Eric Gunnerson

C#

Tutorial und Referenz

Galileo Computing

Inhalt

1 Grundlagen der objektorientierten Programmierung 19

1.1 Was ist ein Objekt? 19

1.2 Vererbung 19

1.3 Das Prinzip des Containments 21

1.4 Polymorphismus und virtuelle Funktionen 21

1.5 Kapselung und Sichtbarkeit 23

2 Die .NET-Laufzeitumgebung 25

2.1 Die Ausführungsumgebung 26
2.1.1 Ein einfacheres Programmiermodell 26
2.1.2 Sicherheit 28
2.1.3 Unterstützung für leistungsfähige Tools 29
2.1.4 Bereitstellen, Packen und weitere Funktionen 29

2.2 Metadaten 30

2.3 Assemblierung 31

2.4 Sprachinteroperabilität 31

2.5 Attribute 32

3 Schnelleinstieg und Entwicklung in C# 33

3.1 Hello, Universe 33

3.2 Namespaces und Using-Klausel 34

3.3 Namespace und Assembly 35

3.4 Grundlegende Datentypen 36

3.5 Klassen, Strukturen und Schnittstellen 38

3.6 Anweisungen 38

3.7 Enum 38

3.8 Delegates und Ereignisse 39

3.9 Eigenschaften und Indizierer 39

3.10 Attribute 40

3.11 Entwickeln in C# 40
3.11.1 Der Befehlszeilencompiler 41
3.11.2 Visual Studio.NET 41
3.11.3 Weitere nennenswerte Tools 43

4 Ausnahmebehandlung 45

4.1 Was ist falsch an Rückgabecodes? 45
4.2 try und catch 46
4.3 Die Ausnahmehierarchie 47
4.4 Übergeben von Ausnahmen an die aufrufende Funktion 49
4.4.1 Caller Beware 50
4.4.2 Caller Confuse 50
4.4.3 Caller Inform 51
4.5 Benutzerdefinierte Ausnahmeklassen 52
4.6 Finally 54
4.7 Effizienz und Overhead 56
4.8 Entwurfsrichtlinien 57

5 101-Klassen 59

5.1 Eine einfache Klasse 59
5.2 Mitgliedsfunktionen 61
5.3 Ref- und Out-Parameter 62
5.4 Überladung 65

6 Basisklassen und Vererbung 67

6.1 Die Engineer-Klasse 67
6.2 Einfache Vererbung 68
6.3 Engineer-Arrays 70
6.4 Virtuelle Funktionen 75
6.5 Abstrakte Klassen 77
6.6 sealed-Schlüsselwort für Klassen und Methoden 81

7 Mitgliedszugriff und Überladung 83

7.1 Klassenzugriff 83
7.2 Verwenden von internal für Klassenmitglieder 83
7.3 internal protected 85
7.4 Die Beziehung zwischen Klassen- und Mitgliedszugriff 85
7.5 Methodenüberladung 86
7.5.1 Verborgene Methoden 86
7.5.2 Die »bessere« Konvertierung 88
7.6 Parameterlisten variabler Länge 89

8 Mehr zu Klassen 93

8.1 Geschachtelte Klassen 93

8.2 Weitere Schachtelungen 94

8.3 Erstellung, Initialisierung, Zerstörung 94
8.3.1 Konstruktoren 94
8.3.2 Initialisierung 97
8.3.3 Destruktoren 98
8.3.4 Verwaltung von Nicht-Speicherressourcen 99
8.3.5 IDisposable und die using-Anweisung 101
8.3.6 IDisposable und langlebige Objekte 102

8.4 Statische Felder 103

8.5 Statische Mitgliedsfunktionen 104

8.6 Statische Konstruktoren 105

8.7 Konstanten 106

8.8 Schreibgeschützte Felder 107

9 Strukturen (Wertetypen) 111

9.1 Eine Punktstruktur 111

9.2 Boxing und Unboxing 112

9.3 Strukturen und Konstruktoren 114

9.4 Entwurfsrichtlinien 115

9.5 Unveränderliche Klassen 115

10 Schnittstellen 117

10.1 Ein einfaches Beispiel 117

10.2 Arbeiten mit Schnittstellen 119

10.3 Der as-Operator 121

10.4 Schnittstellen und Vererbung 122

10.5 Entwurfsrichtlinien 123

10.6 Mehrfachimplementierung 124
10.6.1 Explizite Schnittstellenimplementierung 125
10.6.2 Ausblenden der Implementierung 129

10.7 Auf Schnittstellen basierende Schnittstellen 130

10.8 Schnittstellen und Strukturen 131

11 Versionssteuerung 133

11.1 Ein Beispiel zur Versionssteuerung 133

11.2 Abgestimmt auf Versionssteuerung 136

11.3 Entwickeln unter Berücksichtigung der Versionssteuerung 136

12 Anweisungen und Ausführungsverlauf 137

12.1 Auswahlanweisungen 137
12.1.1 If 137
12.1.2 Switch 137

12.2 Wiederholungsanweisungen 139
12.2.1 While 140
12.2.2 Do 141
12.2.3 For 141
12.2.4 Foreach 142

12.3 Sprunganweisungen 145
12.3.1 Break 145
12.3.2 Continue 145
12.3.3 Goto 145
12.3.4 Return 145

12.4 Weitere Anweisungen 146
12.4.1 lock 146
12.4.2 using 146
12.4.3 try/catch/finally 146
12.4.4 checked/unchecked 146

13 Variablenbereich und feste Zuordnung 147

13.1 Feste Zuordnung 148
13.1.1 Feste Zuordnungen und Arrays 150

14 Operatoren und Ausdrücke 153

14.1 Rangfolge der Operatoren 153

14.2 Integrierte Operatoren 154

14.3 Benutzerdefinierte Operatoren 155

14.4 Numerische Umwandlungen 155

14.5 Arithmetische Operatoren 155
14.5.1 Unär Plus (+) über 155
14.5.2 Unär Minus (-) über 155
14.5.3 Bitweises Komplement (~) über 156
14.5.4 Addition (+) über 156
14.5.5 Numerische Addition 156

14.5.6 Subtraktion (-) über 157
14.5.7 Multiplikation (*) über 157
14.5.8 Division (/) über 157
14.5.9 Restbetrag (%) über 157
14.5.10 Verschiebung (<< und >>) über 157
14.5.11 Wertezuwachs und -abname (++ und --) über 157

14.6 Relationale und logische Operatoren 158
14.6.1 Logische Negation (!) über 158
14.6.2 Relationale Operatoren über 158
14.6.3 Logische Operatoren über 159
14.6.4 Bedingungsoperator (?:) 160

14.7 Zuweisungsoperatoren 160
14.7.1 Einfache Zuweisung 160
14.7.2 Komplexe Zuweisung 160

14.8 Typenoperatoren 161
14.8.1 typeof 161
14.8.2 is 161
14.8.3 as 163

14.9 checked und unchecked 163

15 Konvertierungen 165

15.1 Numerische Typen 165
15.1.1 Konvertierungen und Mitgliedsermittlung 166
15.1.2 Explizite numerische Konvertierungen 168
15.1.3 Geprüfte Konvertierungen 169

15.2 Konvertierung von Klassen (Verweistypen) 170
15.2.1 In die Basisklasse eines Objekts 171
15.2.2 In eine Schnittstelle, die das Objekt implementiert 173
15.2.3 In eine Schnittstelle, die das Objekt möglicherweise implementiert 173
15.2.4 Von einem Schnittstellentyp in einen anderen 175

15.3 Konvertierung von Strukturen (Wertetypen) 175

16 Arrays 177

16.1 Arrayinitialisierung 177

16.2 Mehrdimensionale und unregelmäßige Arrays 177
16.2.1 Mehrdimensionale Arrays 178
16.2.2 Unregelmäßige Arrays 179

16.3 Arrays vom Verweistyp 181

16.4 Arraykonvertierungen 182

16.5 System.Array 183
16.5.1 Sortieren und Suchen 183
16.5.2 Reverse 183

17 Zeichenfolgen 185

17.1 Operationen 185

17.2 Codierung und Konvertierung von Zeichenfolgen 187

17.3 Konvertieren von Objekten in Zeichenfolgen 187

17.4 Ein Beispiel 187

17.5 Interning von Zeichenfolgen 188
17.5.1 StringBuilder 189

17.6 Reguläre Ausdrücke 191
17.6.1 Optionen für reguläre Ausdrücke 192
17.6.2 Komplexere Syntaxanalyse 193

18 Eigenschaften 197

18.1 Zugriffsroutinen 198
18.1.1 Eigenschaften und Vererbung 198
18.1.2 Verwendung von Eigenschaften 199
18.1.3 Nebeneffekte beim Setzen von Werten 200
18.1.4 Statische Eigenschaften 202
18.1.5 Eigenschafteneffizienz 203

18.2 Zugriff auf Eigenschaften 204
18.2.1 Virtuelle Eigenschaften 205

19 Indizierer und Aufzählungsbezeichner 207

19.1 Indizierung mit einem integer-Index 207

19.2 Indizierung mit einem Zeichenfolgenindex 209

19.3 Indizierung mit mehreren Parametern 212

19.4 Aufzählungsbezeichner und foreach 214
19.4.1 Verbesserungen am Aufzählungsbezeichner 218

19.5 Einmal verwendbare Aufzählungsbezeichner 220
19.5.1 GetEnumerator() gibt IEnumerator zurück 221
19.5.2 GetEnumerator() gibt eine Klasse zurück, die IDisposable implementiert 221
19.5.3 GetEnumerator() gibt eine Klasse zurück, die IDisposable nicht implementiert 222

19.6 Entwurfsrichtlinien 222

20 Aufzählungen 223

20.1 Eine Beispielaufzählung 223

20.2 Basistypen für die Aufzählung 224

20.3 Initialisierung 225

20.4 Bitflagaufzählungen 226
20.5 Konvertierungen 227
20.6 System.Enum 228

21 Attribute 231

21.1 Verwenden von Attributen 232
21.1.1 Noch ein paar Details 234
21.2 Eigene Attribute 237
21.2.1 Attributverwendung 237
21.2.2 Attributparameter 238
21.3 Attributreflexion 239

22 Delegates 243

22.1 Verwenden von Delegates 243
22.2 Delegates für Instanzenmitglieder 245
22.3 Multicasting 246
22.4 Delegates als statische Mitglieder 249
22.5 Delegates als statische Eigenschaften 251

23 Ereignisse 255

23.1 Funktionen hinzufügen und entfernen 257
23.2 Benutzerdefiniertes Hinzufügen und Entfernen 260

24 Benutzerdefinierte Konvertierung 269

24.1 Ein einfaches Beispiel 269
24.2 Prä- und Postkonvertierungen 272
24.3 Konvertierungen zwischen Strukturen 273
24.4 Klassen und Prä-/Postkonvertierungen 279
24.5 Entwurfsrichtlinien 286
24.5.1 Implizite Konvertierungen sind sichere Konvertierungen 286
24.5.2 Die Konvertierung im komplexeren Typ definieren 286
24.5.3 Eine Konvertierung aus oder in eine(r) Hierarchie 286
24.5.4 Konvertierungen nur bei Bedarf verwenden 287
24.5.5 Konvertierungen, die in anderen Sprachen funktionieren 287
24.6 So funktioniert's 289
24.6.1 Konvertierungssuche 289

25 Operatorüberladung 293

25.1 Unäre Operatoren 293

25.2 Binäre Operatoren 294

25.3 Ein Beispiel 294

25.4 Beschränkungen 295

25.5 Richtlinien 296

25.6 Eine Klasse komplexer Zahlen 297

26 Weitere Sprachdetails 301

26.1 Die Main()-Funktion 301
26.1.1 Zurückgeben eines int-Status 301
26.1.2 Befehlszeilenparameter 301
26.1.3 Mehrere Main()-Funktionen 302

26.2 Vorverarbeitung 303
26.2.1 Vorverarbeitungsdirektiven 304
26.2.2 Andere Präprozessorfunktionen 305

26.3 Lexikalische Details 306
26.3.1 Bezeichner 306
26.3.2 Literale 308
26.3.3 Kommentare 310

27 Freundschaft schließen mit den .NET Frameworks 313

27.1 Was alle Objekte tun 313
27.1.1 ToString() 313
27.1.2 Equals() 315

27.2 Hashalgorithmen und GetHashCode() 317

27.3 Entwurfsrichtlinien 320
27.3.1 Richtlinien für Wertetypen 320
27.3.2 Richtlinien für Verweistypen 320

28 System.Array und die Auflistungsklassen 325

28.1 Sortieren und Suchen 325
28.1.1 Implementieren von IComparable 325
28.1.2 Verwenden von IComparer 327
28.1.3 IComparer als Eigenschaft 331
28.1.4 Überladen relationaler Operatoren 334
28.1.5 Erweiterte Verwendung von Hashes 336

28.2 Synchronisierte Auflistungen 339

28.3 Auflistungen ohne Berücksichtigung der Groß- und Kleinschreibung 339
28.4 ICloneable 339
28.5 Weitere Auflistungen 342
28.6 Entwurfsrichtlinien 342
28.6.1 Funktionen und Schnittstellen der Frameworkklassen 343

29 Threading und asynchrone Operationen 345

29.1 Datensicherung und Synchronisierung 345
29.1.1 Ein nicht ganz lauffähiges Beispiel 345
29.1.2 Schutzmethoden 350
29.2 Zugriffsreihenfolge und volatile-Schlüsselwort 353
29.2.1 Verwendung von volatile 356
29.3 Threads 358
29.3.1 Join()-Funktion 358
29.3.2 WaitHandle 360
29.3.3 Threadlokale Speicherung 361
29.4 Asynchrone Aufrufe 362
29.4.1 Ein einfaches Beispiel 363
29.4.2 Rückgabewerte 365
29.4.3 Auf Beendigung warten 367
29.5 Klassen, die asynchrone Aufrufe direkt unterstützen 371
29.6 Entwurfsrichtlinien 372
29.7 Threadpools 372

30 Codeerzeugung zur Ausführungszeit 373

30.1 Laden von Assemblies 373
30.1.1 So wird's dynamisch 375
30.2 Benutzerdefinierte Codeerzeugung 377
30.2.1 Polynomauswertung 377
30.2.2 Eine benutzerdefinierte C#-Klasse 383
30.2.3 Eine schnelle benutzerdefinierte C#-Klasse 387
30.2.4 Eine CodeDOM-Implementierung 388
30.2.5 Eine Reflection.Emit-Implementierung 392
30.3 Zusammenfassung 397

31 Interoperabilität 399

31.1 Benutzen von COM-Objekten 399
31.2 Von COM-Objekten benutzt werden 399

31.3 Aufrufen systemeigener DLL-Funktionen 399
31.3.1 Zeiger und deklarative Verankerung 400
31.3.2 Strukturlayout 403
31.3.3 Aufruf einer Funktion mit Strukturparameter 404
31.3.4 Kopplung (Hookup) an einen Windows-Rückruf 406
31.3.5 Entwurfsrichtlinien 408

32 Überblick über die .NET Frameworks 411

32.1 Numerische Formatierung 411
32.1.1 Standardformatzeichenfolgen 411
32.1.2 Benutzerdefinierte Formatzeichenfolgen 416
32.2 Formatierung von Datum und Uhrzeit 421
32.3 Benutzerdefinierte Objektformatierung 423
32.4 Numerische Syntaxanalyse 424
32.5 XML-Verwendung in C# 425
32.6 Eingabe/Ausgabe 425
32.6.1 Binärklassen 425
32.6.2 Text 426
32.6.3 XML 426
32.6.4 Lese- und Schreibvorgänge für Dateien 426
32.6.5 Durchsuchen von Verzeichnissen 427
32.6.6 Starten von Prozessen 429
32.7 Serialisierung 431
32.8 Benutzerdefinierte Serialisierung 434
32.9 Lesen von Webseiten 436
32.10 Zugriff auf Umgebungseinstellungen 438

33 Windows Forms 443

33.1 Die Anwendung 443
33.2 Erste Schritte 443
33.3 Der Formular-Designer 446
33.4 Ermitteln der Verzeichnisgröße 447
33.4.1 Größenberechnung 449
33.5 Anzeige von Verzeichnisstruktur und -größen 451
33.6 Festlegen des Verzeichnisses 454
33.7 Fortschrittsüberwachung 456

34 DiskDiff: Weitere Feinheiten 461

34.1 Auffüllen per Thread 461

34.2 Unterbrechen eines Threads 463
34.2.1 Eine Abbrechen-Schaltfläche 464

34.3 Verschönern der Strukturansicht 464
34.3.1 Expand-o-Matic 467
34.3.2 Auffüllen nach Bedarf 469

34.4 Sortieren der Dateien 471

34.5 Speichern und erneut abrufen 472
34.5.1 Steuerung der Serialisierung 474
34.5.2 Serialisierungsperformance 476
34.5.3 Feinere Steuerung der Serialisierung 477

35 DiskDiff: Noch mehr Sinnvolles 481

35.1 Vergleichen von Verzeichnissen 481

35.2 Dateimanipulation 482

35.3 Ein Bug im Kontextmenü 483

35.4 Datei- und Verzeichnisoperationen 486
35.4.1 Löschen 487
35.4.2 Löschen von Inhalten 487
35.4.3 Anzeige im Editor und Start 487

35.5 Aktualisieren der Benutzerschnittstelle 487
35.5.1 Eine kleine Umstrukturierung 489

35.6 Aufräumarbeiten 489
35.6.1 Tastaturkombinationen für den Schnellzugriff 490
35.6.2 Liste der zuletzt verwendeten Dateien 490
35.6.3 QuickInfo 492

35.7 Höhere Genauigkeit 492
35.7.1 Umstellung auf Verwendung der Clustergröße 495

36 C# im Detail 497

36.1 C#-Stil 497
36.1.1 Benennung 497
36.1.2 Kapselung 498

36.2 Richtlinien für Bibliotheksautoren 498
36.2.1 CLS-Kompatibilität 498
36.2.2 Benennung von Klassen 499

36.3 Unsicherer Kontext 499

36.4 XML-Dokumentation 505
36.4.1 Compilerunterstützungstags 505

36.4.2 XML-Dokumentationstags 509
36.4.3 XML-include-Dateien 510

36.5 Speicherbereinigung in der .NET-Laufzeitumgebung 511
36.5.1 Zuordnung 511
36.5.2 Kennzeichnen und Komprimieren 512
36.5.3 Generationen 512
36.5.4 Finalisierung 514
36.5.5 Steuerung des Verhaltens der Speicherbereinigung 515

36.6 Weitergehende Reflexion 516
36.6.1 Auflisten aller Typen in einer Assembly 516
36.6.2 Ermitteln von Mitgliedern 517
36.6.3 Aufrufen von Funktionen 519

36.7 Optimierung 523

37 Defensive Programmierung 525

37.1 Bedingte Methoden 525

37.2 Debug- und Trace-Klassen 526

37.3 Assert-Anweisungen 527

37.4 Debug- und Trace-Ausgabe 528

37.5 Verwenden von Switch-Klassen zur Steuerung von Debug und Trace 529
37.5.1 BooleanSwitch 530
37.5.2 TraceSwitch 532
37.5.3 Benutzerdefinierte Switch-Klassen 533

38 Der Befehlszeilencompiler 537

38.1 Einfache Verwendung 537

38.2 Antwortdateien 537

38.3 Standardantwortdatei 537

38.4 Befehlszeilenoptionen 538

39 C# im Vergleich zu anderen Sprachen 541

39.1 Unterschiede zwischen C# und C/C++ 541
39.1.1 Eine verwaltete Umgebung 541
39.1.2 .NET-Objekte 542
39.1.3 C#-Anweisungen 542
39.1.4 Attribute 543
39.1.5 Versionssteuerung 543
39.1.6 Codeorganisation 543
39.1.7 Fehlende C++-Funktionen 544

39.2 Unterschiede zwischen C# und Java 544
39.2.1 Datentypen 544
39.2.2 Erweitern des Typensystems 546
39.2.3 Klassen 547
39.2.4 Schnittstellen 550
39.2.5 Eigenschaften und Indizierer 550
39.2.6 Delegates und Ereignisse 551
39.2.7 Attribute 551
39.2.8 Anweisungen 551
39.3 Unterschiede zwischen C# und Visual Basic 6 553
39.3.1 Codeaussehen 554
39.3.2 Datentypen und Variablen 554
39.3.3 Operatoren und Ausdrücke 556
39.3.4 Klassen, Typen, Funktionen und Schnittstellen 557
39.3.5 Steuerung und Programmfluss 557
39.3.6 Select Case 559
39.3.7 On Error 560
39.3.8 Fehlende Anweisungen 560
39.4 Weitere .NET-Sprachen 560

40 C#-Ressourcen und die Zukunft 561

40.1 C#-Ressourcen 561
40.1.1 MSDN 561
40.1.2 GotDotNet 561
40.1.3 Csharpindex 561
40.1.4 C-Sharp Corner 561
40.1.5 DotNet Books 562
40.2 C# und die Zukunft 562

Index 565

1 Grundlagen der objektorientierten Programmierung

In diesem Kapitel erhalten Sie eine Einführung in die objektorientierte Programmierung. Diejenigen unter 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.

1.1 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.

1.2 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 Schnittstel-

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

len ohne Implementierung. Die Schnittstellen werden in Kapitel 10, Schnittstellen, behandelt.

1.3 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).

1.4 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 stattdessen 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.

1.5 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:

- 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.
- 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 unbrauchbar zu machen.
- 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 das Debuggen steigt mit der Anzahl der öffentlichen Felder.

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

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

2 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. Code, der in Visual C++ geschrieben wurde, kann vielleicht von Visual Basic aus aufgerufen werden, dies ist jedoch oft nicht einfach zu bewerkstelligen. Visual C++ verwendet die C- und C++-Laufzeitumgebungen, die ein sehr spezifisches Verhalten aufweisen. Visual Basic setzt eine eigene Ausführungsengine ein, die ebenfalls ein ihr eigenes – und abweichendes – Verhalten besitzt.

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 brauchbarenC++- und ein anderer Programmierer Visual Basic-Code schrieb, war die Zusammenarbeit in der Praxis nicht einfach.

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 war üblicherweise nicht ohne Kompromisse möglich. 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), die Zugriff auf alle Funktionen bieten.

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.

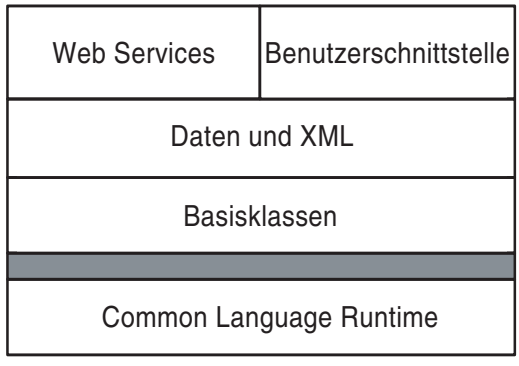

Abbildung 2.1 Aufbau der .NET Frameworks

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 Web Services[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 36, *C# im Detail*.

2.1 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.

2.1.1 Ein einfacheres Programmiermodell

Sämtliche 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 sprachagnostisch und ermöglicht daher eine Sprach-

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

auswahl. Dies vereinfacht sowohl für den Programmierer als auch für den Bibliotheksprovider die Wiederverwendung des Codes.

Die Umgebung unterstützt außerdem die Verwendung von vorhandenem C#-Code, entweder über das Aufrufen von Funktionen in DLLs oder durch Einsatz von COM-Komponenten wie .NET-Laufzeitkomponenten. .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.

Des Weiteren enthält die Umgebung die .NET Frameworks, über die Funktionen bereitgestellt werden, die traditionell in Laufzeitbibliotheken vorhanden sind, plus einige neue Funktionen. Die Frameworks gliedern sich in verschiedene Kategorien.

System

Der Namespace `System` enthält die Hauptklassen der Laufzeitumgebung. Diese Klassen entsprechen grob gesehen denen der C++-Laufzeitbibliothek und beinhalten die folgenden geschachtelten Namespaces:

Namespace	Funktion
`Collections`	Auflistungsobjekte, z. B. Listen, Warteschlangen und Hashtabellen
`Configuration`	Konfigurations- und Installationsobjekte
`Diagnostics`	Debugging und Ablaufverfolgung der Codeausführung
`Globalization`	Globalisierung von Anwendungen
`IO`	Eingabe und Ausgabe
`Net`	Netzwerkoperationen
`Reflection`	Anzeige der Metadaten von Typen und dynamisches Laden und Erstellen von Objekten
`Security`	Unterstützung für das .NET-Sicherheitssystem
`ServiceProcess`	Erstellen und Verwalten von Windows-Diensten
`Text`	Klassen für Codierung und Konvertierung
`Threading`	Threads und Synchronisierung
`Runtime`	Interoperabilität, Remotesteuerung und Serialisierung

System.Data

Der Namespace `System.Data` enthält die Klassen, die für die Unterstützung von Datenbankoperationen benötigt werden.

ABSCHNITT	Funktion
ADO	ADO.NET
Design	Datenbankunterstützung zur Entwicklungszeit
SQL	SQL Server-Unterstützung
SQLTypes	Datentypen für SQL Server

System.Xml

Der Namespace `System.Xml` enthält Klassen zur Verwaltung von XML.

System.Drawing

Der Namespace `System.Drawing` beinhaltet Klassen, die für die Unterstützung von GDI+ erforderlich sind, einschließlich Druck und Bilddarstellung.

System.Web

Der Namespace `System.Web` umfasst Klassen für die Handhabung von Web Services und Klassen zum Erstellen webbasierter Schnittstellen unter Verwendung von ASP.NET.

System.Windows.Forms

Der Namespace `System.Windows.Forms` enthält Klassen zum Erstellen von Rich-Client-Schnittstellen.

2.1.2 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. So wird der Programmierer nicht nur entlastet, in einer Serverumgebung kann auf diese Weise 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 ungeeigneter Stelle ermittelt.

Das Sicherheitssystem interagiert mit dem Prüfcode, um sicherzustellen, dass über den Code nur die Ausführung erfolgt, die ihm 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.

2.1.3 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.

Das Debugging wurde in der .NET-Laufzeitumgebung erheblich verbessert. Das gemeinsame Ausführungsmodell vereinfacht eine sprachübergreifende Fehlersuche, das Debuggen 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-Programmiertasks in der Visual Studio-Umgebung zusammengefasst, d. h., Entwurf, Entwicklung, Debugging und Bereitstellung von Anwendungen werden von hier aus realisiert.

2.1.4 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 von .NET unterstützten Umgebung ausgeführt werden. Darüber hinaus trennt die Umgebung die Anwendungskomponenten, so dass eine Anwendung nur mit den Komponenten läuft, mit denen sie geliefert wurde und nicht mit anderen Versionen, die zum Lieferumfang anderer Anwendungen gehören.

2.2 Metadaten

Metadaten sind das, was die .NET-Laufzeitumgebung zusammenhält. Metadaten stehen in Analogie zu den Typbibliotheken in der COM-Umgebung, 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 Reflexion 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 Reflexion das Aussehen von Objekten ermitteln, nach Attributen suchen oder Methoden ausführen, deren Namen bis zur Laufzeit nicht bekannt sind.

2.3 Assemblierung

In der Vergangenheit konnte ein fertiges Softwarepaket über verschiedene Mechanismen bereitgestellt werden, beispielsweise als ausführbare Datei, als DLL und LIB-Datei oder als DLL mit COM-Objekt und 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 Assembly enthält ein Manifest, in dem die Dateien aufgeführt werden, die in der Assembly enthalten sind. Über das Manifest wird gesteuert, welche Typen und Ressourcen außerhalb der Assembly offen gelegt werden. Des Weiteren werden den Typen und Ressourcen Verweise auf die Dateien zugeordnet, die die Typen und Ressourcen enthalten. Im Manifest werden außerdem die Assemblies aufgelistet, von denen eine Assembly abhängt.

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

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

2.4 Sprachinteroperabilität

Eines der Ziele der .NET-Laufzeitumgebung ist die Sprachagnostik, 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 feststellbar 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, daher wird der Code, der in diesen Sprachen geschrieben wurde, nicht durch die einfacheren Sprachen in seiner Funktionalität eingeschränkt.[4]

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 36, C# im Detail.

2.5 Attribute

Zur Umwandlung einer Klasse in eine Komponente sind häufig weitere Informationen erforderlich, beispielsweise 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, so dass der Programmierer Attribute definieren und verwenden kann.

4. Dies trifft für Managed C++ nicht ganz zu, denn im Vergleich zu C++ müssen einige Einschränkungen hingenommen werden.
5. Jeder, der jemals versucht hat, eine COM-Programmierung ohne Typbibliothek durchzuführen, kennt dieses Problem.

3 Schnelleinstieg und Entwicklung 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 Stichworten in späteren Kapiteln.

Der zweite Teil dieses Kapitels konzentriert sich auf den C#-Compiler und die Vorteile der Verwendung von Visual Studio.NET zum Entwickeln von C#-Anwendungen.

3.1 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");

            // Schleife für Befehlszeilenargumente ausführen
            // und Ausgabe für Befehlszeilenargumente ausführen
        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 bzw. Befehlszeile für diejenigen, die noch nie eine Konsole gesehen haben) eingesetzt wird.

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

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 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. Die `for`-Schleife wird für die übergebenen Parameter ausgeführt und gibt eine Zeile pro Parameter in der Befehlszeile aus.

3.2 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 geschachtelt 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, so dass die Typen leichter referenziert werden können. Das Schlüsselwort `using` ist lediglich ein Shortcut, der den Eingabeaufwand bei der Referenzierung von Elementen verringert. Siehe hierzu die folgende Tabelle:

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.

3.3 Namespace und Assembly

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 Assembly `mscorlib.dll`, die die Hauptfunktionen der Common Language Runtime enthält.

Zur Referenzierung von Objekten, die sich in anderen Assemblies befinden, muss der Name der Assemblydatei an den Compiler übergeben werden. Dies kann über die Befehlszeile und die Option `/r:<Assembly>` 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 Assembly, in der das Objekt gespeichert ist. Die Typen im Namespace `System.Net` beispielsweise befinden sich in der Assembly `System.Net.dll`. Assemblytypen basieren üblicherweise auf den Verwendungsmustern des Objekts in dieser Assembly; ein sehr häufig oder selten verwendeter Typ in einem Namespace kann in einer eigenen Assembly vorliegen.

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

3.4 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# ist 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	ByteS	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`	`Int`-Wert mit Vorzeichen
`Uint`	4	`UInt32`	`Int`-Wert ohne Vorzeichen
`long`	8	`Int64`	Langer `int`-Wert mit Vorzeichen
`ulong`	8	`UInt64`	Langer `int`-Wert ohne Vorzeichen
`float`	4	`Single`	Gleitkommazahl

Typ	ByteS	Laufzeittyp	Beschreibung
`double`	8	`Double`	Gleitkommazahl mit doppelter Genauigkeit
`decimal`	8	`Decimal`	Zahl mit fester Genauigkeit
`string`	n/a	`String`	Unicode-Zeichenfolge
`char`	2	`Char`	Unicode-Zeichen
`bool`	n/a	`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);
    }
}
```

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.Collections` enthalten.

3.5 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 `string`-Typ ist ein Verweistyp. Das folgende Diagramm zeigt die Funktionsweise dieser Typen:

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

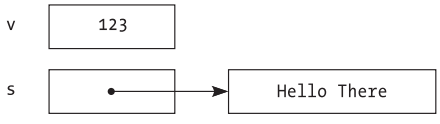

Abbildung 3.1 Wertetyp- und Verweistypzuordnung

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

3.6 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.

3.7 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, Aufzählungen, näher erläutert.

3.8 Delegates und Ereignisse

Delegates sind eine typensichere, objektorientierte Implementierung von Funktionszeigern. Sie werden in vielen Situationen eingesetzt, in denen eine Komponente die Komponente aufrufen muss, von der sie verwendet wird. Delegates werden weitestgehend als Grundlage für Ereignisse eingesetzt, die das einfache Registrieren von Delegates für ein Ereignis ermöglichen. Delegates werden in Kapitel 22 ausführlich behandelt.

In den .NET Frameworks finden Delegates und Ereignisse breite Anwendung.

3.9 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 Radius
    {
        get
        {
            return(radius);
        }
        set
        {
            radius = value;
            Draw();
        }
    }
    public void Draw()
    {
```

```
    }
    int radius;
}
class Test
{
    public static void Main()
    {
        Circle c = new Circle();
        c.Radius = 35;
    }
}
```

In diesem Beispiel wird die Zugriffsroutine `get` oder `set` aufgerufen, wenn die Eigenschaft `Radius` referenziert wird.

3.10 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 Reflexion bezeichnet wird. Neue Attribute können leicht geschrieben und auf Codeelemente angewendet werden (beispielsweise auf Klassen, Mitglieder oder Parameter) und per Reflexion abgerufen werden.

3.11 Entwickeln in C#

Zur Programmierung in C# benötigen Sie eine Möglichkeit zur Erstellung von C#-Programmen. Sie erreichen dies mit einem Befehlszeilencompiler, VS.NET, oder einem C#-Paket für einen Programmiereditor.

3.11.1 Der Befehlszeilencompiler

Den einfachsten Einstieg in das Schreiben von C#-Code bietet das SDK zu .NET Runtime und Frameworks. Das SDK (Software Development Kit) enthält die .NET Runtime und Frameworks sowie Compiler für C#, Visual Basic.NET und die verwalteten Erweiterungen (Managed Extensions) für C++ und JScript.NET. Das Framework kann von der folgenden Website heruntergeladen werden:

```
http://msdn.microsoft.com/net
```

Das SDK ist einfach zu handhaben; Sie schreiben Ihren Code in einem Editor, kompilieren ihn mit Hilfe von CSC und führen ihn aus. Einfach heißt allerdings nicht notwendigerweise auch produktiv. Wenn Sie zur Codierung lediglich das SDK verwenden, werden Sie viel Zeit mit der Dokumentation und der Suche nach einfachen Fehlern im Code verbringen.

Nähere Informationen zum Einsatz des Befehlszeilencompilers finden Sie in Kapitel 38, Der Befehlszeilencompiler.

3.11.2 Visual Studio.NET

Visual Studio.NET ist eine Entwicklungsumgebung, mit der das Programmieren in C# einfach ist und Spaß macht. Betaversionen von Visual Studio.NET können über die gleiche Adresse bezogen werden wie das SDK:

```
http://msdn.microsoft.com/net
```

Wenn Sie Abonnent von MSDN Universal sind, können Sie Visual Studio.NET kostenlos herunterladen; falls nicht, können Sie es gegen eine Schutzgebühr erwerben.

Visual Studio.NET stellt einige wirklich hilfreiche Komponenten für den Entwickler bereit.

Der Editor

Das wichtigste Feature des Visual Studio.NET-Editors ist die IntelliSense-Unterstützung. Eine der Herausforderungen beim Erlernen von C# und den Frameworks ist, eine individuelle Form des Umgangs mit der Sprachsyntax und dem Objektmodell der Frameworks zu entwickeln. Die Autovervollständigung (Auto Completion) ist eine große Hilfe bei der Suche nach der richtigen Methode, die Syntaxprüfung markiert fehlerhafte Codeabschnitte.

Der Formular-Designer

Sowohl auf Windows Forms als auch auf Web Forms basierende Anwendungen werden direkt codiert, ganz ohne separate Ressourcendateien. Dies ermöglicht das Schreiben dieser Anwendungen ohne Visual Studio.NET.

Layout und Eigenschaften »zu Fuß« festzulegen macht keinen großen Spaß, daher stellt Visual Studio einen Formular-Designer bereit, der das einfache Hinzufügen von Steuerelementen zu einem Formular ermöglicht, die Eigenschaften für das Formular festlegt, Ereignisbehandlungsroutinen erstellt usw.

Die Formularbearbeitung erfolgt zweiseitig; Änderungen im Formular-Designer spiegeln sich im Formularcode wider, Änderungen im Formularcode werden auch im Designer vorgenommen.[2]

Das Projektsystem

Das Projektsystem bietet Unterstützung für das Erstellen und Entwickeln von Projekten. Für die meisten Projekttypen sind vordefinierte Vorlagen vorhanden (Windows Forms, Web Forms, Konsolenanwendungen, Klassenbibliotheken usw.).

Das Projektsystem bietet des Weiteren Unterstützung für die Bereitstellung von Anwendungen.

Klassensicht

Während das Projektsystem die Dateien in einem Projekt sichtbar macht, bietet die Klassensicht einen Überblick über die Klassen und andere Typen eines Projekts. Statt die Syntax für eine Eigenschaft direkt zu schreiben, stellt die Klassensicht hierzu einen Assistenten bereit.

Der Objektbrowser

Der Objektbrowser bietet zunächst die gleiche Ansicht wie die Klassensicht, ermöglicht jedoch anstelle einer Codeansicht das Durchsuchen der Komponenten in anderen Assemblies, die das Projekt verwendet. Wenn die Dokumentation einer Klasse unvollständig ist, kann der Objektbrowser die Schnittstelle der Klasse zeigen, wie sie durch die Metadaten dieser Komponente definiert ist.

Der Debugger

Der Debugger von Visual Studio.NET wurde erweitert, um ein sprachübergreifendes Debugging zu ermöglichen. Somit kann ein Debugging von einer Sprache aus für eine andere ausgeführt werden, außerdem kann die Fehlersuche für die Codeausführung von einem Remotecomputer aus erfolgen.

2. Innerhalb eines bestimmten Rahmens

3.11.3 Weitere nennenswerte Tools

Das SDK beinhaltet verschiedene Dienstprogramme, die in der SDK-Dokumentation im Abschnitt zu den .NET Framework-Tools näher beschrieben werden.

ILDASM

ILDASM (IL Disassembler) ist das leistungsstärkste Tool im SDK. Es kann eine Assembly öffnen, alle Typen innerhalb des Assemblies, die für diese Typen definierten Methoden sowie die für diese Methode generierte IL (Intermediate Language) anzeigen.

Dies ist in verschiedener Hinsicht nützlich. Wie der Objektbrowser kann ILDASM dazu verwendet werden, sich den Inhalt einer Assembly anzusehen. Es ist jedoch auch möglich, die spezielle Implementierung einer Methode zu bestimmen. Anhand dieser Fähigkeit können einige Fragen zu C# beantwortet werden.

Wenn Sie beispielsweise wissen möchten, ob C# konstante Zeichenfolgen zur Kompilierungszeit verkettet, kann dies leicht geprüft werden. Zunächst wird ein kleines Programm erstellt:

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

Nach der Kompilierung des Programms kann ILDASM zur Anzeige der IL (Intermediate Language) für `Main()` verwendet werden:

```
.method public hidebysig static void Main() cil managed
{
  .entrypoint
  // Codegröße       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "Hello World"
  IL_0005:  call       void [mscorlib]System.Console::
                       WriteLine(string)
  IL_000a:  ret
} // Ende der Methode Test::Main
```

Selbst ohne umfassende Kenntnis der IL-Sprache ist es relativ klar, dass die zwei Zeichenfolgen zu einer einzigen Zeichenfolge verkettet werden.

Details zu IL finden Sie in der Datei ILinstrset.doc, die Sie im Verzeichnis C:\Programme\Microsoft.NET\FrameworkSDK\Tool Developer Guide finden.

ILDASM kann für jede beliebige Assembly verwendet werden, was einige Fragen hinsichtlich des geistigen Eigentums aufwirft. Obwohl die Codespeicherung in IL das Schreiben von Disassemblern vereinfacht, führt sie doch zu keinem neuen Problem; die x86-Assemblersprache kann auch disassembliert und decodiert werden.

Microsoft ist sich dieses Problems bewusst und arbeitet an neuen Tools, die ein Reverse-Engineering von Assemblies erschweren.

NGEN (PreJIT)

NGEN ist ein Tool, mit dem vor der Programmausführung die Übersetzung von IL in den systemeigenen Prozessorcode erfolgt (statt die Übersetzung nach Bedarf durchzuführen).

Auf den ersten Blick scheint dies eine Möglichkeit zur Umgehung vieler Nachteile des JIT-Ansatzes zu sein: einfach PreJIT für den Code ausführen, anschließend ist die Performance besser und niemand kann die IL decodieren.

Unglücklicherweise funktioniert das so nicht. PreJIT ist lediglich eine Methode zum Speichern der Kompilierungsergebnisse, doch für das Klassenlayout und die Reflexionsunterstützung werden weiterhin Metadaten benötigt. Darüber hinaus ist der generierte systemeigene Code nur für eine spezifische Umgebung gültig. Wenn Änderungen an den Konfigurationseinstellungen (z. B. an der Sicherheitsrichtlinie des Rechners) vorgenommen werden, kehrt das Laufzeitmodul zum normalen JIT zurück.

Obwohl PreJIT den Overhead des JIT-Prozesses beseitigt, wird Code mit geringfügig längeren Ausführungszeiten erzeugt, da eine Ebene indirekten Vorgehens erforderlich ist, die für das normale JIT nicht benötigt wird.

Der wirkliche Vorteil von PreJIT liegt also darin, dass der JIT-Overhead (und damit die Startzeit) einer Clientanwendung verringert wird, anderweitig ist PreJIT nicht von großem Nutzen.

Dies gilt jedenfalls für die erste Version des Laufzeitmoduls. In künftigen Versionen könnte NGEN eine stärker erweiterte Optimierung als das normale JIT liefern, da vermutlich mehr Zeit auf die Analyse verwendet werden kann.

39 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.

39.1 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.

39.1.1 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. Stattdessen werden Verweise verwendet, die denen von C++ ähneln, jedoch nicht alle der C++-Einschränkungen aufweisen.

Quellen werden in Assemblies 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.

39.1.2 .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 Delegates 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.

39.1.3 C#-Anweisungen

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

- Das `new`-Schlüsselwort bedeutet »Abrufen einer neuen Instanz 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 das »Durchfallen« von Code, 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.

39.1.4 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 Reflexion bezeichnet wird.

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

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

39.1.5 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.

39.1.6 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.

39.1.7 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 Erstellung
- Standardargumente für Funktionsparameter

39.2 Unterschiede zwischen C# und Java

C# und Java gehen auf gleiche Wurzeln zurück,[1] 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 C# auf den .NET Frameworks und der .NET-Laufzeit beruht und Java auf den Frameworks und der Laufzeit von Java basiert.

39.2.1 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`	

1. Zusammen mit C, C++ und Pascal gehören sie zu der Sprachfamilie, die geschweifte Klammern verwendet.

C#-Typ	Java-Typ	Kommentar
`double`	`double`	
`char`	`char`	
`string`	`string`	
`object`	`object`	
`byte`		`Byte` ohne Vorzeichen
`ushort`		`Short` ohne Vorzeichen
`uint`		`Int` ohne Vorzeichen
`ulong`		`Long` 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, 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.

39.2.2 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.

39.2.3 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 einen Klassenkonstruktor gesetzt werden.

Basisklassen und Konstruktoren

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

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

Statische Konstruktoren

Anstelle eines statischen Initialisierungsblocks verwendet C# statische Konstruktoren, die mit Hilfe des Schlüsselworts `static` geschrieben werden, das einem parameterlosen Konstruktor vorangestellt wird.

Virtuelle Funktionen, Ausblenden und Versionssteuerung

In C# sind alle Funktionen standardmäßig nicht virtuell. Um eine virtuelle Funktion zu erhalten, muss das Schlüsselwort `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. Da die Versionssteuerung in C+ explizit spezifiziert wird, führt das Hinzufügen einer virtuellen Funktion zu einer Basisklasse nicht zu einer Änderung des Programmverhaltens. 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 Basisklassenimplementierung, was wahrscheinlich falsch ist. In C# setzt das Programm seine Arbeit wie zuvor 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 überschrieben, wenn das Schlüsselwort `override` verwendet wird. Weitere Informationen hierzu finden Sie in Kapitel 11, Versionssteuerung.

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.

Operatorüberladung

C# ermöglicht dem Benutzer das Überladen vieler Operatoren für Klassen und Strukturen, so dass Objekte in mathematischen Ausdrücken eingesetzt werden können. C# bietet jedoch keine Überladung komplexerer Operatoren, beispielsweise Operatoren für Mitgliedszugriff, Funktionsaufruf, Zuordnung oder eine Überladung des `new`-Operators, da eine Überladung in diesen Fällen zu einem sehr komplexen Code führen würde.

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.

Aufzählungen

Der `enum`-Typ in C# ähnelt dem `enum`-Typ in C++ und wird auf ähnliche Weise verwendet. Implizite Konvertierungen zwischen einem `enum` und den zugrunde liegenden Typen unterliegen jedoch stärkeren Einschränkungen.

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`

39.2.4 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.

39.2.5 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 Eigenschaften 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.

39.2.6 Delegates 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 Delegates bereit, die man sich wie typensichere Funktionszeiger vorstellen kann. Eine Klasse kann einen Delegate für eine Funktion der Klasse erstellen, anschließend kann der Delegate an eine Funktion übergeben werden, die diese Zuweisung akzeptiert. Im Anschluss kann die Funktion den Delegate aufrufen.

C# basiert auf Delegates mit Ereignissen, 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 einen Delegate registrieren, der bei Auslösen des Ereignisses aufgerufen werden soll.

39.2.7 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 Reflexion bezeichnet wird.

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

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

39.2.8 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 Assemblies, 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 Assembly 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 Ausdrücken Überläufe ermitteln.

In C# kann eine solche Ermittlung durch die `checked`- und `unchecked`-Anweisungen und über 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 Performance extrem wichtig ist oder eine Integration mit vorhandener Software erforderlich ist, 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 unterschiedliches 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.

39.3 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. VB.NET weist zusätzliche objektorientierte Funktionen auf, daher kann eine Lektüre der VB.NET-Dokumentation sehr aufschlussreich sein.

39.3.1 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.

39.3.2 Datentypen und Variablen

Obwohl viele der in VB und C# verwendeten Datentypen übereinstimmen, gibt es einige wichtige Unterschiede, und ein ähnlicher Name kann 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.[2]

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

2. Der Typ `object` kann einen beliebigen Typ enthalten, es ist jedoch bekannt, welcher Typ enthalten ist.

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 bzw. die Konvertierung fehlschlagen kann. 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.

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.

39.3.3 Operatoren und Ausdrücke

Die C#-Operatoren weisen gegenüber Visual Basic einige 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.

39.3.4 Klassen, Typen, Funktionen und Schnittstellen

Da C# eine objektorientierte Sprache ist,[3] 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 `ByRef` 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.

39.3.5 Steuerung und Programmfluss

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

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
```

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

```
{
    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
Next
```

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.

39.3.6 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;
}
```

39.3.7 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.

39.3.8 Fehlende Anweisungen

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

39.4 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« (Managed Extensions) 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++-Konstrukte 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 Konstrukten, 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.

Neben den Microsoft-Sprachen wurde außerdem eine .NET-Plattformunterstützung für eine Vielzahl von Drittanbietersprachen angekündigt. Weitere Informationen finden Sie unter `http://www.gotdotnet.com`.