überblick über .NET

In den darauffolgenden Jahren wurden immer wieder neuere Versionen von .NET veröffentlicht. Im vergangenen Monat (November 2007) war es dann wieder soweit. Mit der Versionsnummer 3.5 wurde die momentan aktuellste .NET-Version freigegeben.

Mit der ersten Veröffentlichung im Jahre 2002 ist .NET noch relativ jung. Im Vergleich dazu: Die Programmiersprache „Java" erschien (zusammen mit dem JDK 1.0) bereits 6 Jahre früher – die Entwicklung selbst reicht bis in das Jahr 1990 zurück.

B. Der Aufbau von .NET

.NET ist – wie bereits erwähnt – nicht nur eine Klassenbibliothek und auch keine Programmiersprache. Aber was ist es dann? Um etwas besser zu verstehen, was .NET ist, soll im Folgenden die Struktur von .NET kurz erklärt werden.

Zunächst einmal setzt sich .NET aus drei Bestandteilen zusammen:

- der .NET-Klassenbibliothek
- der .NET-Laufzeitumgebung
- der Common Language Infrastructure (CLI)

Dieser Aufbau ähnelt dem von Java. Klassenbibliotheken gibt es in beiden Technologien und auch eine Laufzeitumgebung (auch *Virtual Machine*[2] genannt) gibt es sowohl in .NET als auch in Java. Die Besonderheit, die .NET jedoch von Java unterscheidet, kommt mit der sog. *Common Language Infrastructure*[I].

[1] In diesem Fall gleichbedeutend mit „Framework".

[2] Die Virtual Machine macht es möglich, .NET-Anwendung einmal zu kompilieren und sie dann auf jeder beliebigen Plattform einzusetzen, die eine .NET-Laufzeitumgebung zur Verfügung stellt. Microsoft selbst stellt seine .NET-Laufzeitumgebung jedoch nur für Windows und Windows Mobile (ab Version 5.0) zur Verfügung. Es gibt jedoch mit Mono und dotGnu zwei Projekte, die auch eine Laufzeitumgebung für Linux/Unix bereitstellen. Diese Projekte unterstützen bis dato jedoch nur einen geringen Umfang der .NET-Klassenbibliothek. Insbesondere Windows Forms werden nicht unterstützt.

C. *Common Language Infrastructure (CLI)*

Wenn über Java-Anwendungen geredet wird, ist klar: Diese Anwendungen sind in der Programmiersprache Java geschrieben. Wenn jedoch über .NET-Anwendungen gesprochen wird, heißt das: Diese Anwendung ist *für* (und nicht *in*) .NET geschrieben.

Und genau an dieser Stelle kommt die CLI zum Tragen. Um nicht zu sehr ins Detail zu gehen, könnte man vereinfacht sagen: Die CLI definiert eine gemeinsame (engl. *common*) Programmiersprache, die von allen .NET-Sprachen[3] unterstützt[4] wird.

So kann z.B. eine Klasse, die in C# geschrieben ist, von einer Klasse in Visual Basic .NET beerbt werden. Beide Klassen können ohne Probleme Daten miteinander austauschen und Exceptions weiterreichen.

Diese Programmiersprachen-Unabhängigkeit führt in .NET dazu, dass es keine bevorzugte .NET-Sprache gibt. Der Vorteil liegt ganz klar darin, dass jeder seine „Lieblingsprogrammiersprache“ weiterverwenden und trotzdem auf .NET aufbauen kann. Ein Java-Programmierer könnte z.B. die Sprache J# verwenden, um .NET-Anwendungen zu schreiben und müsste nicht/kaum umgewöhnen.

D. *Common Language Runtime (CLR)*

Die Laufzeitumgebung heißt unter .NET *Common Language Runtime*. Es handelt sich dabei – genau wie in Java – um eine Virtual Machine.

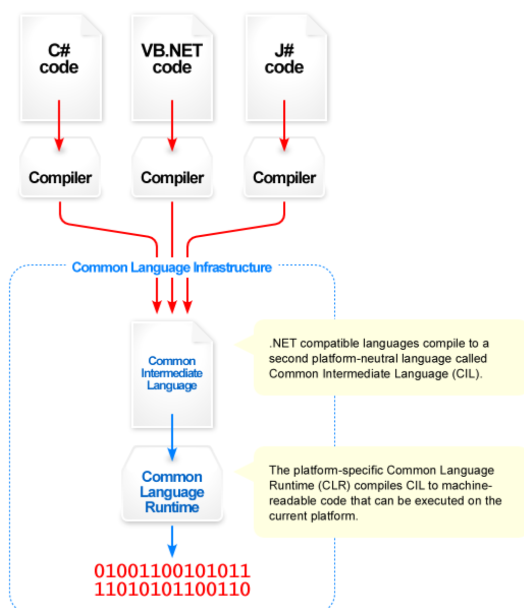

Abbildung 1 Vom Quellcode zur Ausführung[II]

Das heißt, dass es möglich ist, .NET-Anwendung einmalig zu kompilieren und sie dann auf jeder beliebigen Plattform (z.B. Windows oder Linux), die eine .NET-Laufzeitumgebung zur Verfügung stellt, auszuführen. Dieses Konzept ist in **Abbildung 1** abgebildet. Microsoft hat .NET zwar als plattform-unabhängige Technologie konzipiert, stellt selbst seine .NET-Laufzeitumgebung jedoch nur für Windows und Windows Mobile (mittels .NET Compact Framework[III]; ab Pocket PC 2000[IV]) zur Verfügung. Die detaillierteren Systemvoraussetzungen sind in **Tabelle 2** aufgelistet. Genauere Es gibt jedoch mit Mono[5] und dotGnu[6] zwei Projekte, die die Laufzeitumgebung auf Linux/Unix portiert haben. Diese Projekte unterstützen bis dato jedoch nur einen geringen Umfang der .NET-Klassenbibliothek, was ihre Einsetzbarkeit einschränkt. Insbesondere Windows Forms werden nicht (vollständig) unterstützt.

VERSION	VORAUSSETZUNG
.NET 1.1	ab Windows 98 bzw. NT4
.NET 2.0	ab Windows 98 bzw. 2000
.NET 3.0	ab Windows XP SP2
Compact 1.0	ab Pocket PC 2000
Compact 2.0	ab Windows CE .NET

Tabelle 2 .NET-Versionen und ihre Systemvoraussetzungen

Um die Plattform-Unabhängigkeit von .NET zu gewährleisten, müssen .NET-Anwendungen in eine Zwischensprache kompiliert werden: die *Common Intermediate Language* (CIL) – das Äquivalent[7] zum *Bytecode* in Java. Dabei handelt es sich um eine plattform-unabhängige, objektorientierte Assemblersprache. Sie ist sozusagen die „niedrigste“ Stufe der .NET-Programmiersprachen, die ein Mensch noch lesen kann. Danach kommt direkt der .NET-Bytecode, in den jeder CIL-Code übersetzt wird. Ein „Hello World!“-Programm sieht in CIL ist in **Code-Listing II-1** dargestellt.

```
.method public static void Main() cil managed
{
    .entrypoint
    .maxstack 1
    ldstr "Hallo Welt!"
    call void[mscorlib]System.Console::WriteLine(string)
    ret
}
```

Code-Listing II-1 "Hello World!" in CIL

Allerdings müssen .NET-Anwendungen CIL nicht verwenden. Ein Beispiel dafür ist die direkt von Microsoft Sprache C++/CLI (eine Spracherweiterung zum klassischen C++). Anwendungen, die in dieser Programmiersprache geschrieben werden, werden häufig[8] direkt in Maschinencode übersetzt.

III. DIE ENTWICKLUNGSUMGEBUNG VISUAL STUDIO

Um .NET jetzt produktiv nutzen zu können, empfiehlt es sich, eine integrierte Entwicklungsumgebung (IDE) zu verwenden. Zu diesem Zweck stellt Microsoft sein Produkt „Visual Studio“ bereit. Diese bringt alles mit, was man für die Entwicklung von .NET-Anwendungen benötigt.

Es gibt von Visual Studio mehrere Editionen, die auf die verschiedenen Entwickleransprüche zugeschnitten sind. Die

[3] Microsoft stellt die Sprachen *C#*, *J#*, *C++/CLI* und *Visual Basic .NET* zur Verfügung. Eine Auflistung weitere Sprachen findet sich hier: http://de.wikipedia.org/wiki/Liste_der_.NET-Sprachen

[4] Da nicht alle Programmiersprachen die gleichen Programmierkonstrukte unterstützen, ist die CLI nur als Teilmenge der unterstützen Sprachen definiert. So sind z.B. vorzeichenlose Datentypen zwar innerhalb der Implementierung, jedoch nicht an öffentlichen Schnittstellen erlaubt.

[5] http://www.mono-project.com

[6] http://dotgnu.org/

[7] Vereinfacht gesprochen. CIL-Code wird ebenfalls in einen Bytecode übersetzt, jedoch wird Java-Code *direkt* in Bytecode übersetzt. Es gibt unter Java also keine Zwischensprache, die ein Äquivalent für CIL sein könnte.

[8] Sie können jedoch auch in CLI oder eine Mischform übersetzt werden, in der sowohl Maschinencode als CLI vorkommt.

drei wichtigsten[9] Editionen seien hier einmal kurz vorgestellt.

A. *Kostenpflichtige Editionen*

Zwei dieser Editionen sind kostenpflichtig. Es handelt sich dabei um die *Professional Edition* und die *Standard Edition*. Diese beiden Editionen bieten in etwa den gleichen Funktionsumfang, unterscheiden sich aber in einigen Details. U.a. bietet die *Professional Edition* zusätzlich Remote Debugging, XSLT-Unterstützung und eine Integration des Microsoft SQL Servers 2005, der der Professional Edition als *Developer Edition* (und nicht als *Express Edition* wie bei der Standard Edition) beigelegt ist. Für mehr Details sei der interessierte Leser sei an dieser Stelle auf die deutschen Produktseite[V] von Visual Studio verwiesen.

B. *Die kostenlose Edition*

Microsoft bietet auch eine kostenlose Edition von Visual Studio an: die *Express Edition*.

Die Express Edition gibt es in verschiedenen Ausgaben: für jede .NET-Sprache (auch C#) einzeln. D.h. es gibt *keine* Ausgabe, die alle vier Sprachen enthält. Man muss sich jede Ausgabe einzeln herunterladen. Möchte man auf alle Sprachen in einem Paket zugreifen, muss eine der kostenpflichtigen Editionen erworben werden.

Was den Leistungsumfang der Express Edition betrifft, so muss man hier Abstriche hinnehmen, die jedoch den „normalen" Hobby-Entwickler, der einfach nur .NET-Anwendungen für Windows entwickeln will, nur am Rande tangiert. Die markantesten Unterschiede sind:

- Man kann keine Software für mobile Geräte (PDA, Smartphone) entwickeln.
- Es gibt keine 64-Bit-Compiler-Unterstützung.
- Es können keine ASP.NET-Anwendung und keine Webdienste erstellt werden.
- Es können keine Windows-Dienste und keine Windows-Steuerelemente erstellt werden.

C. *Ausblick*

Momentan sind zwei Versionen von Visual Studio, die es jeweils in den beschriebenen Editionen gibt, aktuell bzw. verbreitet:

- *Visual Studio 2005:* Diese Version wurde vor drei Jahren veröffentlicht und ist wurde Mitte Februar 2008 durch Visual Studio 2008 abgelöst.
- *Visual Studio 2008:* Diese Version ist die neuste Version und wurde Mitte Februar 2008 offiziell auch auf Deutsch vorgestellt.

Im Gegensatz zu Visual Studio 2005 bietet die neuer 2008er Version hauptsächlich Neuerungen und Verbesserungen im Bereich „Webentwicklung". Außerdem wird die neue .NET-Version 3.5 unterstützt. Genauere Information finden sich auf den offiziellen Visual Studio Seiten von Microsoft[VI].

IV. Eine kurze Einführung in C#

Anhand eines klassischen „Hello World!"-Programms soll in diesem Abschnitt kurz gezeigt werden, wie Anwendungen in Visual Studio (hier die *Express Edition C# 2005*) erstellt werden. Danach wird noch illustriert, wie man eine zweite Klasse zu einem Visual Studio Projekt hinzufügt, und zum Abschluss auf die Struktur einer C#-Datei eingegangen.

A. *Hello World!*

Dazu starten wir zunächst einmal Visual Studio. Danach präsentiert sich Visual Studio in etwa so, wie in **Abbildung 2** dargestellt.

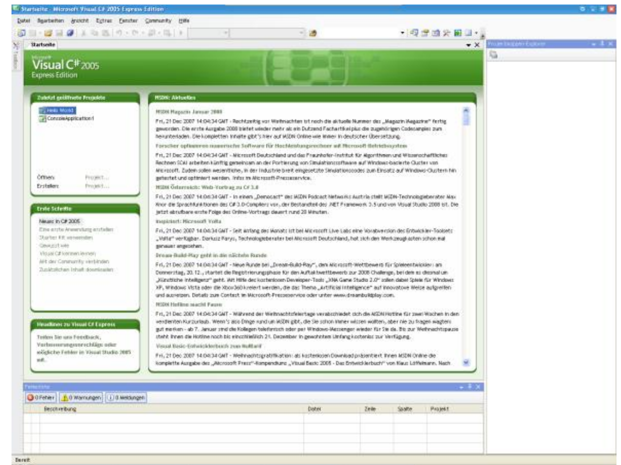

Abbildung 2 Visual-Studio-Startbildschirm

Um ein neues Projekt zu erstellen, klickt man oben links auf den Knopf „Neues Projekt". Daraufhin öffnet sich das Dialogfeld „Neues Projekt", in dem man den gewünschten Projekttyp auswählt – in unserem Fall wählen wir die Vorlage „Konsolenanwendung" und geben dem neuen Projekt den Namen „Hello World" (siehe **Abbildung 3**).

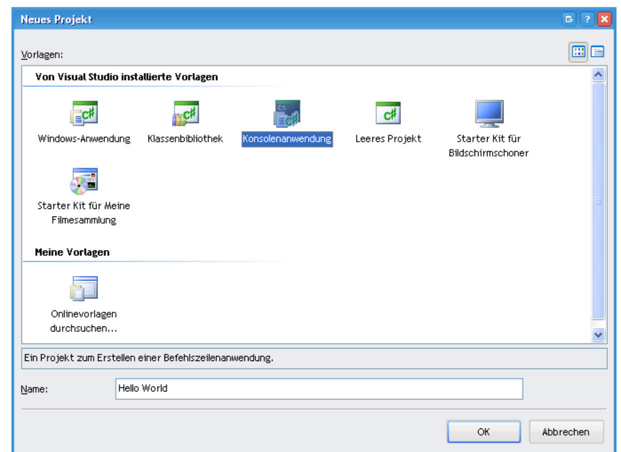

Abbildung 3 Neues Projekt erstellen

Nach einer kurzen Rechenzeit hat Visual Studio das Projekt erstellt und zeigt den Quellcode für die neue Anwendung an. Jetzt erweitern wir die `Main`-Methode um die Codezeile, die „Hello World!" auf der Konsole ausgibt. Damit sieht der gesamte Quellcode der Datei „Program.cs" wie in **Code-Listing IV-1** dargestellt aus.

```
using System;
using System.Collections.Generic;
using System.Text;
```

[9] Aus Sicht des Autors

```
namespace Hello_World
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Code-Listing IV-1 Vollständiges "Hello World!"-Programm

Jetzt kann die Anwendung mit der Tastenkombination *Strg+F5* gestartet werden und das Ergebnis unseres ersten C#-Programms wird sichtbar (**Abbildung 4**).

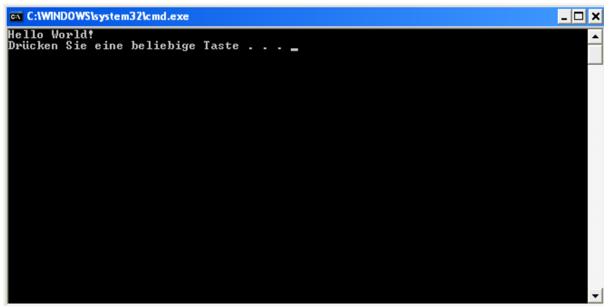

Abbildung 4 Ausgabe auf der Konsole

B. *Eine weitere Klasse*

Um dem Projekt eine weitere Klasse hinzuzufügen, wählt man aus dem Menü „Projekt" und dann „Neue Klasse hinzufügen…" aus. Im daraufhin erscheinenden Dialog „Neues Element hinzufügen" wählt man die Option „Klasse" aus, gibt einen Namen für sie ein (hier: „Class1.cs") und klickt auf „OK". Ein neues Quellcodefenster öffnet sich, dessen Inhalt wir durch den Code aus **Code-Listing IV-2** ersetzen.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Hello_World
{
    class Class1
    {
        public void test()
        {
            Console.WriteLine("Hello C sharp!");
        }
    }
}
```

Code-Listing IV-2 Eine zweite Klasse

Den Inhalt der `Main`-Methode in der Klasse `Program` wird dann noch wie in **Code-Listing IV-3** abgeändert.

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");

    Class1 anotherClass = new Class1();
    anotherClass.test();
}
```

Code-Listing IV-3 Veränderte Main-Methode

Wird das Programm jetzt ausgeführt, erscheint erwartungsgemäß auf der Konsole die in **Code-Listing IV-4** dargestellte Ausgabe.

```
Hello World!
Hello C sharp!
```

Code-Listing IV-4 Ausgabe der Konsole

C. *Die Main-Methode*

In dieser Stelle seien noch ein paar Worte zur `Main`-Methode gesagt:

- Der Name der `Main`-Methode lautet `Main` (mit großem M; und nicht `main` wie in *Java*).
- Der Rückgabewert der `Main`-Methode muss entweder `void` oder `int` sein. Andere Rückgabewerte sind nicht erlaubt[10].
- Die Parameterliste der `Main`-Methode besteht entweder aus einem `string`-Array oder ist leer.
- Die Sichtbarkeit der Main-Methode ist egal. (In *Java* muss sie `public` sein.) Es wird jedoch empfohlen, die Methode `private` zu machen.
- Enthält ein Projekt mehrere `Main`-Methoden, so muss eine ausgewählt werden, damit der Startpunkt der Anwendung eindeutig festgelegt ist. Unter Visual Studio geschieht das über das Menü „Projekt" und dann „Eigenschaften…". Die Klasse, die die entsprechende Main-Methode enthält, kann dann unter „Startobjekt" ausgewählt werden.

D. *Die Struktur einer C#-Datei*

C#-Dateien haben einen etwas anderen Aufbau als Java-Dateien. Es gibt zwei größere Unterschiede.

1) *Namespaces – das etwas andere Package*

Während der Aufbau einer einzelnen Klasse in C# dem Aufbau einer Klasse in Java relativ ähnlich ist (siehe unten), ist das ganze Drumherum anders.

C#-Klassen werden in sog. Namensräumen (engl. *namespaces*) gruppiert. Das Prinzip entspricht in etwa dem, was unter Java ein *Package* ist; es gibt allerdings ein paar Unterschiede:

- *Namensräume müssen explizit angegeben werden.* Im oben beschriebenen „Hello World!"-Beispiel heißt der Namensraum `Hello_World`. Er umschließt die Klasse `Program`. Das zeigt an, dass sich die Klasse `Program` im Namensraum `Hello_World` befindet. Der Namensraum kann jedoch auch *weggelassen* werden. In diesem Fall würde sich die Klasse Program im *globalen Namensraum* (der keinen Namen hat) befinden.
- *Namensräume sind nicht an die Verzeichnisstruktur der Quellcodedateien gebunden.* Namensräume können sich – genau wie Packages – über mehrere Dateien erstrecken. Anders als Packages können sich Namensräume jedoch auch über mehrere Verzeichnisse erstrecken; und die Namen der Verzeichnisse müssen nichts mit den Namen der Namensräume, die innerhalb[11] der Verzeichnisse definiert sind, zu tun

[10] Compilerwarnung CS0028

[11] D.h. innerhalb der Quellcode-Dateien, die in eben diesen Verzeichnissen liegen.

haben.

Um eine Klasse, die sich nicht im selben Namensraum wie die momentan „aktive“ Klasse befindet, zu verwenden, gibt man – genau wie in *Java* – einfach den voll-qualifizierten Namen der Klasse an. Um z.B. die Klasse `DirectoryNotFoundException` aus dem Namensraum `System.IO` zuzugreifen, verwendet man `System.IO.DirectoryNotFoundException`.

Da das auf Dauer etwas mühselig werden kann, gibt es das Schlüsselwort `using`. Es entspricht dem Java-Schlüsselwort `import`, jedoch mit dem Unterschied, dass mit `using` ganze Namensräume (und nicht nur einzelne Klassen) eingebunden werden. Würde man also an den Anfang einer Datei die Zeile aus **Code-Listing IV-5** setzen, könnte man auf die Klasse `DirectoryNotFoundException` direkt über ihren Namen (ohne sie voll zu qualifizieren) zugreifen.

```
using System.IO;
```

Code-Listing IV-5 Verwendung von "using"

Es gibt darüber hinaus noch eine Erweiterung gegenüber dem `import`-Schlüsselwort aus *Java*: der *Alias*. Damit ist es möglich, eine Klasse[13] oder einen Namensraum „umzubenennen“. Hierfpr verwendet man weiterhin `using`, ergänzt den Befehl aber um den Zuweisungsoperator, wie **Code-Listing IV-6** zeigt.

```
using con = System.Console;
...
con.WriteLine("Hello World!");
```

Code-Listing IV-6 Verwendung eines Alias'

Allerdings führt das Verwenden von Aliasen zu schlecht wartbarem Code, weshalb diese Variante nur dort verwendet werden sollte, wo es unbedingt notwendig ist.

Noch ein *Hinweis* zu Klassen[13] ohne Namensraum: Möchte man aus dem Namensraum *A* heraus auf eine Klasse zugreifen, die keinem Namensraum angehört, dann verwendet man einfach ihren Namen. Gibt es im Namensraum *A* aber eine gleichnamige Klasse, so würde diese statt der „globalen“ Klasse verwendet werden. Um trotzdem auf die „globale“ Klasse zugreifen zu können, muss man (bei der Verwendung) dem Klassennamen `global::` voran stellen[12]. Damit teilt man dem C#-Compiler mit, dass die „globale“ Klasse (und nicht die im eigenen Namensraum) gemeint ist.

2) Die Anzahl der Klassen

Ein weiterer Unterschied gegenüber Java betrifft die Anzahl der öffentlichen Top-Level-Klassen[13] innerhalb einer Datei. Während in Java pro Datei nur eine öffentliche Top-Level-Klasse[14] erlaubt ist und der Dateiname dem Namen der Klasse gleichen muss, gibt es in C# eine solche Beschränkung nicht.

Jede Datei kann beliebig viele öffentliche Top-Level-Klassen, -Schnittstellen u.ä. aufnehmen. Es können sogar beliebig Namensräume pro Datei nebeneinander definiert werden.

V. Assemblies – das Kompilat

Wird .NET-Quellcode übersetzt, dann entsteht ein sog. *Assembly* (vergleichbar mit einer Jar-Datei unter Java). Je nach gewähltem Projekttyp hat das Assembly die Form einer EXE-Datei (sog. *process assembly*) oder einer DLL-Datei (sog. *library assembly*). Jedoch unterscheidet sich die entstehende Datei von „normalen“ EXE- oder DLL-Dateien, denn sie enthält keinen[15] plattform-abhängigen Code mehr. Stattdessen bestehen sie aus Bytecode und Metadaten.

A. Metadaten

Dass fast alle .NET-Sprachen in Bytecode[16] übersetzt werden, wurde weiter oben bereits erklärt. Das Problem bei kompiliertem Code ist jedoch, dass die Struktur des ursprünglichen Quellcodes nicht mehr wirklich rekonstruierbar ist[17]. Damit aber z.B. ein library assembly von einer .NET-Anwendung verwendet werden kann, benötigt diese Informationen über den Aufbau der Klassen, Schnittstellen etc. in dem Assembly. Und genau diese Informationen werden in den sog. *Metadaten* gespeichert. Diese werden automatisch bei Kompilieren erzeugt und mit in das Assembly gespeichert.

Metadaten beschreiben in .NET alle Klassen und Klassenelemente, die in einem Assembly definiert sind[18]. Außerdem wird gespeichert, welche Klassen und Klassenelemente von dem Assembly aus verwendet bzw. aufgerufen werden. Die Metadaten für eine Methode enthalten z.B. die Klasse, zu der die Methode gehört, den Typ des Rückgabewerts und die Parameterliste.

Wenn der CLR (Common Language Runtime) „mitgeteilt“ wird, dass sie eine Methode aufrufen soll, wird als erstes geprüft, ob die Metadaten der auszuführenden Methode mit denen übereinstimmen, die zur Übersetzungszeit aktuell waren. Dadurch wird sichergestellt, dass die Methode auch tatsächlich mit den richtigen Parametern aufgerufen wird[19].

B. Assembly-Typen

Neben der Unterscheidung zwischen process assemblies und library assemblies gibt es noch eine weitere Einteilung, die hauptsächlich[20] die library assemblies betrifft: den „Nutzerkreis“. Es gibt:

- *Private Assemblies:* Dieses liegen im gleichen Ver-

[12] Z.B. `global::GlobaleKlasse.StatischeMethode()`
[13] Oder ähnlicher Strukturen wie Interfaces oder Aufzählungstypen.
[14] Gemeint ist die äußerste Klasse. Es gibt in Java natürlich die Möglichkeit weitere Klassen als innere Klassen zu definieren.
[15] Process assemblies enthalten eine kleine Routine, die die CLR startet. Diese ist plattform-abhängig, der restliche Code des Assemblies jedoch nicht.
[16] Eigentlich erst in CIL-Code und dieser dann in Bytecode.
[17] Wer einmal versucht hat, ein Binary zu dekompilieren, weiß, wovon hier die Rede ist.
[18] Das sind die Elemente, die nicht die Sichtbarkeit *internal* haben. Siehe VI.B Sichtbarkeitsmodifizierer.
[19] Das ist vor allem sinnvoll, wenn die aufgerufene Methode in einem anderen Assembly, also in einer anderen Datei, liegt – denn diese Datei kann ausgetauscht werden, was zu einer unterschiedlichen Methodensignatur führen kann.
[20] Seit .NET 2.0 können process assemblies aber auch wie library assemblies benutzt werden.

zeichnis wie die Anwendung selbst. Die CLR geht bei diesem Typ davon aus, dass Version des Assemblies mit der Anwendung kompatibel ist und wird deshalb nicht geprüft.

- *Gemeinsame (shared) Assemblies:* Diese werden von mehreren Anwendungen verwendet und liegen in einem zentralen Verzeichnis.
- *Globale Assemblies:* Sind genau wie gemeinsame Assemblies, liegen allerdings an einem festgelegten Ort[21] und meistens in mehreren Versionen vor.

VI. Gemeinsamkeiten mit Java

Dieser und der nächste Abschnitt (VII) sollen die Unterschiede zwischen C# und Java aufzeigen. Dabei geht es in diesem Abschnitt um Programmierkonstrukte, die sowohl in Java als auch in C# (zumindest in ähnlicher Form) existieren. Teilweise können die Konstrukte nicht eindeutig dem einen oder anderen Abschnitt zugeordnet werden. Der Autor hat hierbei versucht, die Aufteilung möglichst sinnvoll zu treffen.

Neben verschiedenen, syntaktischen Unterschieden zu Java gibt es einen sehr wichtigen Unterschied, der im Abschnitt VI.*C Polymorphie* beschrieben ist.

A. Der Klassenaufbau

Der Aufbau einer C# Klasse entspricht ungefähr dem einer *Java*-Klasse (siehe **Code-Listing VI-1**).

```
public class Beispiel
{
    private int m_iEineVariable;
    private string m_strNochEineVariable = "Hallo";

    public Beispiel(int p_iWert)
    {
        m_iEineVariable = p_iWert;
    }

    public int getWert()
    {
        return this.m_iEineVariable;
    }
}
```

Code-Listing VI-1 Normaler Aufbau einer Klasse in C#

Es gibt genau wie in *Java* sowohl Felder (Variablen und Konstanten) als auch Methoden. Neben diesen klassischen Elementen gibt noch Erweiterungen wie Eigenschaften (siehe Abschnitt VII.D), Indexer oder (überladene) Operatoren, von denen einige weiter unten beschrieben werden.

Es ist jedoch nicht alles gleich. Gegenüber *Java* gibt es zwei syntaktische Unterschiede.

1) Aufruf von Konstruktoren

Der Aufruf von Konstruktoren wird in C# anders notiert. Möchte man in Java einen anderen Konstruktor – z. B. der Basisklasse – aufrufen, fügt man den Befehl `super(...)` als ersten Befehl in den „eigenen“ Konstruktor. Unter C# wird dieser Aufruf – genau wie in C++ – hinter die Parameterliste geschrieben. Also:

- Konstruktor der gleichen Klasse (**Code-Listing VI-2**)

```
public Konstruktor(int wert, ...) : this(wert)
{ ... }
```

Code-Listing VI-2 Aufruf eines Konstruktors der gleichen Klasse

- Konstruktor der Basis-Klasse (**Code-Listing VI-3**)

```
public Konstruktor(int wert, ...) : base(wert)
{ ... }
```

Code-Listing VI-3 Aufruf eines Konstruktors der Vaterklasse

Der Aufruf des Basis-Klassen-Konstruktors zeigt auch ein „neues“ Schlüsselwort: `base`. `base` liefert eine „Referenz“ auf die Basisklasse zurück. Es entspricht also 1-zu-1 dem Java-Schlüsselwort `super`. D. h. mit `base` kann man insbesondere auch auf die Methoden der Basisklasse zugreifen.

Einen weiteren Syntax-Unterschied gibt es noch beim statischen Konstruktor einer Klasse. Während in Java einfach nur das Schlüsselwort `static` gefolgt von der öffnenden, geschweiften Klammer verwendet wird, muss unter C# noch der Name der Klasse gefolgt von einer leeren Parameterliste notiert werden (siehe **Code-Listing VI-4**).

```
static KlassenName() { ... }
```

Code-Listing VI-4 Signatur eines statischen Konstruktors

2) Ableiten und Implementieren

Auch beim Ableiten von Basisklassen bzw. beim Implementieren von Schnittstellen gibt es eine syntaktische Änderung. Statt den Schlüsselwörtern `extends` und `implements` aus Java wird in C# – in Anlehnung an C++ – ein einfacher Doppelpunkt verwendet. Die Basisklasse und/oder die Schnittstellen werden dann einfach als kommagetrennte Liste nach dem Doppelpunkt aufgeführt (**Code-Listing VI-5**).

```
class Klasse : BasisKlasse, Interface1, Interface2
```

Code-Listing VI-5 Ableiten und Implementieren in C#

Hinweis: Genau wie in Java ist *Mehrfachvererbung*[VII] auch in C# verboten. Es können jedoch stets mehrere Schnittstellen implementiert werden.

B. Sichtbarkeitsmodifizierer

Sichtbarkeitsmodifizierer modifizieren, wie der Name schon sagt, die Sichtbarkeit eines (Klassen-)Elements. Es handelt sich dabei um die klassischen Vertreter `public`, `protected` und `private`. Diese gibt es sowohl in C# als auch in Java.

Jedoch würde dieser Abschnitt in diesem Artikel nicht auftauchen, wenn es nicht doch mindestens einen Unterschied zu Java geben würde. Und es gibt ihn: `internal`. Dieser Modifizierer ersetzt den Package-Modifizierer[22] aus

[21] Dem sog. Global Assembly Cache (GAC). Unter Windows heißt dieser Ordner „assembly“ und befindet sich direkt im Windows-Verzeichnis.

[22] Ein Element hat unter Java Package-Sichtbarkeit, wenn man den Sichtbarkeitsmodifizierer weglässt. Es gibt also kein explizites Schlüsselwort für diese Sichtbarkeitsstufe. Übrigens: Lässt man in C# den Sichtbarkeitsmodifizierer weg, ist das Element automatisch `private`.

Java – allerdings mit Änderungen:

- `internal`-Sichtbarkeit bedeutet, dass das Element nur innerhalb des Assemblies, also innerhalb der dll- oder exe-Datei, sichtbar ist.
- Hat ein Element `internal`-Sichtbarkeit, dann wird es innerhalb des Assemblies als `public` angesehen. Manchmal möchte man die Sichtbarkeit jedoch noch weiter einschränken. Zu diesem Zweck kann man `internal` mit `protected` kombinieren (was dann zu `protected internal` wird).
- Es gibt *keine* Namensraum-Sichtbarkeit, d.h. dass ein Element nur innerhalb seines Namensraums sichtbar ist. (Eine solche Sichtbarkeit würde der Package-Sichtbarkeit[22] aus Java entsprechen.)

C. Polymorphie

Dieser Abschnitt befasst sich mit einem *wichtigen* Unterschied gegenüber Java: der *Polymorphie*. Zur Erinnerung: Polymorphie ist das Konzept des Ableitens von Klassen.

1) Bestandsaufnahme

In C# funktioniert genau dieses Ableiten anders als in Java. Um den Unterschied aufzuzeigen, sei das Beispiel in **Code-Listing VI-6** gegeben.

```
class KlasseA
{
    public void Methode()
    {
        Console.WriteLine("Methode der Klasse A");
    }
}
```

Code-Listing VI-6 Eine Beispielklasse

Diese Klasse besitzt genau eine Methode (namens `Methode`), die auf der Konsole ausgibt, woher sie stammt. Erweitern wir nun den Quellcode um eine weitere Klasse (**Code-Listing VI-7**).

```
class KlasseB : KlasseA
{
    public void Methode()
    {
        Console.WriteLine("Methode der Klasse B");
    }
}
```

Code-Listing VI-7 Eine abgeleitete Klasse

Diese zweite Klasse ist von der ersten Klasse (`KlasseA`) abgeleitet und implementiert ebenfalls die Methode `Methode` mit der identischen Methodensignatur.

Jetzt erzeugen wir (an einer anderen Stelle im Code) eine neue Instanz der Klasse `KlasseB` und führen `Methode()` aus (**Code-Listing VI-8**).

```
// Aufruf-Code
KlasseB obj = new KlasseB();
obj.Methode();
```

Code-Listing VI-8 Aufrufen von Methode()

Das Ausführen dieses Codes liefert – wie erwartet – die Ausgabe aus **Code-Listing VI-9** auf der Konsole.

```
Methode der Klasse B
```

Code-Listing VI-9 Ausgabe auf der Konsole

Jetzt ändern wir den Aufruf-Code von oben ab und speichern die Instanz der Klasse `KlasseB` nicht mehr in einer Variable der `KlasseB`, sondern in einer Variable der `KlasseA` (**Code-Listing VI-10**).

```
// Aufruf-Code
KlasseA obj = new KlasseB();
obj.Methode();
```

Code-Listing VI-10 Veränderter Aufruf von Methode()

Wir erzeugen also weiterhin eine Instanz der Klasse `KlasseB`, speichern sie aber nur anders ab. Würden wir diesen Code nun als Java-Code ausführen, wäre die Ausgabe auf der Konsole identisch mit der vorherigen Ausgabe aus **Code-Listing VI-9**.

Wir führen den Code jetzt aber als C#-Code aus und erhalten die in **Code-Listing VI-11** abgebildete, (evtl.) unerwartete Ausgabe.

```
Methode der Klasse A
```

Code-Listing VI-11 Konsolenausgabe bei C#-Code

Es wurde die Methode der Klasse `KlasseA` (und nicht die Methode der Klasse `KlasseB`) aufgerufen. Woran das liegt, wird im folgenden Abschnitt erklärt.

2) Java-Verhalten wiederherstellen

In C# ist ein Methodenaufruf *standardmäßig* an den Typ der Variable (hier `KlasseA`) und nicht an den Typ der Referenz, auf die die Variable zeigt, (hier `KlasseB`) gebunden.

Es ist jedoch möglich, das gleiche Verhalten wie in Java wiederherzustellen. Um das zu erreichen muss der Quellcode wie in **Code-Listing VI-12** dargestellt abgeändert werden.

```
class KlasseA
{
    public virtual void Methode()
    {
        Console.WriteLine("Methode der Klasse A");
    }
}

class KlasseB : KlasseA
{
    public override void Methode()
    {
        Console.WriteLine("Methode der Klasse B");
    }
}
```

Code-Listing VI-12 Wiederherstellung des Java-Verhaltens

Was hat sich geändert? Der Methode in Klasse `KlasseA` wurde das Schlüsselwort `virtual` hinzugefügt. Mit diesem Schlüsselwort teilt man dem C#-Compiler mit, dass diese Methode überladen werden kann.

Der Methode der Klasse `KlasseB` wurde um das Schlüsselwort `override` ergänzt. Dieses Schlüsselwort teilt dem C#-Compiler mit, dass die Basis-Methode auch tatsächlich überladen werden soll. Das ist notwendig, da der

C#-Compiler standardmäßig annimmt, dass eine Methode nicht überladen werden soll. (Man kann dieses Verhalten auch manuell herstellen, indem man das Schlüsselwort `new` statt `override` verwendet. Dieses Vorgehen wird sogar empfohlen, wenn man eine „virtuelle“ Methode nicht überladen möchte, auch wenn der Compiler das Schlüsselwort `new` gar nicht benötigt.)

Ruft man mit diesen Codeänderungen das Programm wieder auf, erhält man die gewünschte Konsolen-Ausgabe (**Code-Listing VI-13**).

```
Methode der Klasse B
```

Code-Listing VI-13 Ausgabe des veränderten C#-Codes

Zusammenfassend kann man also sagen: In C# muss man (mit `virtual`) explizit festlegen, welche Methoden überladen werden dürfen. In Java muss man (mit `final`) explizit festlegen, welche Methoden *nicht* überladen werden dürfen. (Das Pendant zu `final` in C# heißt `sealed`; siehe nächster Abschnitt.)

3) Die Überladungshierarchie kontrollieren

Im vorherigen Abschnitt wurden bereits die beiden Schlüsselwörter `new` und `override` vorgestellt, mit denen man die „Überladungshierarchie“ kontrollieren kann. Ihre Wirkungsweise soll in diesem Abschnitt noch ein wenig vertieft werden.

a) Überladung verbieten mit new

Es gibt die Möglichkeit, mit dem Schlüsselwort `new` das weitere Überladen einer Methode zu verbieten. Als Beispiel sei der Code aus **Code-Listing VI-14** gegeben.

```
class KlasseA
{
    public virtual void Methode() { ... }
}

class KlasseB : KlasseA
{
    public new void Methode() { ... }
}

class KlasseC : KlasseB
{
    public override void Methode() { ... } // Fehler!
}
```

Code-Listing VI-14 Überladung auf Grund von "new" nicht möglich

Das Verwenden des Schlüsselworts `override` in `KlasseC` führt zu einem Compilerfehler. Der Grund liegt darin, dass durch `new` in `KlasseB` dem Compiler mitgeteilt wurde, dass es sich bei der Methode aus `KlasseB` um eine neue Methode handelt (und diese nichts mehr mit der Methode aus `KlasseA` zu tun hat). Und da die Methode aus `KlasseB` nicht als `virtual` deklariert wurde, kann man sie nicht mehr überladen.

b) Erneut überladen mit new virtual

Wie im vorherigen Abschnitt angedeutet, kann eine Methode, die mit dem Schlüsselwort `new` versehen ist, zusätzlich auch das Schlüsselwort `virtual` erhalten. Damit „eröffnet“ die Methode eine neue Überladungshierarchie. Sie kann jetzt wieder überladen werden (**Code-Listing VI-15**).

```
class KlasseA
{
    public virtual void Methode() { ... }
}

class KlasseB : KlasseA
{
    public new virtual void Methode() { ... }
}

class KlasseC : KlasseB
{
    public override void Methode() { ... } // OK!
}
```

Code-Listing VI-15 Überladung durch "virtual" wieder möglich

Auch wenn dieses Beispiel nicht viel Sinn ergibt, so enthält es doch eine interessante bzw. beachtungswürdige Eigenheit: Welche der drei Methoden wird in dem Code-Abschnitt aus **Code-Listing VI-16** aufgerufen?

```
KlasseA obj = new KlasseC();
obj.Methode();
```

Code-Listing VI-16 Welche Methode wird aufgerufen?

Die Antwort lautet: Die Methode aus `KlasseA`. Auch wenn man beliebig viele Klassen in der Ableitungshierarchie zwischen `KlasseA` und `KlasseB` einfügen würde, die Methode von `KlasseC` oder `KlasseB` würden niemals aufgerufen werden – solange die Methode in `KlasseB` weiterhin als `new` deklariert ist. Das sollte beachtet werden.

c) Überladung verbieten mit sealed

Im Abschnitt *VI.C.2) Java-Verhalten wiederherstellen* wurde bereits das Schlüsselwort `sealed` kurz erwähnt. An dieser Stelle noch ein paar Worte dazu:

`sealed` ist das Pendant zu dem Java-Schlüsselwort `final`. Es hat die gleiche Funktionalität. Mit ihm kann man z.B. bei Klassen verhindern, dass diese beerbt werden können. Ebenso kann man verhindern, dass virtuelle Methoden weiter überladen werden können.

```
class KlasseA
{
    public virtual void Methode() { ... }
}

class KlasseB : KlasseA
{
    public override sealed void Methode() { ... }
}

class KlasseC : KlasseB
{
    public override void Methode() { ... } // Fehler!
}
```

Code-Listing VI-17 Überladen auf Grund von "sealed" nicht möglich

Das Verhalten (siehe **Code-Listing VI-17**) ist mit dem in *Java* identisch. Es sei allerdings noch auf zwei Dinge hingewiesen:

- `sealed` ergibt nur in Zusammenhang mit `override` Sinn. Die Verwendung zusammen mit `new` hingegen ist sinnfrei.

- Der Unterschied zum Verbieten des Überladens mit `new` ist, dass mit `new` die Überladungshierarchie vor[23] der Methode „gekappt" wird – bei `sealed` danach.

D. Ausnahmebehandlung

Auch die Ausnahmebehandlung (Stichwort: *Exceptions*) funktioniert in C# fast genau so wie in Java. Ausnahmen werden mit `throw` geworfen und mit `catch` gefangen. Auch ein `finally`-Block ist möglich. Es gibt allerdings auch zwei erwähnenswerte Unterschiede.

Der erste Unterschied ist eher nebensächlich, aber sollte dennoch erwähnt werden: Eigene Ausnahmen(klassen) sollten nicht direkt von der Klasse `Exception` sondern stattdessen von der Klasse `ApplicationException` abgeleitet werden.

Der zweite Unterschied[24] ist schon schwerwiegender: Ausnahmen müssen in C# *nicht* behandelt[25] werden. D.h. wenn eine Methode eine Ausnahme wirft und diese nirgendwo behandelt wird, so kompiliert der Code doch ohne Fehlermeldungen.

Desweiteren werden Ausnahmen auch nicht in der Methodendeklaration (wie in Java mit `throws`) angegeben. Man kann einer Methode also nicht einfach[26] ansehen, ob und welche Ausnahmen von ihr geworfen werden. Man ist auf die Dokumentation der Methoden angewiesen. Das führt unter Umständen dazu, dass eine Ausnahme nicht behandelt wird, weil keine Kenntnis von ihr besteht. Diese Ausnahme wird dann bis zur obersten Methode durchgereicht und lässt das Programm dann „abstürzen", wie in **Abbildung 5** dargestellt.

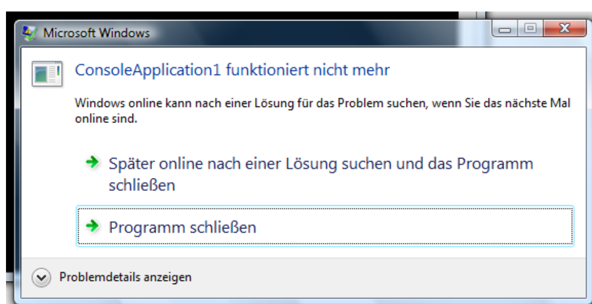

Abbildung 5 Abgestürzte Konsolenanwendung

E. Nebenläufigkeit (Threads)

Für Nebenläufigkeit stehen in C# – genau wie in Java – Threads zur Verfügung. Alle hierfür verwendeten Klassen befinden sich im Namespace `System.Threading`. Die Verwendung eines Threads ist in C# im Großen und Ganzen genauso geregelt wie in Java.

[23] Bildlich gesprochen, wenn man sich die Überladungshierarchie als Graphen (Baum) vorstellt, bei dem Methoden aus Oberklassen näher (=vor) an der Wurzel sind.

[24] In Java müssen zumindest alle „normalen" Exceptions (die nicht von `RuntimeException` oder `Error` abgeleitet sind) gefangen werden.

[25] Meint, dass die Ausnahme entweder „weitergeworfen" oder „konsumiert" wird.

[26] D.h. durch einfaches Anschauen der Methode. Man kann zwar die Ausnahmen erkennen, die direkt in der Methode geworfen werden – kommen jedoch Ausnahmen von Methoden, die innerhalb der Methode aufgerufen werden, wird die Erkennung dieser Ausnahme doch bereits wesentlich schwieriger.

Man legt (durch einen Delegate – siehe Abschnitt VII.E) fest, welcher Code (d. h. eigentlich, welche Methode) in dem Thread ausgeführt werden soll. Diesen teilt man der Klasse Thread mit und startet den Thread dann. (siehe **Code-Listing VI-18**) Der Code wird von Anfang bis Ende ausgeführt und der Thread danach automatisch wieder beendet.

```
public class Program
{
  static void Main(string[] args)
  {
    // Den Thread anlegen
    Thread myFirstThread = new Thread(MyProcedure);
    // Den Thread starten
    myFirstThread.Start();
  }

  // Diese Methode wird in einem
  // eigenen Thread ausgeführt
  public static void MyProcedure()
  {
    ...
  }
}
```

Code-Listing VI-18 Starten eines Threads

Allerdings ist es in C# nicht möglich, die Klasse `Thread` zu beerben, da diese `sealed` ist.

Zur Synchronisierung von Threads gibt es das Schlüsselwort `lock`, das in etwa dem Java-Schlüsselwort `synchronized` entspricht. (siehe **Code-Listing VI-19**) Allerdings kann man – anders als in Java – `lock` nicht als Modifizierer für eine Methode verwenden.

```
public class Sync
{
  // Es wird entweder der Code aus
  // SyncLockA oder SyncLockB ausgeführt.
  // Aber niemals bei "gleichzeitig".
  public void SyncLockA()
  {
    lock(this)
    {
      ...
    }
  }

  public void SyncLockB()
  {
    lock(this)
    {
      ...
    }
  }
}
```

Code-Listing VI-19 Synchronisierung

VII. Erweiterungen gegenüber Java

Neben den Programmierkonstrukten, die in C# und Java gleich oder ähnlich sind, gibt es natürlich auch eine ganze Reihe neuer Konzepte, die es so in Java nicht gibt. Allen vorran kommen sicherlich die sog. Eigenschaften. Aber auch aus C++ entlehnte Konzepte wie Funktionszeiger (sog. Delegate) oder Operatorüberladung gehören zu C#.

Auf Grund der Menge kann allerdings nicht auf alle Konstrukte im Rahmen dieses Abschnitts eingegangen werden. Weitere Verweise zu diesen Themen finden sich im Anhang I weiter unten.

A. Verbessertes String-Handling

Eine sehr häufig bemängelte Eigenschaft von Java ist, dass

der Vergleichsoperator `==` nicht zum Vergleichen von Zeichenketten (engl. *strings*) verwendet werden kann. Dieses Problem besteht unter C# nicht. Das Codebeispiel aus **Code-Listing VII-1** funktioniert unter C# und die Bedingung wird zu `true` ausgewertet, wenn der Inhalt der beiden Zeichenketten (und nicht nur deren Adresse im Speicher) gleich ist.

```
string str1 = "Hallo Welt!";
string str2 = "Hallo " + "Welt!";
if (str1 == str2)
    Console.WriteLine("Gut!");
```

Code-Listing VII-1 Vergleich von Zeichenketten

Eine weitere Verbesserung ist, dass Zeichenketten jetzt in einem `switch` (oder genauer: `case`) Anweisung verwendet werden können (siehe **Code-Listing VII-2**). Zum Vergleich: Unter Java können nur ganzzahlige Werte, Buchstaben und Aufzählungskonstanten in einer `switch` Anweisung verwendet werden.

```
string str = "Ein Test";
switch (str)
{
    case "Hallo Welt":
        break;

    case "Ein Test":
        break;
}
```

Code-Listing VII-2 "switch" Anweisung mit Zeichenketten

Hinweis: Vergleiche[27] von Zeichenketten sind in C# sehr schnell. Der Grund dafür liegt darin, dass Strings mit gleichem Inhalt auch auf den gleichen Speicherbereich zeigen. Somit ist in Wirklichkeit der Vergleichsoperator in C# nicht anders implementiert, sondern der Zuweisungsoperator. Dieser stellt in C# eben diese Eigenschaft von Strings sicher.

Es gibt noch einen weiteren Unterschied, dem Beachtung geschenkt werden sollte. Er betrifft die sog. *Format-Strings*, also den ersten Parameter in Methoden wie `String.Format()` in Java oder `Console.WriteLine()` in C#. Die restlichen Parameter einer solchen Methode nennen wir ab sofort Argumente. Zur Verdeutlichung:

```
public static void WriteLine(string format,
                             params Object[] arg)
```

Code-Listing VII-3 Signatur der Methode "Console.WriteLine"

Der erste Parameter – aus **Code-Listing VII-3** – (`format`) ist der *Format-String*, der zweite Parameter (`arg`) sind[28] die *Argumente*. Damit die Methoden wissen, an welcher Stelle die Argumente in den Format-String eingefügt werden sollen, benötigen sie *Platzhalter*. Und genau bei diesen liegt der Unterschied zwischen Java und C#. Während Java die Platzhalter von C bzw. C++ übernimmt[29], geht man in C# einen anderen Weg. Dazu sei das Beispiel aus **Code-Listing VII-4** gegeben.

```
string strText1 = "C#";
string strText2 = "Spaß";
Console.Write("{0} macht {1}.", strText1, strText2);
// Ausgabe: "C# macht Spaß."
```

Code-Listing VII-4 Format-Strings unter C#

Im Gegensatz zu Java fällt auf: Statt durch den Platzhalter mitzuteilen, welchen Typ das jeweilige Argument hat, wird einfach sein Index benutzt. Unter Java würde z.B. der Platzhalter `%s` der Methode mitteilen, dass es sich bei dem entsprechenden Argument um eine Zeichenkette handelt. Da aber sowieso alle Argumente vom Typ `object` abgeleitet sind[30], lässt man einfach diese entscheiden, wie sie sich darstellen wollen. Es gibt daher keine Notwendigkeit, den Typ des Arguments auch noch zusätzlich im Format-String anzugeben. Die gewählte Schreibweise stellt also eine Vereinfachung gegenüber der traditionellen Schreibweise dar.

Desweiteren macht sie es überflüssig, die Argumente in einer speziellen Reihenfolge notieren zu müssen. Man kann die Argumente in beliebiger Reihenfolge – und sogar mehrfach – verwenden (siehe **Code-Listing VII-5**).

```
string strText1 = "C#";
string strText2 = "Spaß";
Console.Write("{1}, ja wirklich {1}, macht {0}.",
              strText1, strText2);
// Ausgabe: "Spaß, ja wirklich Spaß, macht C#."
```

Code-Listing VII-5 (mehrfache) Verwendung von Argumenten

Manchmal ist es aber dennoch notwendig, den Argumenten ein spezielles Format zu geben. Zu diesem Zweck kann man den Platzhaltern noch weitere Angaben beisteuern. Das vollständige Format eines solchen Platzhalters sieht wie in **Code-Listing VII-6** dargestellt aus.

```
{Index [,Länge][:Format]}
```

Code-Listing VII-6 Syntax der Platzhalter

Wie die eckigen Klammern bereits andeuten, sind die Angaben `Länge` und `Format` optional. Zunächst ein paar Worte zu `Länge`:

Mit `Länge` gibt man die minimale Ausgabelänge des Arguments an. Ist die Standardausgabe des Arguments kürzer, wird der restliche Platz mit Leerzeichen aufgefüllt. Das Beispiel aus **Code-Listing VII-7** erzeugt dabei die Ausgabe aus **Code-Listing VII-8**.

```
int intVar = 10;
Console.WriteLine("Ich kaufe {0,3} Eier", intVar);
Console.WriteLine("Ich kaufe {0,10} Eier", intVar);
```

Code-Listing VII-7 Verschiedene minimale Ausgabelängen

```
Ich kaufe  10 Eier
Ich kaufe         10 Eier
```

Code-Listing VII-8 Ausgabe bei verschiedenen Ausgabelängen

[27] Mit Vergleichsoperator oder `switch` Anweisung.

[28] Die Argumente werden von den Programmiersprachen jeweils automatisch in ein Array umgewandelt. Deshalb hat die Signatur nur zwei Parameter, während bei der Verwendung beliebig viele angegeben werden können.

[29] Verwendung von z.B. `%s` oder `%d`.

[30] Primitive Datentypen wie `int` oder `float` werden durch das sog. Boxing automatisch in Objekte umgewandelt.

Während die erste Ausgabe (aus **Code-Listing VII-8**) eine minimale Länge von drei hat (und somit nur ein Leerzeichen vor der Zehn eingefügt wird), ist die minimale Länge bei der zweiten Zeile zehn. Es werden entsprechend acht Leerzeichen eingefügt. Zu beachten ist allerdings: Es handelt sich bei dieser Zahl um eine Minimal-Länge und *keine Maximal-Länge*. D. h. ist die Ausgabe des Arguments zu lang, dann wird es *vollständig* dargestellt (und nicht einfach abgeschnitten).

Man kann auch einen negativen Wert für `Länge` angeben: In diesem Fall wird die Ausgabe linksbündig (und nicht rechtsbündig wie bei einem positiven Wert) angeordnet.

Neben der Angabe `Länge` gibt es noch `Format`. Diese Angabe hat nur eine Auswirkung auf Zahlen-Argumente wie `int` oder `float`. Die **Tabelle 3** gibt Auskunft über die verfügbaren Formatangaben.

FORMATANGABE	BESCHREIBUNG
C	Zeigt die Zahl im lokalen Währungsformat an.
D	Zeigt die Zahl als dezimalen Integer an.
E	Zeigt die Zahl im wissenschaftlichen Format an (Exponentialschreibweise).
F	Zeigt die Zahl im Festpunktformat an.
G	Eine numerische Zahl wird entweder im Festpunkt- oder im wissenschaftlichen Format angezeigt. Zur Anzeige kommt das »kompakteste« Format.
N	Zeigt eine numerische Zahl einschließlich Kommaseparatoren an.
P	Zeigt die numerische Zahl als Prozentzahl an.
X	Die Anzeige erfolgt in Hexadezimalnotation.

Tabelle 3 Formatangaben für Platzhalter im Format-String

Den meisten Formatangaben kann eine Zahl angehängt werden, die für die Anzahl der auszugebenden Nachkommastellen steht, z.B. `E2`. In **Code-Listing VII-9** sind einige Beispiele für Formatangaben und die aus ihnen resultierenden Ausgaben dargestellt.

```
int intVar = 4711;
Console.WriteLine("intVar={0:C}", intVar);
// Ausgabe: intVar=4.711,00 €

Console.WriteLine("intVar={0:E}", intVar);
// Ausgabe: intVar=4,711000E+003

Console.WriteLine("intVar={0:E2}", intVar);
// Ausgabe: intVar=4,71E+003

int i = 225;
Console.WriteLine("i={0:X}", i);
// Ausgabe: i=E1

float fltVar = 0.2512F;
Console.WriteLine("fltVar={0,10:G}", fltVar);
// Ausgabe: fltVar=    0,2512

Console.WriteLine("fltVar={0:P4}", fltVar);
// Ausgabe: fltVar=25,1200%
```

Code-Listing VII-9 Beispiele für Formatangaben

B. *Partielle Klassen*

Seit .NET 2.0 gibt es die Möglichkeit, Klassen über mehrere Dateien zu verteilen, die sog. *partiellen Klassen*. Dafür wird der Klassensignatur das Schlüsselwort `partial` hinzugefügt. Stößt der C#-Compiler dann auf dieses Schlüsselwort, weiß er, dass zu dieser Klasse evtl. noch weitere Dateien gehören und setzt die Klasse dann aus diesen Teilen zusammen.

```
// in der Quellcodedatei 'ClassA1.cs'
partial class ClassA
{
  public int intX;
  ...
}

// in der Quellcodedatei 'ClassA2.cs'
partial class ClassA
{
  public int intY;
  ...
}
```

Code-Listing VII-10 Partielle Klasse

Die Klasse `ClassA`, die letztendlich beim Übersetzen der beiden Quellcodedateien (siehe **Code-Listing VII-10**) erstellt wird, enthält in unserem Fall sowohl die Variable `intX` als auch die Variable `intY`.

Jetzt stellt sich natürlich die Frage: Wofür benötigt man ein solches Konzept? Bei der Erstellung einer Klasse ist es sicherlich ratsam, den Code in einer Datei zusammen zu halten[31]. Aus diesem Grund wird dieses Konzept bei der „normalen“ Implementierung einer Klasse wahrscheinlich nicht so häufig zum Einsatz kommt. Bei der automatischen Code-Generierung hingegen werden die Vorteile sichtbar.

Man stelle sich folgendes Szenario vor: Es gibt einen Editor, mit dem man sich grafische Oberflächen zusammenklicken kann. Hat man seine Oberfläche fertig, drückt man auf einen Knopf und der Editor erstellt den passenden C#-Code. Jetzt ergänzt man die Klasse um eigenen Code, um die Funktionalität des gerade erzeugten Dialogs bereitzustellen. Nach einiger Zeit möchte man nun weitere Elemente zu dem Dialog hinzufügen. Man öffnet den Editor und fügt diese hinzu. Doch was jetzt? Wenn man den Editor jetzt anweist, den passend C#-Code zu generieren, bleibt ihm entweder die Möglichkeit, den vorhandenen Code komplett zu überschreiben, oder den vorhandenen Code zu parsen und an bestimmten Stellen den Code einzusetzen. Da die erste Variante zum Verlust des eigenen Codes führt und die zweite Variante evtl. fehlerträchtig ist, bieten partielle Klassen in diesem Bereich eine große Vereinfachung. Der Editor erzeugt einfach seinen Code in einer Datei und der Programmierer seinen Code in einer anderen Datei. Der C#-Compiler setzt die Klasse dann automatisch zusammen und man hat einen funktionierenden, leicht zu modifizierenden Dialog.

Es gibt allerdings eine kleine „Einschränkung“ für partielle Klassen: Die Teile müssen sich allesamt in einem Assembly befinden. Es gibt also nicht die Möglichkeit, die eine Hälfte einer partiellen Klasse in einer DLL-Datei zur Verfügung zu stellen und die andere Hälfte dann in einer EXE-Datei, die diese DLL-Datei verwendet, zu implementieren.

C. *Strukturen*

Strukturen sind – vereinfacht gesagt – Klassen, deren In-

[31] Man könnte an dieser Stelle argumentieren, dass das der Grund war, warum moderne, objekt-orientierte Sprachen keine getrennten Header- und Quellcode-Dateien mehr haben.

stanzen nicht auf dem Heap sondern auf dem Stack liegen. Daraus folgt, dass sie an Methoden nicht per Referenz sondern als Kopie (Werteparameter) übergeben werden. Strukturell sind sich Strukturen und Klassen sehr ähnlich. Strukturen können – genau wie Klassen – Felder und Methoden mit den zugehörigen Sichtbarkeitsmodifizierern enthalten. Für die Definition einer Struktur wird statt dem Schlüsselwort `class` einfach das Schlüsselwort `struct` verwendet.

Es gibt jedoch auch einige Unterschiede und Einschränkungen. Auf die wichtigsten wird im Folgenden eingegangen:

1. Den ersten Unterschied stellt die Erzeugung einer Instanz einer Struktur dar. Da diese Instanzen nur auf dem Stack existieren, besitzt eine Variable, die vom Typ einer Struktur ist, immer einen Wert. D.h. sie ist niemals `null` oder undefiniert (genau wie primitive Datentypen). Eine Instanz einer Struktur wird also in dem Moment angelegt, in dem die zugehörige Variable deklariert wird (siehe **Code-Listing VII-11**).
2. Eine Struktur kann zwar Schnittstellen implementieren, jedoch nicht von anderen Strukturen abgeleitet sein[32]. Dementsprechend ist der Sichtbarkeitsmodifizierer `protected` im Kontext einer Struktur sinnlos.
3. Der parameterlose Konstruktor einer Struktur kann nicht überschrieben werden. Soll ein eigener Konstruktor für eine Struktur geschrieben werden, so muss dieser mindestens einen Parameter haben. Konstruktoren müssen darüberhinaus *alle* Membervariablen der Struktur initialisieren. Der Aufruf eines Konstruktors erfolgt – genau wie bei Klassen – mit `new`[33].

```
public struct Person
{
    string Name;
    int Alter;
}

...

Person einePerson;
```

Code-Listing VII-11 Erzeugen einer Struktur-Instanz.

D. Eigenschaften

Eigenschaften stellen eine Erweiterung zu Feldern (Variablen und Konstanten) einer Klasse dar. Stellen wir uns vor, wir hätten eine Klasse `Kreis`. Da jeder Kreis einen Radius hat, besitzt diese Klasse eine entsprechende Membervariable (siehe **Code-Listing VII-12**).

```
class Kreis
{
    public double Radius;
}
```

Code-Listing VII-12 Kreis-Klasse mit Radius-Membervariable

[32] Intern werden Strukturen jedoch von `ValueType` abgeleitet, das wiederum von `Object` abgeleitet ist.

[33] Dabei ist jedoch zu beachten, dass die Instanz zwei Mal initialisiert wird. Einmal beim Anlegen der Variable (durch den parameterlosen Konstruktor) und dann einmal durch den Aufruf des parametrisierten Konstruktor.

Der Zugriff auf diese Variable wäre einfach mit dem Zuweisungsoperator möglich, wie in **Code-Listing VII-13** angegeben.

```
Kreis derKreis = new Kreis();
derKreis.Radius = 5.0;
```

Code-Listing VII-13 Änderung des Radius

Nachdem die Klasse nun einige Zeit in Verwendung war, fällt uns auf, dass es ja ohne Probleme möglich ist, auch einen negativen Radius anzugeben – und das ist natürlich schlecht. Es muss also eine Möglichkeit gefunden werden, die Eingabe eines negativen Radius‘ irgendwie zu verhindern. Überlicherweise würde man die Klasse um Getter und Setter erweitern und die Variable „verstecken“. Wir müssten die Implementierung unserer Klasse also wie in **Code-Listing VII-14** dargestellt abändern.

```
class Kreis
{
    private double Radius;

    public void setRadius(double Radius)
    {
        if (Radius < 0.0)
            Console.WriteLine("Negativer Radius!");
        else
          this.Radius = Radius;
    }

    public dobule getRadius()
    {
        return this.Radius;
    }
}
```

Code-Listing VII-14 Werte für Radius durch Setter-Methode einschränken

Das Problem bei diesem Ansatz ist jedoch, dass wir an allen Stellen, an denen außerhalb der Klasse auf `Radius` zugegriffen wurde, den Zuweisungsoperator `=` durch die Getter- bzw. Setter-Methode ersetzen müssten. Und das stellt, gerade wenn die Klasse bereits lange in Gebrauch war – einen großen Aufwand dar.

In C# wird dieses Problem mit Hilfe der sog. *Eigenschaften* gelöst. Das Szenario bleibt das gleiche, allerdings verändern wir jetzt die Variable `Radius` und wandeln sie – durch Entfernen des Semikolons und Hinzufügen der sog. *Accessoren* `get` und `set` – in eine Eigenschaft um. Das Ergebnis dieser Aktion könnte so aussehen, wie in **Code-Listing VI-15** dargestellt.

```
class Kreis
{
    private double radius;
    public double Radius
    {
        set
        {
            if (value < 0.0)
              Console.WriteLine("Negativer Radius!");
            else
              this.radius = value;
        }

        get
        {
            return this.radius;
        }
    }
}
```

Code-Listing VII-15 Implementierung von "Radius" als Eigenschaft

Die eben angesprochenen Accessoren kann man sich als Methoden vorstellen, die dem Getter bzw. dem Setter entsprechen. Der Vorteil dieses Verfahrens ist, dass keine Änderungen am Code *außerhalb* der Klasse vorgenommen werden müssen – der Zugriff auf die Eigenschaft erfolgt weiterhin über den Zuweisungsoperator, wie in **Code-Listing VII-16** dargestellt.

```
Kreis derKreis = new Kreis();

// Ruft den Accessor "set" auf
derKreis.Radius = 5.0;

// Ruft den Accessor "get" auf
double rad = derKreis.Radius;
```

Code-Listing VII-16 Verwendung von Accessoren

So können zusätzliche Getter und Setter in eine Klasse eingebaut werden, ohne Code außerhalb der Klasse ändern zu müssen, was sehr viel Änderungsaufwand spart. Und gleichzeitig spart man bei der Verwendung von Eigenschaften (im Gegensatz zur Verwendung von Gettern und Settern) Schreibarbeit, da ein Zuweisungsoperator schneller geschrieben ist als ein Methodenname.

Noch ein paar *Hinweise* zur Verwendung von Eigenschaften:

- Im Accessor `set` steht automatisch eine Konstante namens `value`[34] zur Verfügung. Diese enthält den Wert, der an den Accessor (mittels Zuweisungsoperator) übergeben wurde.
- Eine Eigenschaft selbst kann *keine* Werte speichern. Aus diesem Grund musste im obigen Code eine neue Variable namens `radius`[35] eingeführt werden, die jetzt den eigentlichen Speicherort des Radius darstellt.
- Die Sichtbarkeit *eines* der beiden Accessoren kann verändert werden, um z.B. nur Kindklassen Schreibzugriff zu gewähren.
- Durch Weglassen eines der beiden Accessoren kann die Eigenschaft lese- oder schreibgeschützt werden.

[34] Es handelt sich dabei um ein Schlüsselwort.

[35] An diesem Variablennamen erkennt man die übliche Schreibweise von Bezeichnern in C#: private Bezeichner beginnen mit einem Kleinbuchstaben, alle anderen mit einem Großbuchstaben.

E. Funktionszeiger (Delegate)

Das Konzept der Funktionszeiger[36] ist nicht neu. Bereits in C und C++ gab es dieses Konzept. In Java findet man sie nicht, weil sie vielleicht im Zuge der Eliminierung der Pointerarithmetik mit entfernt oder aus Simplizitätsgründen einfach weggelassen wurden. Was auch immer der wirkliche Grund war – in C# wurden sie wieder[37] eingeführt, und zwar unter dem Namen *Delegates*.

1) Allgemein

Was ist nun ein Delegate? Kurz gesagt: Eine Variable oder Konstante, in der eine Methode „gespeichert“ werden kann.

Die Funktionsweise erklärt sich am besten an einem Beispiel: Nehmen wir an, wir wollten einen Taschenrechner programmieren, der die Grundrechenarten unterstützt. Die Klasse hätte also mindestens vier Methoden – eine für jede Grundrechenart. (siehe **Code-Listing VII-17**)

```
class Taschenrechner
{
    static double Addition(double x, double y)
    {
        return x + y;
    }

    static double Subtraktion(double x, double y)
    {
        return x - y;
    }

    static double Multiplikation(double x, double y)
    {
        return x * y;
    }

    static double Division(double x, double y)
    {
        return x / y;
    }
}
```

Code-Listing VII-17 Methoden der Grundrechenarten

Als nächstes möchten wir eine Methode definieren, die die der Grundrechenart entsprechenden Methode zurück gibt. Um so etwas zu realisieren, müsste man die Methode in einem Datentyp speichern können – und genau das geht mit Delegaten.

Wir erweitern also unsere Klasse um einen entsprechenden Delegatentyp. Dazu verwendet man das Schlüsselwort `delegate`, wie in **Code-Listing VII-18** dargestellt.

```
class Taschenrechner
{
    public delegate double Operation(double x,
                                     double y);

    ...
}
```

Code-Listing VII-18 Delegaten für Grundrechenarten definieren

Wir definieren mit diesem Befehl einen neuen Datentyp (also keine Variable/Konstante) namens `Operation`, der ein Delegate ist. Da C# typensicher ist, müssen bei einem Delegaten der Rückgabewert und die Parameterliste festge-

[36] Die korrekte Bezeichnung müsste Methodenzeiger lauten. Dieser Name ist aber nicht üblich.

[37] Bei der Entwicklung von C# wurde versucht, Designschwächen aus u.a. Java auszumerzen – auch wenn Microsoft vehement dementiert, dass es Ähnlichkeiten zwischen C# und Java gibt.

legt sein. D.h. es können nur Methoden in einer Variable vom Typ `Operation` gespeichert werden, die – in unserem Fall – zwei `double`-Parameter besitzen und einen `double`-Wert zurückgeben.

Als letztes fügen wir unserer Klasse noch eine Methode hinzu, die eine Methoden – abhängig von der gewählten Rechenart – zurückgibt. Das könnte so aussehen, wie in **Code-Listing VII-19** dargestellt.

```
class Taschenrechner
{
    ...

    public static Operation Mache(char Typ)
    {
        switch (Typ)
        {
            case '+': return Addition;
            case '-': return Subtraktion;
            case '*': return Multiplikation;
            case '/': return Division;
        }

        return null;
    }
}
```

Code-Listing VII-19 Methode für Grundrechenarten zurückgeben

Wie man erkennen kann, werden als Rückgabewerte tatsächlich einfach die Namen der Methoden verwendet – es werden keine Wrapperklassen oder ähnliches benötigt.

Auf Seite des Aufrufers spielt sich die Verwendung der Delegaten wie in **Code-Listing VII-20** dargestellt ab.

```
Taschenrechner.Operation op;
// Erhalten des Funktionszeigers
op = Taschenrechner.Mache('*');
// Aufrufen wie eine Methode
double ergebnis = op(5.0, 3.0);

Console.WriteLine("Ergebnis: {0}", ergebnis);
// Ausgabe: "Ergebnis: 15"
```

Code-Listing VII-20 Verwendung eines Delegaten

Die Methode in einem Delegaten wird also wie eine „normale“ Methode aufgerufen: `Variablenname(Parameter);`

2) *Anonyme Delegates*

Mit dem .NET-Framework 2.0 wurde eine Erweiterung der Delegaten eingeführt: die *anonymen* Delegaten. Bei einem anonymen Delegate existiert die zugewiesene Methode nicht als Methode einer Klasse, sondern wird direkt bei der Zuweisung „erzeugt“.

Um einen solchen anonymen Delegaten zu verwenden, notiert man das Schlüsselwort `delegate` gefolgt von der Parameterliste und der Implementierung der Methode. Abgeschlossen wird der Befehl von einem Semikolon. Die Klasse `Taschenrechner` könnte man damit wie in **Code-Listing VII-21** umschreiben, ohne dass sich an der Funktionalität etwas ändert.

Zu beachten ist, dass es sich bei dem Code aus **Code-Listing VII-21** um die gesamte Klasse (und nicht nur einen Ausschnitt) handelt. D.h. sie besteht nur noch aus einer Methode. Die Methoden für die einzelnen Grundrechenarten wurden als anonyme Delegaten *in* die Methode `Mache()` verlagert.

```
class Taschenrechner
{
    public delegate double Operation(double x,
                                     double y);

    public static Operation Mache(char Typ)
    {
        Operation Addition = delegate(double x,
                                      double y)
        {
            return x + y;
        };

        Operation Subtraktion = delegate(double x,
                                         double y)
        {
            return x - y;
        };

        Operation Multiplikation = delegate(double x,
                                            double y)
        {
            return x * y;
        };

        Operation Division = delegate(double x,
                                      double y)
        {
            return x / y;
        };

        switch (Typ) {
            case '+': return Addition;
            case '-': return Subtraktion;
            case '*': return Multiplikation;
            case '/': return Division;
        }
        return null;
    }
}
```

Code-Listing VII-21 Taschenrechner mit anonymen Delegaten

3) *Delegaten und Schnittstellen*

Beim Betrachten des Konzepts der Delegaten stellt sich die Frage: Wofür brauche ich sie, wenn ich doch Schnittstellen[38] habe?

Sicherlich dienen Delegaten der Vereinfachung von Code. Wer in Java mit Schnittstellen[39] programmiert hat, wird sich so manches Mal gewünscht haben, dass die Methode einer Schnittstelle nicht `public` sein müssen. Es gibt zwar die Möglichkeit, für eine Schnittstelle eine innere oder anonyme Klasse zu schreiben, aber das reißt den Code der Klasse doch irgendwie auseinander. Außerdem sind solche Klassen immer nur ein Hilfskonstrukt[41], da sie eigentlich keine Datenelemente enthalten und somit ihre Methoden eigentlich in die umgebende Klasse integriert werden sollten.

Genau an dieser Stelle sind Delegaten sehr praktisch. Nehmen wir als Beispiel die Java-Schnittstelle `ActionListener`. Diese Schnittstelle hat nur eine Methode namens `actionPerformed()`. Um diese jetzt zu implementieren hätten wir unter Java nur die beiden oben genannten Möglichkeiten. Unter C# hingegen könnte man auf einen Delegaten ausweichen[40], sodass keine semantisch[41] unnötigen Klassen benötigt würden und die Sichtbarkeit der Listener-Methode frei wählbar wäre. Aus diesem Grund

[38] Unter C sind Funktionszeiger notwendig, da es dort keine Interfaces (oder sonstige objekt-orientierte Strukturen) gibt.

[39] Gemeint sind Konstrukte mit dem Schlüsselwort `interface`

[40] Das muss natürlich von der Methode, die den Listener verwendet, unterstützt werden.

[41] „semantisch unnötig“ meint hier, dass die Klasse keinen „tieferen“ Sinn hat – sie dient nur als Hilfskonstrukt. Die Methoden der Klasse würden bei einem programmiersprachen-unabhängigen Entwurf direkt in die umgebende Klasse integriert.

werden in C# übrigens Ereignisse (engl. *events*) auch mit Delegaten und nicht mit Schnittstellen realisiert.

Allerdings sind Delegaten auch kein Allheilmittel und können Schnittstellen nicht überall ersetzen. Da beide Konzepte aber sehr nahe beieinander liegen, hat Microsoft im MSDN einen kleinen Leitfaden[VIII] herausgegeben, wann man ihrer Meinung nach Delegaten und wann Schnittstellen benutzen soll. Der Vollständigkeit halber sei die Liste aus dem Leitfaden hier zitiert:

> Verwenden Sie einen Delegaten, wenn:
> - ein Ereignisentwurfsmuster verwendet wird.
> - eine statische Methode gekapselt werden soll.
> - der Aufrufer keinen Zugriff auf weitere Eigenschaften, Methoden oder Schnittstellen des Objekts benötigt, das die Methode implementiert.
> - die einfache Verknüpfung von Delegaten gewünscht ist.
> - eine Klasse möglicherweise mehr als eine Implementierung der Methode benötigt.
>
> Verwenden Sie eine Schnittstelle, wenn:
> - es eine Gruppe verwandter Methoden gibt, die möglicherweise aufgerufen werden.
> - eine Klasse nur eine Implementierung der Methode benötigt.
> - die Klasse, die die Schnittstelle verwendet, eine Umwandlung dieser Schnittstelle in andere Schnittstellen oder Klassentypen durchführt.
> - die Methode, die implementiert wird, mit dem Typ oder der Identität der Klasse verknüpft ist, wie zum Beispiel bei Vergleichsmethoden.

F. *Ereignisse (Events)*

Ereignisse sind bei *Swing* in *Java* Gang und Gebe. Diese werden dort über bestimmte (verschiedene[42]) Klassen und Schnittstellen definiert. In C# ist dieses Konzept bereits in der Sprache selbst verankert. Es gibt dafür extra ein Schlüsselwort, nämlich `event`, das mit *einem* Datentyp auskommt.

Zur Erinnerung – Ereignisse müssen folgende Funktionalitäten aufweisen:

1. Hinzufügen und Entfernen von Listenern[43]
2. Auslösen des Ereignisses

Beides ist in C# sehr elegant gelöst, doch dazu etwas später. Zunächst benötigt man die Signatur der Listener-Methoden, die auf das Ereignis reagieren sollen. In *Java* wird das über Schnittstellen realisiert. Schnittstellen für Listener zu verwenden, hat jedoch einige Nachteile:

1. Methoden aus Schnittstellen müssen immer `public` sein. Das ist häufig ungewollt und kann nur durch innere oder anonyme Klassen umgangen werden.
2. Es müssen immer alle Methoden einer Schnittstelle implementiert werden – auch wenn man nur eine der Methoden benötigt.

[42] Meistens eine Klasse für die Verwaltung des Listener-Liste und eine Schnittstelle zur Implementierung der Listener selbst.

[43] Auch „Event-Handler" genannt. Methoden, die über das Auslösen eines Ereignisses informiert werden.

In C# wurden glücklicherweise ein anderer Weg gewählt. Hier werden Listener über Delegaten (siehe VII.E Funktionszeiger (Delegate)) realisiert. Dadurch werden alle eben genannten Nachteile aufgehoben.

Um jetzt ein Ereignis zu verwenden, müssen zunächst ein Delegat und ein Ereignis definiert werden. Während es sich bei dem Delegaten um einen „Datentyp" handelt, ist das Ereignis eine (bereits vollständig instanziierte) Variable. **Code-Listing VII-22** zeigt den hierfür notwendigen Code. Mit den zwei abgebildeten Zeilen ist das Ereignis bereits vollständig einsatzfähig.

```
// Delegat: Signatur für Listener
public delegate void MeasureErrorEventHandler();

// Ereignis: verwendet den Delegaten
public event MeasureErrorEventHandler MeasureError;
```

Code-Listing VII-22 Anlegen eines Ereignisses

Kommen wir nun zu den beiden oben geforderten Funktionalitäten eines Ereignisses zurück. Diese werden in C# wie folgt umgesetzt (siehe **Code-Listing VII-23**):

1. Die *Hinzufügen und Entfernen von Listenern* funktioniert in C# ganz einfach über die Operatoren `+=` und `-=`. Auf der linken Seite steht dabei jeweils das Ereignis und auf der rechten Seite der Listener (in Form eines Methodennamens).
2. Der Aufruf des Ereignisses (also das Benachrichtigen aller Listener) geschieht genau wie der Aufruf eines Delegaten. Als Name für den Delegaten wird hierbei der Name des Ereignisses verwendet. *Hinweis:* Dabei sollte aber darauf geachtet werden, dass das Ereignis `null` ist, falls dem Ereignis keine Listener hinzugefügt wurden. Wird das nicht überprüft, kommt es zu einer `NullReferenceException`.

```
// Definieren eines Listeners innerhalb einer Klasse
private void EinListener()
{
  ...
}

// Hinzufügen des Listeners
MeasureError += EinListener;

// Auslösen des Ereignisses
if (MeasureError != null)
  MeasureError();
```

Code-Listing VII-23 Verwendung eines Ereignisses

ANHANG I
ZUM WEITERLESEN

Zum Abschluss noch ein paar Worte zu diesem Artikel: Die meisten Informationen in diesem Artikel stammen aus dem Buch „Visual C# 2005" aus dem Galileo-Computing-Verlag[44]. Das schöne an diesem Buch ist, dass es

- sehr einfach und in Deutsch geschrieben ist
- alle Grundlagen und auch einige fortgeschrittene Themen abdeckt

[44] http://www.galileocomputing.de/

- kostenlos zum Download[45] verfügbar ist.

Trotzdem ist dieser Artikel keine Kopie zu diesem Buch, da er wesentlich kürzer ist, was daran liegt, dass er für (erfahrene) Java-Programmierer und das Buch für Anfänger geschrieben ist. Im Buch finden sich also viele Themen, die in diesem Artikel getrost übersprungen werden können. Desweiteren wurde – auf Grund der Ausrichtung speziell auf Java-Programmierer – in diesem Artikel viele Parallelen zwischen Java und C# gezogen (gerade in den Abschnitten VI und VI.E), die sich in dem Buch so nicht finden.

Dennoch ist das Buch sowohl als Tutorial als auch als Referenz für jeden C#-Programmier sehr empfehlenswert – gerade auch weil die 16 Seiten dieses Artikels nicht ansatzweise den Inhalt der über 1300 Seiten des Buchs fassen können[46].

Da in diesem Beitrag viele Themen nur angerissen werden konnten, seien die folgenden Kapitel aus dem Buch zum Weiterlesen empfohlen.

Gemeinsamkeiten mit Java:
- Mehrdimensionale Arrays: Kapitel *3.5.4* und *3.5.8*
- Schleifen mit `foreach`: Kapitel *3.7.2*
- Typen-Operatoren: Kapitel *4.7.5* (`is`), Kapitel *6.10.4* (`as`), Kapitel *14.2.1* (`typeof` und `GetType()`)
- Neue, primitive Datentypen: Kapitel *3.3.5*
- Aufzählungstypen (`enum`): Kapitel *5.5*
- Generics: Kapitel *7.4*
- Multithreading: Kapitel *11*

Erweiterungen gegenüber Java:
- Parameter mit `ref` und `out`: Kapitel *4.3.7*
- Operator-Überladung: Kapitel *7.1*
- Indexer: Kapitel *7.2*
- Attribute: Kapitel *7.6*

Darüberhinaus wurden – wie bereits am Anfang dieses Artikels erwähnt – die Kapitel 8 – 27 in diesem Artikel nicht behandelt. Diese beschäftigen sich allerdings eher mit allgemeinen Themen rund um .NET. Die Sprachgrundlagen für C# wurden aber in den Kapiteln 1 bis 7 größtenteils abgehandelt.

LITERATUR

[1] Andreas Kühnel: Visual C# 2005 : Galileo Computing Verlag (2005), ISBN 3-89842-586-X
[2] Microsoft: MSDN

[I] Die CLI ist standardisiert und kann unter folgender URL eingesehen werden: http://www.ecma-international.org/publications/standards/Ecma-335.htm
[II] http://commons.wikimedia.org/wiki/Image:Overview_of_the_Common_Language_Infrastructure.png
[III] http://msdn2.microsoft.com/de-de/netframework/aa497273(en-us).aspx
[IV] http://www.microsoft.com/downloads/details.aspx?FamilyID=a5a02311-194b-4c00-b445-f92bec03032f
[V] http://www.microsoft.com/germany/msdn/vstudio/products/
[VI] http://msdn2.microsoft.com/en-us/vstudio/products/default.aspx
[VII] http://de.wikipedia.org/wiki/Vererbung_(Programmierung)
[VIII] Quelle: http://msdn2.microsoft.com/de-de/library/ms173173.aspx

[45] http://www.galileocomputing.de/katalog/openbook/
[46] Dieser Artikel deckt gerade mal die ersten sieben Kapitel der insgesamt 27 Kapitel des Buchs ab – und das auch nicht vollständig.