C#

**Sprachspezifikation**

**Version**

**Hinweis**

© 1999-2012 Microsoft Corporation. Alle Rechte vorbehalten.

Microsoft, Windows, Visual Basic, Visual C# und Visual C++ sind entweder eingetragene Marken oder Marken der Microsoft Corporation in den USA und/oder anderen Ländern.

Weitere in diesem Dokument aufgeführte Produkt- und Firmennamen sind möglicherweise geschützte Marken ihrer jeweiligen Inhaber.

**Inhaltsverzeichnis**

1. Einführung 1

1.1 Hello, World 1

1.2 Programmstruktur 2

1.3 Typen und Variablen 4

1.4 Ausdrücke 8

1.5 Anweisungen 10

1.6 Klassen und Objekte 14

1.6.1 Member 15

1.6.2 Barrierefreiheit 15

1.6.3 Typparameter 15

1.6.4 Basisklassen 16

1.6.5 Felder 16

1.6.6 Methoden 17

1.6.6.1 Parameter 17

1.6.6.2 Methodentext und lokale Variablen 19

1.6.6.3 Statische Methoden und Instanzmethoden 19

1.6.6.4 Virtuelle und abstrakte Methoden sowie Überschreibungsmethoden 20

1.6.6.5 Methodenüberladungen 23

1.6.7 Weitere Funktionsmember 23

1.6.7.1 Konstruktoren 25

1.6.7.2 Eigenschaften 26

1.6.7.3 Indexer 26

1.6.7.4 Ereignisse 27

1.6.7.5 Operatoren 27

1.6.7.6 Destruktoren 28

1.7 Strukturen 28

1.8 Arrays 30

1.9 Schnittstellen 31

1.10 Enumerationen 32

1.11 Delegaten 33

1.12 Attribute 34

2. Lexikalische Struktur 37

2.1 Programme 37

2.2 Grammatiken 37

2.2.1 Grammatiknotation 37

2.2.2 Lexikalische Grammatik 38

2.2.3 Syntaktische Grammatik 38

2.3 Lexikalische Analyse 38

2.3.1 Zeilenabschlusszeichen 39

2.3.2 Kommentare 40

2.3.3 Leerraum 41

2.4 Token 41

2.4.1 Escapesequenzen für Unicode-Zeichen 41

2.4.2 Bezeichner 42

2.4.3 Schlüsselwörter 44

2.4.4 Literale 45

2.4.4.1 Boolesche Literale 45

2.4.4.2 Ganzzahlliterale 45

2.4.4.3 Reelle Literale 46

2.4.4.4 Zeichenliterale 47

2.4.4.5 Zeichenfolgenliterale 48

2.4.4.6 Das NULL-Literal 50

2.4.5 Operatoren und Markierungszeichen 50

2.5 Präprozessordirektiven 50

2.5.1 Symbole für bedingte Kompilierung 52

2.5.2 Präprozessorausdrücke 52

2.5.3 Deklarationsdirektiven 53

2.5.4 Direktiven für bedingte Kompilierung 54

2.5.5 Diagnosedirektiven 56

2.5.6 Region-Direktiven 57

2.5.7 Line-Direktiven 57

2.5.8 pragma-Direktiven 58

2.5.8.1 pragma-Warnung 58

3. Grundlegende Konzepte 61

3.1 Anwendungsstart 61

3.2 Anwendungsbeendigung 62

3.3 Deklarationen 62

3.4 Member 65

3.4.1 Namespacemember 65

3.4.2 Strukturmember 65

3.4.3 Enumerationsmember 66

3.4.4 Klassenmember 66

3.4.5 Schnittstellenmember 66

3.4.6 Arraymember 66

3.4.7 Delegatmember 66

3.5 Memberzugriff 66

3.5.1 Deklarierter Zugriff 66

3.5.2 Zugriffsdomänen 67

3.5.3 Geschützter Zugriff für Instanzmember 70

3.5.4 Zugriffsbeschränkungen 71

3.6 Signaturen und Überladung 72

3.7 Gültigkeitsbereiche 73

3.7.1 Verdecken von Namen 76

3.7.1.1 Verdecken durch Schachtelung 77

3.7.1.2 Verdecken durch Vererbung 77

3.8 Namespace- und Typnamen 79

3.8.1 Vollqualifizierte Namen 81

3.9 Automatische Speicherverwaltung 82

3.10 Ausführungsreihenfolge 84

4. Typen 87

4.1 Werttypen 87

4.1.1 Der System.ValueType-Typ 88

4.1.2 Standardkonstruktoren 88

4.1.3 Strukturtypen 89

4.1.4 Einfache Typen 89

4.1.5 Ganzzahlige Typen 90

4.1.6 Gleitkommatypen 92

4.1.7 Der decimal-Typ 93

4.1.8 Der bool-Typ 94

4.1.9 Enumerationstypen 94

4.1.10 Typen, die NULL-Werte zulassen 94

4.2 Verweistypen 95

4.2.1 Klassentypen 95

4.2.2 Der object-Typ 96

4.2.3 Der dynamische Typ 96

4.2.4 Der string-Typ 96

4.2.5 Schnittstellentypen 96

4.2.6 Arraytypen 96

4.2.7 Delegattypen 97

4.3 Boxing und Unboxing 97

4.3.1 Boxingkonvertierungen 97

4.3.2 Unboxingkonvertierungen 98

4.4 Konstruierte Typen 99

4.4.1 Typargumente 100

4.4.2 Offene und geschlossene Typen 100

4.4.3 Gebundene und ungebundene Typen 101

4.4.4 Erfüllen von Einschränkungen 101

4.5 Typparameter 102

4.6 Ausdrucksbaumstrukturtypen 103

4.7 Der dynamische Typ 104

5. Variablen 105

5.1 Variablenkategorien 105

5.1.1 Statische Variablen 105

5.1.2 Instanzvariablen 105

5.1.2.1 Instanzvariablen in Klassen 105

5.1.2.2 Instanzvariablen in Strukturen 106

5.1.3 Arrayelemente 106

5.1.4 Werteparameter 106

5.1.5 Verweisparameter 106

5.1.6 Ausgabeparameter 106

5.1.7 Lokale Variablen 107

5.2 Standardwerte 108

5.3 Definitive Zuweisung 108

5.3.1 Anfänglich zugewiesene Variablen 109

5.3.2 Anfänglich nicht zugewiesene Variablen 109

5.3.3 Exakte Regeln zur Ermittlung von definitiven Zuweisungen 109

5.3.3.1 Allgemeine Regeln für Anweisungen 110

5.3.3.2 Blockanweisungen, die checked-Anweisung und die unchecked-Anweisung 110

5.3.3.3 Ausdrucksanweisungen 111

5.3.3.4 Deklarationsanweisungen 111

5.3.3.5 If-Anweisungen 111

5.3.3.6 Switch-Anweisungen 111

5.3.3.7 While-Anweisung 111

5.3.3.8 Do-Anweisungen 112

5.3.3.9 For-Anweisungen 112

5.3.3.10 Break-, continue- und goto-Anweisungen 112

5.3.3.11 Throw-Anweisungen 112

5.3.3.12 Return-Anweisungen 113

5.3.3.13 Try-catch-Anweisungen 113

5.3.3.14 Try-finally-Anweisungen 113

5.3.3.15 Try-catch-finally-Anweisungen 114

5.3.3.16 Foreach-Anweisungen 114

5.3.3.17 Using-Anweisungen 115

5.3.3.18 Lock-Anweisungen 115

5.3.3.19 Yield-Anweisungen 115

5.3.3.20 Allgemeine Regeln für einfache Ausdrücke 115

5.3.3.21 Allgemeine Regeln für Ausdrücke mit eingebetteten Ausdrücken 115

5.3.3.22 Aufrufausdrücke und Objekterstellungsausdrücke 116

5.3.3.23 Einfache Zuweisungsausdrücke 116

5.3.3.24 &&-Ausdrücke 117

5.3.3.25 ||-Ausdrücke 117

5.3.3.26 ! Ausdrücke 118

5.3.3.27 ??-Ausdrücke 119

5.3.3.28 ?:-Ausdrücke 119

5.3.3.29 Anonyme Funktionen 120

5.4 Variablenverweise 120

5.5 Unteilbarkeit von Variablenverweisen 120

6. Konvertierungen 123

6.1 Implizite Konvertierungen 123

6.1.1 Identitätskonvertierungen 124

6.1.2 Implizite numerische Konvertierungen 124

6.1.3 Implizite Enumerationskonvertierungen 124

6.1.4 Implizite Konvertierungen, die NULL-Werte zulassen 124

6.1.5 NULL-Literalkonvertierungen 125

6.1.6 Implizite Verweiskonvertierungen 125

6.1.7 Boxingkonvertierungen 126

6.1.8 Implizite dynamische Konvertierungen 126

6.1.9 Implizite Konvertierungen konstanter Ausdrücke 127

6.1.10 Implizite Konvertierungen mit Typparametern 127

6.1.11 Benutzerdefinierte implizite Konvertierungen 127

6.1.12 Konvertierungen anonymer Funktionen und Methodengruppen 128

6.2 Explizite Konvertierungen 128

6.2.1 Explizite numerische Konvertierungen 128

6.2.2 Explizite Enumerationskonvertierungen 130

6.2.3 Explizite Konvertierungen, die NULL-Werte zulassen 130

6.2.4 Explizite Verweiskonvertierungen 131

6.2.5 Unboxingkonvertierungen 132

6.2.6 Explizite dynamische Konvertierungen 132

6.2.7 Explizite Konvertierungen mit Typparametern 133

6.2.8 Benutzerdefinierte explizite Konvertierungen 134

6.3 Standardkonvertierungen 134

6.3.1 Implizite Standardkonvertierungen 134

6.3.2 Explizite Standardkonvertierungen 134

6.4 Benutzerdefinierte Konvertierungen 135

6.4.1 Zugelassene benutzerdefinierte Konvertierungen 135

6.4.2 Heraufgestufte Konvertierungsoperatoren 135

6.4.3 Auswertung benutzerdefinierter Konvertierungen 135

6.4.4 Benutzerdefinierte implizite Konvertierungens 136

6.4.5 Benutzerdefinierte explizite Konvertierungens 137

6.5 Konvertierungen anonymer Funktionen 138

6.5.1 Auswertung von Konvertierungen anonymer Funktionen in Delegattypen 140

6.5.2 Auswertung von Konvertierungen anonymer Funktionen in Ausdrucksbaumstrukturtypen 140

6.5.3 Implementierungsbeispiel 140

6.6 Konvertierungen von Methodengruppen 143

7. Ausdrücke 147

7.1 Ausdrucksklassifizierungen 147

7.1.1 Werte von Ausdrücken 148

7.2 Statische und dynamische Bindung 148

7.2.1 Bindungszeit 149

7.2.2 Dynamische Bindung 149

7.2.3 Typen von enthaltenden Ausdrücken 150

7.3 Operatoren 150

7.3.1 Operatorvorrang und -orientierung 150

7.3.2 Überladen von Operatoren 152

7.3.3 Überladungsauflösung unärer Operatoren 153

7.3.4 Überladungsauflösung binärer Operatoren 153

7.3.5 Benutzerdefinierte Kandidatenoperatoren 154

7.3.6 Numerische Erweiterungen 154

7.3.6.1 Unäre numerische Erweiterungen 154

7.3.6.2 Binäre numerische Erweiterungen 154

7.3.7 Heraufgestufte Operatoren 155

7.4 Membersuche 156

7.4.1 Basistypen 158

7.5 Funktionsmember 158

7.5.1 Argumentlisten 161

7.5.1.1 Zusammengehörige Parameter 162

7.5.1.2 Laufzeitauswertung von Argumentlisten 163

7.5.2 Typrückschluss 164

7.5.2.1 Die erste Phase 165

7.5.2.2 Die zweite Phase 165

7.5.2.3 Eingabetypen 166

7.5.2.4 Ausgabetypen 166

7.5.2.5 Abhängigkeit 166

7.5.2.6 Ausgabetyprückschluss 166

7.5.2.7 Explizite Parametertyprückschlüsse 167

7.5.2.8 Genaue Rückschlüsse 167

7.5.2.9 Rückschlüsse auf Untergrenzen 167

7.5.2.10 Rückschlüsse auf Obergrenzen 168

7.5.2.11 Fixieren 168

7.5.2.12 Hergeleiteter Rückgabetyp 169

7.5.2.13 Typrückschluss für Konvertierungen von Methodengruppen 170

7.5.2.14 Ermitteln des optimalen Typs für einen Satz von Ausdrücken 171

7.5.3 Überladungsauflösung 171

7.5.3.1 Anwendbare Funktionsmember 171

7.5.3.2 Besserer Funktionsmember 172

7.5.3.3 Bessere Konvertierung von Ausdrücken 173

7.5.3.4 Bessere Konvertierung von Typen 174

7.5.3.5 Besseres Konvertierungsziel 174

7.5.3.6 Überladen in generischen Klassen 174

7.5.4 Prüfung der dynamischen Überladungsauflösung bei der Kompilierung 175

7.5.5 Funktionsmemberaufruf 175

7.5.5.1 Aufrufe in geschachtelten Instanzen 176

7.6 Primäre Ausdrücke 177

7.6.1 Literale 177

7.6.2 Einfache Namen 177

7.6.2.1 Unveränderliche Bedeutung in Blöcken 179

7.6.3 Ausdrücke in Klammern 180

7.6.4 Memberzugriff 180

7.6.4.1 Identische einfache Namen und Typnamen 182

7.6.4.2 Grammatikalische Mehrdeutigkeiten 183

7.6.5 Aufrufausdrücke 183

7.6.5.1 Methodenaufrufe 184

7.6.5.2 Erweiterungsmethodenaufrufe 185

7.6.5.3 Delegataufrufe 188

7.6.6 Elementzugriff 188

7.6.6.1 Arrayzugriff 188

7.6.6.2 Indexerzugriff 189

7.6.7 This-Zugriff 190

7.6.8 Basiszugriff 191

7.6.9 Postfix-Inkrement- und -Dekrementoperatoren 191

7.6.10 Der new-Operator 192

7.6.10.1 Objekterstellungsausdrücke 192

7.6.10.2 Objektinitialisierer 194

7.6.10.3 Auflistungsinitialisierer 196

7.6.10.4 Arrayerstellungsausdrücke 197

7.6.10.5 Delegaterstellungsausdrücke 200

7.6.10.6 Anonyme Objekterstellungsausdrücke 201

7.6.11 Der typeof-Operator 203

7.6.12 Der checked-Operator und der unchecked-Operator 205

7.6.13 Standardwertausdrücke 207

7.6.14 Ausdrücke für anonyme Methoden 207

7.7 Unäre Operatoren 207

7.7.1 Unärer Plus-Operator 208

7.7.2 Unärer Minus-Operator 208

7.7.3 Logischer Negationsoperator 209

7.7.4 Bitweiser Komplementoperator 209

7.7.5 Präfix-Inkrement- und -Dekrementoperatoren 209

7.7.6 Umwandlungsausdrücke 210

7.7.7 Await-Ausdrücke 211

7.7.7.1 Awaitable-Ausdrücke 211

7.7.7.2 Klassifizierungen von Await-Ausdrücken 212

7.7.7.3 Laufzeitauswertung von Await-Ausdrücken 212

7.8 Arithmetische Operatoren 212

7.8.1 Multiplikationsoperator 213

7.8.2 Divisionsoperator 214

7.8.3 Restoperator 215

7.8.4 Additionsoperator 216

7.8.5 Subtraktionsoperator 218

7.9 Verschiebeoperatoren 220

7.10 Relationale und Typtestoperatoren 221

7.10.1 Vergleichsoperatoren für ganze Zahlen 222

7.10.2 Vergleichsoperatoren für Gleitkommazahlen 222

7.10.3 Vergleichsoperatoren für Dezimalzahlen 223

7.10.4 Boolesche Gleichheitsoperatoren 223

7.10.5 Vergleichsoperatoren für Enumerationen 224

7.10.6 Gleichheitsoperatoren für Verweistypen 224

7.10.7 Gleichheitsoperatoren für Zeichenfolgen 226

7.10.8 Gleichheitsoperatoren für Delegaten 226

7.10.9 Gleichheitsoperatoren und NULL 226

7.10.10 Der is-Operator 227

7.10.11 Der as-Operator 227

7.11 Logische Operatoren 228

7.11.1 Logische Operatoren für ganze Zahlen 229

7.11.2 Logische Operatoren für Enumerationen 229

7.11.3 Boolesche logische Operatoren 229

7.11.4 Boolesche logische Operatoren, die NULL-Werte zulassen 229

7.12 Logische Bedingungsoperatoren 230

7.12.1 Boolesche logische Bedingungsoperatoren 231

7.12.2 Benutzerdefinierte logische Bedingungsoperatoren 231

7.13 Der NULL-Sammeloperator 231

7.14 Bedingungsoperator 232

7.15 Konvertierungen anonymer Ausdrücke 233

7.15.1 Signaturen anonymer Funktionen 235

7.15.2 Anonyme Funktionstexte 236

7.15.3 Überladungsauflösung 236

7.15.4 Anonyme Funktionen und dynamische Bindungen 237

7.15.5 Äußere Variablen 237

7.15.5.1 Erfasste äußere Variablen 237

7.15.5.2 Instanziierung von lokalen Variablen 238

7.15.6 Auswertung anonymer Funktionsausdrücke 240

7.16 Abfrageausdrücke 240

7.16.1 Mehrdeutigkeiten in Abfrageausdrücken 242

7.16.2 Übersetzung von Abfrageausdrücken 242

7.16.2.1 Select- und groupby-Klauseln mit Fortsetzungen 242

7.16.2.2 Explizite Bereichsvariablentypen 243

7.16.2.3 Degenerieren von Abfrageausdrücken 244

7.16.2.4 Die Klauseln from, let, where, join und orderby 244

7.16.2.5 Select- Klauseln 247

7.16.2.6 Groupby-Klauseln 248

7.16.2.7 Transparente Bezeichner 248

7.16.3 Das Muster von Abfrageausdrücken 250

7.17 Zuweisungsoperatoren 251

7.17.1 Einfache Zuweisung 251

7.17.2 Verbundzuweisung 253

7.17.3 Ereigniszuweisung 254

7.18 Ausdruck 255

7.19 Konstante Ausdrücke 255

7.20 Boolesche Ausdrücke 256

8. Anweisungen 259

8.1 Endpunkte und Erreichbarkeit 259

8.2 Blöcke 261

8.2.1 Anweisungslisten 262

8.3 Die leere Anweisung 262

8.4 Anweisung mit Bezeichnungen 262

8.5 Deklarationsanweisungen 263

8.5.1 Deklaration von lokalen Variablen 263

8.5.2 Deklaration von lokalen Konstanten 265

8.6 Ausdrucksanweisungen 265

8.7 Auswahlanweisungen 266

8.7.1 Die if-Anweisung 266

8.7.2 Die switch-Anweisung 267

8.8 Iterationsanweisungen 270

8.8.1 Die while-Anweisung 271

8.8.2 Die do-Anweisung 271

8.8.3 Die for-Anweisung 272

8.8.4 Die foreach-Anweisung 273

8.9 Sprunganweisungen 276

8.9.1 Die break-Anweisung 277

8.9.2 Die continue-Anweisung 278

8.9.3 Die goto-Anweisung 278

8.9.4 Die return-Anweisung 280

8.9.5 Die throw-Anweisung 280

8.10 Die try-Anweisung 282

8.11 Die checked-Anweisung und die unchecked-Anweisung 285

8.12 Die lock-Anweisung 285

8.13 Die using-Anweisung 286

8.14 Die yield-Anweisung 288

9. Namespaces 290

9.1 Kompilationseinheiten 290

9.2 Namespacedeklarationen 290

9.3 extern-Aliase 292

9.4 Using-Direktiven 292

9.4.1 Using-Alias-Direktiven 293

9.4.2 Using-Namespace-Direktiven 296

9.5 Namespacemember 297

9.6 Typdeklarationen 298

9.7 Namespacealias-Qualifizierer 298

9.7.1 Eindeutigkeit von Aliasen 299

10. Klassen 301

10.1 Klassendeklarationen 301

10.1.1 Klassenmodifizierer 301

10.1.1.1 Abstrakte Klassen 302

10.1.1.2 Versiegelte Klassen 302

10.1.1.3 Statische Klassen 303

10.1.2 Partial-Modifizierer 303

10.1.3 Typparameter 304

10.1.4 Basisspezifikation für Klassen 304

10.1.4.1 Basisklassen 304

10.1.4.2 Schnittstellenimplementierungen 306

10.1.5 Typparametereinschränkungen 306

10.1.6 Klassendefinition 311

10.2 Partielle Typen 311

10.2.1 Attribute 311

10.2.2 Modifizierer 311

10.2.3 Typparameter und Einschränkungen 312

10.2.4 Basisklasse 312

10.2.5 Basisschnittstellen 313

10.2.6 Member 313

10.2.7 Partielle Methoden 314

10.2.8 Namensbindung 316

10.3 Klassenmember 316

10.3.1 Der Instanztyp 318

10.3.2 Member konstruierter Typen 318

10.3.3 Vererbung 319

10.3.4 Der new-Modifizierer 320

10.3.5 Zugriffsmodifizierer 320

10.3.6 Konstituierende Typen 321

10.3.7 Statische Member und Instanzmember 321

10.3.8 Geschachtelte Typen 322

10.3.8.1 Vollqualifizierter Name 322

10.3.8.2 Deklarierter Zugriff 322

10.3.8.3 Verdecken 323

10.3.8.4 This-Zugriff 324

10.3.8.5 Zugriff auf private und geschützte Member des enthaltenden Typs 324

10.3.8.6 Geschachtelte Typen in generischen Klassen 325

10.3.9 Reservierte Membernamen 326

10.3.9.1 Für Eigenschaften reservierte Membernamen 326

10.3.9.2 Für Ereignisse reservierte Membernamen 327

10.3.9.3 Für Indexer reservierte Membernamen 327

10.3.9.4 Für Destruktoren reservierte Membernamen 327

10.4 Konstanten 327

10.5 Felder 329

10.5.1 Statische Felder und Instanzfelder 330

10.5.2 Schreibgeschützte Felder 331

10.5.2.1 Verwenden von statischen schreibgeschützten Feldern für Konstanten 331

10.5.2.2 Versionskontrolle von Konstanten und statischen schreibgeschützten Feldern 332

10.5.3 Flüchtige Felder 332

10.5.4 Feldinitialisierung 334

10.5.5 Variableninitialisierungen 334

10.5.5.1 Initialisierung statischer Felder 335

10.5.5.2 Initialisierung von Instanzfeldern 336

10.6 Methoden 337

10.6.1 Methodenparameter 339

10.6.1.1 Werteparameter 341

10.6.1.2 Verweisparameter 341

10.6.1.3 Ausgabeparameter 342

10.6.1.4 Parameterarrays 343

10.6.2 Statische Methoden und Instanzmethoden 345

10.6.3 Virtuelle Methoden 345

10.6.4 Überschreibungsmethoden 347

10.6.5 Versiegelte Methoden 350

10.6.6 Abstrakte Methoden 350

10.6.7 Externe Methoden 352

10.6.8 Partielle Methoden 352

10.6.9 Erweiterungsmethoden 352

10.6.10 Methodentext 353

10.6.11 Methodenüberladungen 354

10.7 Eigenschaften 354

10.7.1 Statische Eigenschaften und Instanzeigenschaften 355

10.7.2 Accessoren 356

10.7.3 Automatisch implementierte Eigenschaften 361

10.7.4 Barrierefreiheit 362

10.7.5 Virtuelle, versiegelte, überschreibungsbezogene und abstrakte Accessoren 363

10.8 Ereignisse 365

10.8.1 Feldähnliche Ereignisse 367

10.8.2 Ereignisaccessoren 368

10.8.3 Statische Ereignisse und Instanzereignisse 370

10.8.4 Virtuelle, versiegelte, überschreibungsbezogene und abstrakte Accessoren 370

10.9 Indexer 371

10.9.1 Indexerüberladung 374

10.10 Operatoren 374

10.10.1 Unäre Operatoren 376

10.10.2 Binäre Operatoren 377

10.10.3 Konvertierungsoperatoren 377

10.11 Instanzkonstruktoren 380

10.11.1 Konstruktorinitialisierungen 381

10.11.2 Instanzvariableninitialisierungen 382

10.11.3 Konstruktorausführung 382

10.11.4 Standardkonstruktoren 384

10.11.5 Private Konstruktoren 384

10.11.6 Parameter für optionale Instanzkonstruktoren 384

10.12 Statische Konstruktoren 385

10.13 Destruktoren 387

10.14 Iteratoren 389

10.14.1 Enumeratorschnittstellen 389

10.14.2 Aufzählbare Schnittstellen 389

10.14.3 Ausgabetyp 389

10.14.4 Enumeratorobjekte 389

10.14.4.1 Die MoveNext-Methode 390

10.14.4.2 Die Current-Eigenschaft 391

10.14.4.3 Die Dispose-Methode 391

10.14.5 Aufzählbare Objekte 392

10.14.5.1 Die GetEnumerator-Methode 392

10.14.6 Implementierungsbeispiel 392

10.15 Async-Funktionen 399

10.15.1 Auswertung einer Async-Funktion, die eine Aufgabe zurückgibt 399

10.15.2 Auswertung einer Async-Funktion, die "void" zurückgibt 399

11. Strukturen 401

11.1 Strukturdeklarationen 401

11.1.1 Strukturmodifizierer 401

11.1.2 Partial-Modifizierer 402

11.1.3 Strukturschnittstellen 402

11.1.4 Strukturkörper 402

11.2 Strukturmember 402

11.3 Unterschiede zwischen Klassen und Strukturen 402

11.3.1 Wertsemantik 403

11.3.2 Vererbung 404

11.3.3 Zuweisung 404

11.3.4 Standardwerte 404

11.3.5 Boxing und Unboxing 405

11.3.6 Bedeutung von "this" 406

11.3.7 Feldinitialisierungen 407

11.3.8 Konstruktoren 407

11.3.9 Destruktoren 408

11.3.10 Statische Konstruktoren 408

11.4 Strukturbeispiele 408

11.4.1 Datenbankganzzahltyp 408

11.4.2 Boolescher Datenbanktyp 410

12. Arrays 413

12.1 Arraytypen 413

12.1.1 Der System.Array-Typ 414

12.1.2 Arrays und die generische IList-Schnittstelle 414

12.2 Arrayerstellung 415

12.3 Arrayelementzugriff 415

12.4 Arraymember 415

12.5 Arraykovarianz 415

12.6 Arrayinitialisierungen 416

13. Schnittstellen 419

13.1 Schnittstellendeklarationen 419

13.1.1 Schnittstellenmodifizierer 419

13.1.2 Partial-Modifizierer 420

13.1.3 Listen der varianten Typparameter 420

13.1.3.1 Varianzsicherheit 420

13.1.3.2 Varianzkonvertierung 421

13.1.4 Basisschnittstellen 421

13.1.5 Schnittstellendefinition 422

13.2 Schnittstellenmember 422

13.2.1 Schnittstellenmethoden 423

13.2.2 Schnittstelleneigenschaften 424

13.2.3 Schnittstellenereignisse 424

13.2.4 Schnittstellenindexer 424

13.2.5 Schnittstellenmemberzugriff 425

13.3 Vollqualifizierte Schnittstellenmembernamen 426

13.4 Schnittstellenimplementierungen 427

13.4.1 Explizite Implementierungen von Schnittstellenmembern 428

13.4.2 Eindeutigkeit von implementierten Schnittstellen 430

13.4.3 Implementierungen von generischen Methoden 431

13.4.4 Schnittstellenzuordnung 432

13.4.5 Vererbung der Schnittstellenimplementierung 435

13.4.6 Erneute Implementierung von Schnittstellen 436

13.4.7 Abstrakte Klassen und Schnittstellen 437

14. Enumerationen 439

14.1 Enumerationsdeklarationen 439

14.2 Enumerationsmodifizierer 439

14.3 Enumerationsmember 440

14.4 Der System.Enum-Typ 442

14.5 Enumerationswerte und -operationen 442

15. Delegaten 443

15.1 Delegatdeklarationen 443

15.2 Kompatibilität von Delegaten 445

15.3 Delegatinstanziierung 446

15.4 Delegataufruf 446

16. Ausnahmen 449

16.1 Ursachen von Ausnahmen 449

16.2 Die System.Exception-Klasse 449

16.3 Behandeln von Ausnahmen 449

16.4 Häufige Ausnahmeklassen 450

17. Attribute 453

17.1 Attributklassen 453

17.1.1 Attributverwendung 453

17.1.2 Positionelle und benannte Parameter 454

17.1.3 Attributparametertypen 455

17.2 Attributspezifikation 455

17.3 Attributinstanzen 461

17.3.1 Kompilieren eines Attributs 461

17.3.2 Abrufen einer Attributinstanz zur Laufzeit 461

17.4 Reservierte Attribute 462

17.4.1 Das AttributeUsage-Attribut 462

17.4.2 Das Conditional-Attribut 463

17.4.2.1 Bedingte Methoden 463

17.4.2.2 Bedingte Attributklassen 465

17.4.3 Das Obsolete-Attribut 466

17.4.4 Aufruferinformationsattribute 467

17.4.4.1 Das CallerLineNumber-Attribut 468

17.4.4.2 Das CallerFilePath-Attribut 468

17.4.4.3 Das CallerMemberName-Attribut 468

17.5 Attribute für die Interoperabilität 469

17.5.1 Interoperabilität mit COM- und Win32-Komponenten 469

17.5.2 Interoperabilität mit anderen .NET-Sprachen 469

17.5.2.1 Das IndexerName-Attribut 469

18. Unsicherer Code 471

18.1 Unsichere Kontexte 471

18.2 Zeigertypen 474

18.3 Feste und bewegliche Variablen 476

18.4 Zeigerkonvertierungen 477

18.4.1 Zeigerarrays 478

18.5 Zeiger in Ausdrücken 479

18.5.1 Zeigerdereferenzierung 479

18.5.2 Zeigermemberzugriff 480

18.5.3 Zeigerelementzugriff 481

18.5.4 Der address-of-Operator 481

18.5.5 Inkrementieren und Dekrementieren von Zeigern 482

18.5.6 Zeigerarithmetik 482

18.5.7 Zeigervergleich 483

18.5.8 Der sizeof-Operator 484

18.6 Die fixed-Anweisung 484

18.7 Puffer mit fester Größe 488

18.7.1 Deklarationen von Puffern mit fester Größe 488

18.7.2 Puffer mit fester Größe in Ausdrücken 490

18.7.3 Überprüfung des definitiven Zuweisungszustands 490

18.8 Stapelreservierung 491

18.9 Dynamische Speicherbelegung 492

A. Dokumentationskommentare 495

A.1 Einführung 495

A.2 Empfohlene Tags 496

A.2.1 <c> 497

A.2.2 <code> 498

A.2.3 <example> 498

A.2.4 <exception> 498

A.2.5 <include> 499

A.2.6 <list> 500

A.2.7 <para> 500

A.2.8 <param> 501

A.2.9 <paramref> 501

A.2.10 <permission> 502

A.2.11 <remark> 502

A.2.12 <returns> 502

A.2.13 <see> 503

A.2.14 <seealso> 503

A.2.15 <summary> 504

A.2.16 <value> 504

A.2.17 <typeparam> 504

A.2.18 <typeparamref> 505

A.3 Verarbeiten der Dokumentationsdatei 505

A.3.1 ID-Zeichenfolgenformat 505

A.3.2 ID-Zeichenfolgenbeispiele 507

A.4 Ein Beispiel 510

A.4.1 C#-Quellcode 510

A.4.2 Resultierende XML 512

B. Grammatik 518

B.1 Lexikalische Grammatik 518

B.1.1 Zeilenabschlusszeichen 518

B.1.2 Kommentare 518

B.1.3 Leerraum 519

B.1.4 Token 519

B.1.5 Escapesequenzen für Unicode-Zeichen 519

B.1.6 Bezeichner 519

B.1.7 Schlüsselwörter 520

B.1.8 Literale 521

B.1.9 Operatoren und Markierungszeichen 523

B.1.10 Präprozessordirektiven 523

B.2 Syntaktische Grammatik 525

B.2.1 Grundlegende Konzepte 525

B.2.2 Typen 525

B.2.3 Variablen 527

B.2.4 Ausdrücke 527

B.2.5 Anweisungen 534

B.2.6 Namespaces 537

B.2.7 Klassen 538

B.2.8 Strukturen 545

B.2.9 Arrays 546

B.2.10 Schnittstellen 546

B.2.11 Enumerationen 547

B.2.12 Delegaten 548

B.2.13 Attribute 548

B.3 Grammatische Erweiterungen für unsicheren Code 550

C. Referenzen 553

# Einführung

C# (gesprochen „Si Scharp“) ist eine einfache, moderne, objektorientierte und typsichere Programmiersprache. C# gehört zur Familie der C-Programmiersprachen und wird C-, C++- und Java-Programmierern unmittelbar vertraut erscheinen. C# wurde von ECMA International als ECMA-334 und von ISO/IEC als ISO/IEC 23270 standardisiert. Der Microsoft C#-Compiler für .NET Framework stellt eine standardkonforme Implementierung dieser beiden Standards dar.

Bei C# handelt es sich um eine objektorientierte Sprache. C# mit seiner Unterstützung für komponentenorientierte Programmierung geht darüber noch hinaus. Zeitgemäße Softwareentwicklung baut zunehmend auf Softwarekomponenten in Form abgeschlossener und selbstbeschreibender Funktionsblöcke auf. Bei diesen Komponenten ist entscheidend, dass sie ein Programmiermodell mit Eigenschaften, Methoden und Ereignissen bieten, über Attribute verfügen, die deklarative Informationen über die Komponente bereitstellen, und ihre Dokumentation bereits integriert ist. Die Sprachkonstrukte in C# unterstützen diese Konzepte von Haus aus. Dies macht C# zu einer äußerst intuitiven Sprache für die Erstellung und Verwendung von Softwarekomponenten.

Verschiedene Funktionen von C# unterstützen die Entwicklung stabiler und langlebiger Anwendungen: Die Garbage Collection gibt automatisch den von nicht mehr benötigten Objekten belegten Speicher frei, die Ausnahmebehandlung erlaubt einen strukturierten und erweiterbaren Ansatz bei der Fehlererkennung und ‑behandlung, und der typsichere Aufbau der Sprache verhindert das Lesen nicht initialisierter Variablen, von Indizes außerhalb der Arraygrenzen und von ungeprüften Typumwandlungen.

C# besitzt ein vereinheitlichtes Typensystem. Alle C#-Typen, einschließlich primitiver Typen wie int und double, werden von einem einzigen Stammtyp vererbt, dem Typ object. So verfügen alle Typen über eine Reihe gemeinsamer Operationen, und die Werte aller Typen können auf konsistente Weise gespeichert, übertragen und bearbeitet werden. Darüber hinaus unterstützt C# benutzerdefinierte Verweistypen und Werttypen. Dies ermöglicht es, Objekte dynamisch zu reservieren oder einfache Strukturen inline zu speichern.

Bei der Entwicklung von C# wurde der Versionsverwaltung große Aufmerksamkeit geschenkt, um eine kompatible Weiterentwicklung von C#-Programmen und ‑Bibliotheken sicherzustellen. In vielen Programmiersprachen wird dieser Aspekt eher vernachlässigt. So kommt es, dass in diesen Sprachen verfasste Programme oft nicht mehr ordnungsgemäß funktionieren, sobald neue Versionen abhängiger Bibliotheken veröffentlicht werden. Zu den Features von C#, die direkt im Hinblick auf die Versionsverwaltung entwickelt wurden, gehören die separaten Modifizierer virtual und override, die Regeln für die Auflösung von Methodenüberladungen und die Unterstützung einer expliziten Deklaration von Schnittstellenmembern.

Im verbleibenden Teil dieses Kapitels werden die wesentlichen Funktionen der C#-Sprache beschrieben. Während in späteren Kapiteln die Regeln und Ausnahmen in einer detaillierten, manchmal mathematischen Weise beschrieben werden, wurde in diesem Kapitel mehr auf Verständlichkeit und Kürze als auf Vollständigkeit geachtet. Das Ziel ist, dem Leser eine Einführung in die Sprache zur Verfügung zu stellen, die ihm das Schreiben einfacher Programme sowie das Verstehen der späteren Kapitel ermöglicht.

## Hello, World

Üblicherweise beginnt jede Einführung in eine Programmiersprache mit dem Programm „Hello, World“. In C# sieht dieses wie folgt aus:

using System;

class Hello  
{  
 static void Main() {  
 Console.WriteLine("Hello, World");  
 }  
}

C#-Quelldateien besitzen i. d. R. die Dateinamenerweiterung .cs. Wenn das Programm "Hello, World" in der Datei hello.cs gespeichert ist, kann das Programm mit dem Microsoft C#-Compiler unter Verwendung der Befehlszeile

csc hello.cs

kompiliert werden. Hierbei wird eine ausführbare Assembly mit dem Namen hello.exe erstellt. Wenn Sie diese Anwendung ausführen, wird Folgendes ausgegeben:

Hello, World

Das Programm "Hello, World" beginnt mit einer using-Direktive, die auf den System-Namespace verweist. Namespaces erlauben die hierarchische Gruppierung von C#-Programmen und ‑Bibliotheken. Namespaces enthalten Typen und weitere Namespaces. Beispielsweise enthält der System-Namespace eine Reihe von Typen wie die Console-Klasse, auf die im Programm verwiesen wird, und weitere Namespaces wie IO und Collections. Mithilfe einer using-Direktive, die auf einen bestimmten Namespace verweist, können Typen aus diesem Namespace ohne ihren vollständig qualifizierten Namen verwendet werden. Aufgrund der using-Direktive kann daher im Programm die Kurzschreibweise Console.WriteLine anstelle von System.Console.WriteLine verwendet werden.

Die im Programm "Hello, World" deklarierte Hello-Klasse besitzt als einzigen Member die Methode mit dem Namen Main. Die Main-Methode wurde mit dem Modifizierer static deklariert. Während Instanzenmethoden mithilfe des Schlüsselworts this auf eine bestimmte einschließende Objektinstanz verweisen, arbeiten statische Methoden ohne Verweise auf ein bestimmtes Objekt. Üblicherweise bildet eine statische Methode mit dem Namen Main den Einstiegspunkt eines Programms.

Die Ausgabe des Programms wird von der WriteLine-Methode der Console-Klasse im System-Namespace generiert. Diese Klasse ist Bestandteil der .NET Framework-Klassenbibliotheken, auf die der Microsoft C#-Compiler standardmäßig automatisch verweist. Beachten Sie, dass C# selbst keine eigene Laufzeitbibliothek besitzt. .NET Framework ist bereits die Laufzeitbibliothek für C#.

## Programmstruktur

Die wichtigsten Kategorien in C# sind Programme, Namespaces, Typen, Member und Assemblys. C#-Programme bestehen aus einer oder mehreren Quelldateien. In Programmen werden Typen deklariert, die Member enthalten und in Namespaces gruppiert werden können. Beispiele solcher Typen sind Klassen und Schnittstellen. Beispiele für Member sind hingegen Felder, Methoden, Eigenschaften und Ereignisse. Bei der Kompilierung eines C#-Programms werden diese physikalisch zu Assemblys zusammengefasst. Assemblys verfügen üblicherweise über die Dateinamenerweiterung .exe oder .dll, je nachdem, ob es sich um die Implementierung einer Anwendung oder einer einer Bibliothek handelt.

In dem Beispiel

using System;

namespace Acme.Collections  
{  
 public class Stack  
 {  
 Entry top;

public void Push(object data) {  
 top = new Entry(top, data);  
 }

public object Pop() {  
 if (top == null) throw new InvalidOperationException();  
 object result = top.data;  
 top = top.next;  
 return result;  
 }

class Entry  
 {  
 public Entry next;  
 public object data;

public Entry(Entry next, object data) {  
 this.next = next;  
 this.data = data;  
 }  
 }  
 }  
}

deklariert eine Klasse mit dem Namen Stack in einem Namespace mit dem Namen Acme.Collections. ist der vollqualifizierte Name dieser Klasse Acme.Collections.Stack. Die Klasse enthält mehrere Member: ein Feld mit dem Namen top, zwei Methoden mit den Namen Push und Pop sowie eine geschachtelte Klasse mit dem Namen Entry. Die Entry-Klasse enthält wiederum drei Member: ein Feld mit dem Namen next, ein Feld mit dem Namen data sowie einen Konstruktor. Wenn der Quellcode des Beispiels in der Datei acme.cs gespeichert wird, wird das Beispiel mit der Befehlszeile

csc /t:library acme.cs

als Bibliothek (Code ohne Einstiegspunkt Main) kompiliert, und es wird eine Assembly mit dem Namen acme.dll generiert.

Assemblys enthalten ausführbaren Code in Form von Anweisungen in Intermediate Language (IL) und symbolische Informationen in Form von Metadaten. Vor der Ausführung wird der IL-Code in einer Assembly vom Just-In-Time (JIT)-Compiler der .NET Common Language Runtime automatisch in prozessorspezifischen Code konvertiert.

Da eine Assembly eine selbstbeschreibende Funktionseinheit darstellt, die sowohl Code als auch Metadaten enthält, sind in C# keine #include-Direktiven oder Headerdateien erforderlich. Die in einer bestimmten Assembly enthaltenen öffentlichen Typen und Member stehen einem C#-Programm durch einfaches Verweisen auf diese Assembly beim Kompilieren des Programms zur Verfügung. Das folgende Programm verwendet als Beispiel die Acme.Collections.Stack-Klasse aus der acme.dll-Assembly:

using System;  
using Acme.Collections;

class Test  
{  
 static void Main() {  
 Stack s = new Stack();  
 s.Push(1);  
 s.Push(10);  
 s.Push(100);  
 Console.WriteLine(s.Pop());  
 Console.WriteLine(s.Pop());  
 Console.WriteLine(s.Pop());  
 }  
}

Wenn das Programm in der Datei test.cs gespeichert ist, kann beim Kompilieren dieser Datei ein Verweis auf die acme.dll-Assembly mithilfe der /r-Compileroption angegeben werden:

csc /r:acme.dll test.cs

Hierdurch wird eine ausführbare Assembly mit dem Namen test.exe erstellt, die bei der Ausführung die folgende Ausgabe generiert:

100  
10  
1

In C# kann der Quelltext eines Programms in mehreren Quelldateien gespeichert werden. Wenn ein C#-Programm mit mehreren Quelldateien kompiliert wird, werden alle Quelldateien gemeinsam verarbeitet, und die Quelldateien können uneingeschränkt aufeinander verweisen – im Prinzip so, als würden alle Quelldateien vor der Verarbeitung zu einer großen Datei zusammengefasst. Vorwärtsdeklaration sind in C# zu keiner Zeit erforderlich, da die Deklarationsreihenfolge mit wenigen Ausnahmen ohne Bedeutung ist. Unter C# sind Quelldateien nicht auf die Deklaration eines einzigen öffentlichen Typs beschränkt. Der Name der Quelldatei muss auch nicht mit dem in dieser Datei deklarierten Typ übereinstimmen.

## Typen und Variablen

In C# gibt es zwei Arten von Typen: Werttypen und Verweistypen. Variablen von Werttypen enthalten direkt die entsprechenden Daten, wohingegen Variablen von Verweistypen Verweise auf diese Daten enthalten, die in diesem Fall als Objekte bezeichnet werden. Bei Verweistypen können zwei Variablen auf dasselbe Objekt verweisen. Somit ist es möglich, dass die eine Variable betreffenden Operationen auch Auswirkungen auf das von der anderen Variablen referenzierte Objekt haben. Bei Werttypen verfügt jede Variable über eine eigene Kopie der Daten, sodass sich Operationen immer nur auf eine Kopie auswirken (wobei ref- und out-Parametervariablen eine Ausnahme darstellen).

Werttypen in C# lassen sich darüber hinaus in einfache Typen, Enumerationstypen, Strukturtypen und Typen, die NULL-Werte zulassen, unterteilen und Verweistypen in Klassentypen, Schnittstellentypen, Arraytypen und Delegattypen.

Die folgende Tabelle enthält eine Übersicht über das C#-Typensystem.

|  |  |  |
| --- | --- | --- |
| **Kategorie** | | **Beschreibung** |
| Werttypen | Einfache Typen | Ganze Zahl mit Vorzeichen: sbyte, short, int, long |
| Ganze Zahl ohne Vorzeichen: byte, ushort, uint, ulong |
| Unicodezeichen: char |
| IEEE-Gleitkomma: float, double |
| Dezimal mit hoher Genauigkeit: decimal |
| Boolesch: bool |
| Aufzählungstypen | Benutzerdefinierte Typen der Form enum E {...} |
| Strukturtypen | Benutzerdefinierte Typen der Form struct S {...} |
| Typen, die NULL-Werte zulassen | Erweiterungen aller anderen Werttypen mit einem null-Wert |
| Verweistypen | Klassentypen | Oberste Basisklasse aller anderen Typen: object |
| Unicode-Zeichenfolgen: string |
| Benutzerdefinierte Typen der Form class C {...} |
| Schnittstellentypen | Benutzerdefinierte Typen der Form interface I {...} |
| Arraytypen | Ein- und mehrdimensional, z. B. int[] und int[,] |
| Delegattypen | Benutzerdefinierte Typen z. B. der Form delegate int D(...) |

Die acht Ganzzahltypen bieten Unterstützung für 8-Bit-, 16-Bit-, 32-Bit- und 64-Bit-Werte mit oder ohne Vorzeichen.

Die beiden Gleitkommatypen float und double werden in Formaten nach IEEE 754 mit 32 Bit (einfacher Genauigkeit) oder 64 Bit (doppelter Genauigkeit) dargestellt.

Der decimal-Typ ist ein 128-Bit-Datentyp, der sich insbesondere für Finanz- und Währungskalkulationen eignet.

Der C#-Typ bool wird zum Darstellen boolescher Werte verwendet. Das sind Werte, die entweder wahr (true) oder falsch (false) sind.

Für die Zeichen- und Zeichenfolgenverarbeitung wird in C# die Unicode-Codierung verwendet. Der char-Typ stellt eine UTF-16-Codeeinheit und der string-Typ eine Folge von UTF-16-Codeeinheiten dar.

In der folgenden Tabelle sind die numerischen C#-Typen zusammengefasst.

|  |  |  |  |
| --- | --- | --- | --- |
| **Kategorie** | **Bits** | **Typ** | **Bereich/Genauigkeit** |
| Ganze Zahl mit Vorzeichen | 8 | sbyte | –128...127 |
| 16 | short | –32,768...32,767 |
| 32 | int | –2,147,483,648...2,147,483,647 |
| 64 | long | –9,223,372,036,854,775,808...9,223,372,036,854,775,807 |
| Ganze Zahl ohne Vorzeichen | 8 | byte | 0...255 |
| 16 | ushort | 0...65,535 |
| 32 | uint | 0...4,294,967,295 |
| 64 | ulong | 0...18,446,744,073,709,551,615 |
| Gleitkomma | 32 | float | 1,5 × 10−45 bis 3,4 × 1038, Genauigkeit von 7 Ziffern |
| 64 | double | 5,0 × 10−324 bis 1,7 × 10308, Genauigkeit von 15 Ziffern |
| Dezimal | 128 | decimal | 1,0 × 10−28 bis 7,9 × 1028, Genauigkeit von 28 Ziffern |

Zum Erstellen neuer Typen werden in C#-Programmen Typdeklarationen verwendet. In einer Typdeklaration werden der Name und die Member des neuen Typs angegeben. Unter den C#-Typkategorien sind fünf benutzerdefinierbare Typen: Klassentypen, Strukturtypen, Schnittstellentypen, Enumerationstypen und Delegattypen.

Ein Klassentyp definiert eine Datenstruktur, die Datenmember (Felder) und Funktionsmember (u. a. Methoden, Eigenschaften) enthält. Klassentypen unterstützen einfache Vererbung und Polymorphie. Diese Mechanismen erlauben es, Basisklassen in abgeleiteten Klassen zu erweitern und zu spezialisieren.

Ein Strukturtyp ähnelt einem Klassentyp insofern, als er eine Struktur mit Datenmembern und Funktionsmembern darstellt. Im Gegensatz zu Klassen handelt es sich bei Strukturen jedoch um Werttypen, die keine Heapzuordnung benötigen. Bei Strukturtypen wird keine benutzerdefinierte Vererbung unterstützt, und alle Strukturtypen werden implizit vom Typ object vererbt.

Ein Schnittstellentyp spezifiziert einen benannten Satz von öffentlichen Funktionsmembern. Eine Klasse oder Struktur, die eine Schnittstelle implementiert, muss Implementierungen der Funktionsmember der Schnittstelle bereitstellen. Eine Schnittstelle kann von mehreren Basisschnittstellen erben. Klassen oder Strukturen können jeweils mehrere Schnittstellen implementieren.

Ein Delegattyp stellt Verweise auf Methoden mit einer bestimmten Parameterliste und einem bestimmten Rückgabetyp dar. Delegaten erlauben es, Methoden als Einheiten zu behandeln, die Variablen zugewiesen und als Parameter übergeben werden können. Somit ähneln Delegaten dem Konzept der Funktionszeiger, wie sie in anderen Programmiersprachen unterstützt werden. Im Gegensatz zu Funktionszeigern sind Delegaten jedoch objektorientiert und typsicher.

Klassentypen, Strukturtypen, Schnittstellentypen und Delegattypen unterstützen Generika und können dabei mit anderen Typen parametrisiert werden.

Ein Enumerationstyp ist ein eigenständiger Typ mit benannten Konstanten. Jedem Enumerationstyp liegt ein Typ zugrunde, der einer der acht Ganzzahltypen sein muss. Die Wertegruppe eines Enumerationstyps ist mit der Wertegruppe des zugrunde liegenden Typs identisch.

C# unterstützt ein- und mehrdimensionale Arrays beliebiger Typen. Arraytypen unterscheiden sich von den anderen oben aufgelisteten Typen darin, dass sie vor ihrer Verwendung nicht deklariert werden müssen. Stattdessen werden Arraytypen erstellt, indem einem Typnamen eckige Klammern nachgestellt werden. Beispielsweise stellt int[] ein eindimensionales int-Array dar, int[,] ein zweidimensionales int-Array und int[][] ein eindimensionales Array von eindimensionalen int-Arrays.

Typen, die NULL-Werte zulassen, müssen vor ihrer Verwendung ebenfalls nicht deklariert werden. Es ist zu jedem Werttyp T, der keine NULL-Werte zulässt, ein zugehöriger Typ TT? vorhanden, der NULL-Werte zulässt und einen zusätzlichen null-Wert enthalten kann. int? ist z. B. ein Typ, der beliebige 32-Bit-Ganzzahlen oder den Wert null enthalten kann.

Das Typensystem von C# ist dahingehend vereinheitlicht, dass ein Wert beliebigen Typs als ein Objekt behandelt werden kann. Alle Typen in C# sind direkt oder indirekt vom object-Klassentyp abgeleitet, und object ist die oberste Basisklasse für alle Typen. Werte von Verweistypen werden als Objekte behandelt, indem die Werte einfach als object-Typen betrachtet werden. Die Werte der Werttypen werden durch die Ausführung von Boxing- und Unboxing-Operationen als Objekte behandelt. Im folgenden Beispiel wird ein int-Wert in object und zurück in int konvertiert.

using System;

class Test  
{  
 static void Main() {  
 int i = 123;  
 object o = i; // Boxing  
 int j = (int)o; // Unboxing  
 }  
}

Wenn der Wert eines Werttyps in den Typ object konvertiert wird, wird eine Objektinstanz für die Aufnahme des Werts ("Box") zugewiesen und der Wert in diese Box kopiert ("geschachtelt"). Wenn hingegen ein Verweis auf object in einen Werttyp umgewandelt wird, wird überprüft, ob das referenzierte Objekt eine Box mit dem korrekten Werttyp ist, und der Wert wird bei erfolgreicher Überprüfung aus der Box kopiert.

So besagt das vereinheitlichte Typensystem von C# im Grunde, dass Werttypen ganz nach Bedarf zu Objekten werden können. Aufgrund der Vereinheitlichung können Universalbibliotheken, die den Typ object verwenden, sowohl mit Verweistypen als auch mit Werttypen verwendet werden.

In C# gibt es mehrere Arten von Variablen, z. B. Felder, Arrayelemente, lokale Variablen und Parameter. Variablen stellen Speicherorte dar, und jede Variable besitzt einen Typ, von dem entsprechend der Übersicht in der folgenden Tabelle abhängt, welche Werte in der Variablen gespeichert werden können.

|  |  |
| --- | --- |
| **Variablentyp** | **Zulässiger Inhalt** |
| Werttyp, der keine NULL-Werte zulässt | Ein Wert von genau diesem Typ |
| Werttyp, der NULL-Werte zulässt | Ein NULL-Wert oder ein Wert von genau diesem Typ |
| object | Ein NULL-Verweis, ein Verweis auf ein Objekt eines beliebigen Verweistyps oder ein Verweis auf den geschachtelten Wert eines beliebigen Werttyps |
| Klassentyp | Ein NULL-Verweis, ein Verweis auf eine Instanz dieses Klassentyps oder ein Verweis auf die Instanz einer von diesem Klassentyp abgeleiteten Klasse |
| Schnittstellentyp | Ein NULL-Verweis, ein Verweis auf die Instanz eines Klassentyps, der diesen Schnittstellentyp implementiert, oder ein Verweis auf den geschachtelten Wert eines Werttyps, der diesen Schnittstellentyp implementiert |
| Arraytyp | Ein NULL-Verweis, ein Verweis auf eine Instanz dieses Arraytyps oder ein Verweis auf die Instanz eines kompatiblen Arraytyps |
| Delegattyp | Ein NULL-Verweis oder ein Verweis auf eine Instanz dieses Delegattypen |

## Ausdrücke

Ausdrücke werden aus Operanden und Operatoren gebildet. Die Operatoren eines Ausdrucks geben an, welche Operationen auf die Operanden angewendet werden. Beispiele für Operatoren sind +, -, \*, / und new. Beispiele für Operanden sind Literale, Felder, lokale Variablen und Ausdrücke.

Wenn ein Ausdruck mehrere Operatoren enthält, wird anhand des Vorrangs der Operatoren festgelegt, in welcher Reihenfolge die einzelnen Operatoren ausgewertet werden. Der Ausdruck x + y \* z wird beispielsweise als x + (y \* z) ausgewertet, da der \*-Operator vor dem +-Operator Vorrang hat.

Bestimmte Operatoren können überladen werden. Durch das Überladen von Operatoren können benutzerdefinierte Operatoren für Operationen implementiert werden, wobei einer oder beide der Operanden einem benutzerdefinierten Klassen- oder Strukturtypen entsprechen.

Die folgende Tabelle enthält eine Zusammenfassung der C#-Operatoren in der Reihenfolge der Operatorkategorie mit dem höchsten Vorrang zu der mit dem niedrigsten Vorrang: Operatoren der gleichen Kategorie besitzen den gleichen Vorrang.

|  |  |  |
| --- | --- | --- |
| **Kategorie** | **Ausdruck** | **Beschreibung** |
| Primär | x.m | Memberzugriff |
| x(...) | Methoden- und Delegataufruf |
| x[...] | Array- und Indexerzugriff |
| x++ | Postimkrementierung |
| x-- | Postdekrementierung |
| new T(...) | Objekt- und Delegaterstellung |
| new T(...){...} | Objekterstellung mit Initialisierer |
| new {...} | Anonymer Objektinitialisierer |
| new T[...] | Arrayerstellung |
| typeof(T) | Abrufen des System.Type-Objekts für T |
| checked(x) | Auswerten des Ausdrucks im geprüften Kontext |
| unchecked(x) | Auswerten des Ausdrucks im ungeprüften Kontext |
| default(T) | Abrufen des Standardwerts vom Typ T |
| delegate {...} | Anonyme Funktion (anonyme Methode) |
| Unär | +x | Identität |
| -x | Negation |
| !x | Logische Negation |
| ~x | Bitweise Negation |
| ++x | Präinkrementierung |
| --x | Prädekrementierung |
| (T)x | Explizite Umwandlung von x in Typ T |
| await x | Asynchrones Warten auf den Abschluss von x |
| Multiplikativ | x \* y | Multiplikation |
| x / y | Division |
| x % y | Rest |

|  |  |  |
| --- | --- | --- |
| Additiv | x + y | Addition, Zeichenfolgenverkettung, Delegatkombination |
| x – y | Subtraktion, Delegatentfernung |
| Verschiebung | x << y | Verschiebung nach links |
| x >> y | Verschiebung nach rechts |
| Relational und Typtest | x < y | Kleiner als |
| x > y | Größer als |
| x <= y | Kleiner oder gleich |
| x >= y | Größer oder gleich |
| x is T | Gibt true zurück, wenn x ein T ist, andernfalls false. |
| x as T | Gibt x als Typ T zurück oder null, wenn x nicht vom Typ T ist |
| Gleichheit | x == y | Gleich |
| x != y | Ungleich |
| Logisches AND | x & y | Ganzzahliges bitweises AND, boolesches logisches AND |
| Logisches XOR | x ^ y | Ganzzahliges bitweises XOR, boolesches logisches XOR |
| Logisches OR | x | y | Ganzzahliges bitweises OR, boolesches logisches OR |
| Bedingtes AND | x && y | Wertet y nur aus, wenn xtrue ist |
| Bedingtes OR | x || y | Wertet y nur aus, wenn xfalse ist |
| Null Coalesce | X ?? y | Ergibt y, wenn x den Wert null hat, andernfalls x |
| Bedingung | x ? y : z | Wertet y aus, wenn x den Wert true hat, und z, wenn x den Wert false hat |
| Zuweisung oder anonyme Funktion | x = y | Zuweisung |
| x op= y | Verbundzuweisung mit den unterstützten Operatoren  \*= /= %= += -= <<= >>= &= ^= |= |
| (T x) => y | Anonyme Funktion (Lambda-Ausdruck) |

## Anweisungen

Die Aktionen eines Programms werden mithilfe von Anweisungen ausgedrückt. In C# werden verschiedene Arten von Anweisungen unterstützt, von denen eine Reihe als eingebettete Anweisungen definiert sind.

Ein Block ermöglicht das Schreiben von mehreren Anweisungen in Kontexten, in denen nur eine Anweisung zulässig ist. Ein Block besteht aus einer Liste von Anweisungen, die zwischen den Trennzeichen { und } stehen.

Deklarationsanweisungen werden für die Deklaration lokaler Variablen und Konstanten verwendet.

Mit Ausdrucksanweisungen werden Ausdrücke ausgewertet. Zu den Ausdrücken, die als Anweisungen verwendet werden können, zählen Methodenaufrufe, Objektzuordnungen mithilfe des new-Operators, Zuweisungen mit = und den Verbundzuweisungsoperatoren sowie Inkrement- und Dekrementoperationen mithilfe der Operatoren ++ und -- sowie von Await-Ausdrücken.

In Auswahlanweisungen wird entsprechend dem Wert eines Ausdrucks aus einer Reihe möglicher Anweisungen eine ausgewählt, die ausgeführt werden soll. Zu dieser Gruppe gehören die Anweisungen if und switch.

Bei Iterationsanweisungen wird eine eingebettete Anweisung wiederholt ausgeführt. Zu dieser Gruppe gehören die Anweisungen while, do, for und foreach.

Sprunganweisungen werden verwendet, um die Steuerung an andere Anweisungen zu übergeben. Zu dieser Gruppe gehören die Anweisungen break, continue, goto, throw, return und yield.

Mit der try...catch-Anweisung werden Ausnahmen abgefangen, die während der Ausführung eines Blocks auftreten. Die try...finally-Anweisung wird verwendet, um Abschlusscode anzugeben, der in jedem Fall, auch bei Auftreten einer Ausnahme, ausgeführt wird.

Die checked-Anweisung und die unchecked-Anweisung werden verwendet, um den Kontext der Überlaufprüfung für arithmetische Ganzzahloperationen und Konvertierungen zu steuern.

Mit der lock-Anweisung wird die Sperre für gegenseitigen Ausschluss eines bestimmten Objekts abgerufen, eine Anweisung ausgeführt und die Sperre anschließend aufgehoben.

Mit der using-Anweisung wird eine Ressource abgerufen, eine Anweisung ausgeführt und diese Ressource dann verworfen.

In der folgenden Tabelle werden die C#-Anweisungen jeweils mit entsprechenden Beispielen aufgeführt.

|  |  |
| --- | --- |
| **Anweisung** | **Beispiel** |
| Deklaration von lokalen Variablen | static void Main() {  int a;   int b = 2, c = 3;   a = 1;  Console.WriteLine(a + b + c); } |
| Deklaration von lokalen Konstanten | static void Main() {  const float pi = 3.1415927f;  const int r = 25;  Console.WriteLine(pi \* r \* r); } |
| Ausdrucksanweisung | static void Main() {  int i;  i = 123; // Expression statement  Console.WriteLine(i); // Expression statement  i++; // Expression statement  Console.WriteLine(i); // Expression statement } |
| if-Anweisung | static void Main(string[] args) {  if (args.Length == 0) {  Console.WriteLine("No arguments");  }  else {  Console.WriteLine("One or more arguments");  } } |

|  |  |
| --- | --- |
| switch-Anweisung | static void Main(string[] args) {  int n = args.Length;  switch (n) {  case 0:  Console.WriteLine("No arguments");  break;  case 1:  Console.WriteLine("One argument");  break;  default:  Console.WriteLine("{0} arguments", n);  break;  }  } } |
| while-Anweisung | static void Main(string[] args) {  int i = 0;  while (i < args.Length) {  Console.WriteLine(args[i]);  i++;  } } |
| do-Anweisung | static void Main() {  string s;  do {  s = Console.ReadLine();  if (s != null) Console.WriteLine(s);  } while (s != null); } |
| for-Anweisung | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  Console.WriteLine(args[i]);  } } |
| foreach-Anweisung | static void Main(string[] args) {  foreach (string s in args) {  Console.WriteLine(s);  } } |
| break-Anweisung | static void Main() {  while (true) {  string s = Console.ReadLine();  if (s == null) break;  Console.WriteLine(s);  } } |
| continue-Anweisung | static void Main(string[] args) {  for (int i = 0; i < args.Length; i++) {  if (args[i].StartsWith("/")) continue;  Console.WriteLine(args[i]);  } } |

|  |  |
| --- | --- |
| goto-Anweisung | static void Main(string[] args) {  int i = 0;  goto check;  loop:  Console.WriteLine(args[i++]);  check:  if (i < args.Length) goto loop; } |
| return-Anweisung | static int Add(int a, int b) {  return a + b; }  static void Main() {  Console.WriteLine(Add(1, 2));  return; } |
| yield-Anweisung | static IEnumerable<int> Range(int from, int to) {  for (int i = from; i < to; i++) {  yield return i;  }  yield break; }  static void Main() {  foreach (int x in Range(-10,10)) {  Console.WriteLine(x);  } } |
| Anweisungen throw und try | static double Divide(double x, double y) {  if (y == 0) throw new DivideByZeroException();  return x / y; }  static void Main(string[] args) {  try {  if (args.Length != 2) {  throw new Exception("Two numbers required");  }  double x = double.Parse(args[0]);  double y = double.Parse(args[1]);  Console.WriteLine(Divide(x, y));  }  catch (Exception e) {  Console.WriteLine(e.Message);  }  finally {  Console.WriteLine(“Good bye!”);  } } |
| Anweisungen checked und unchecked | static void Main() {  int i = int.MaxValue;  checked {  Console.WriteLine(i + 1); // Exception  }  unchecked {  Console.WriteLine(i + 1); // Overflow  } } |

|  |  |
| --- | --- |
| lock-Anweisung | class Account {  decimal balance;  public void Withdraw(decimal amount) {  lock (this) {  if (amount > balance) {  throw new Exception("Insufficient funds");  }  balance -= amount;  }  } } |
| using-Anweisung | static void Main() {  using (TextWriter w = File.CreateText("test.txt")) {  w.WriteLine("Line one");  w.WriteLine("Line two");  w.WriteLine("Line three");  } } |

## Klassen und Objekte

Klassen stellen den grundlegenden C#-Typ dar. Eine Klasse ist eine Datenstruktur, in der Zustände (Felder) und Aktionen (Methoden und andere Funktionsmember) zu einer Einheit zusammengefasst werden. Eine Klasse stellt eine Definition für dynamisch erstellte Instanzen der Klasse bereit, die auch als Objekte bezeichnet werden. Klassen unterstützen Vererbung und Polymorphie. Diese Mechanismen erlauben es, Basisklassen in abgeleiteten Klassen zu erweitern und zu spezialisieren.

Neue Klassen werden mithilfe von Klassendeklaration erstellt. Eine Klassendeklaration beginnt mit einem Kopf, in dem die Attribute und Modifizierer der Klasse, der Name der Klasse, die Basisklasse (sofern angegeben) und die von dieser Klasse implementierten Schnittstellen angegeben werden. Auf den Kopf folgt der Klassentext, der aus einer Liste von Memberdeklarationen zwischen den Trennzeichen { und } besteht.

Hier sehen Sie die Deklaration einer einfachen Klasse mit dem Namen Point:

public class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Instanzen von Klassen werden mithilfe des new-Operators erstellt, durch den Speicher für eine neue Instanz belegt, ein Konstruktor zum Initialisieren der Instanz aufgerufen und ein Verweis auf die Instanz zurückgegeben wird. Mit den folgenden beiden Anweisungen werden zwei Point-Objekte erstellt und Verweise auf diese Objekte in zwei Variablen gespeichert:

Point p1 = new Point(0, 0);  
Point p2 = new Point(10, 20);

Der von einem Objekt belegte Speicherplatz wird automatisch freigegeben, wenn das Objekt nicht mehr verwendet wird. In C# ist es nicht erforderlich und auch gar nicht möglich, die Objektzuordnung explizit aufzuheben.

### Member

Die Member einer Klasse sind entweder statische Member oder Instanzmember. Statische Member gehören zu Klassen, Instanzmember gehören zu Objekten (Klasseninstanzen).

Die folgende Tabelle enthält eine Übersicht der Arten von Membern, über die eine Klasse verfügen kann.

|  |  |
| --- | --- |
| **Member** | **Beschreibung** |
| Konstanten | Die dieser Klasse zugeordneten Konstantenwerte |
| Felder | Variablen der Klasse |
| Methoden | Berechnungen und Aktionen, die eine Klasse ausführen kann |
| Eigenschaften | Aktionen im Zusammenhang mit dem Lesen und Schreiben benannter Eigenschaften der Klasse |
| Indexer | Aktionen im Zusammenhang mit dem Indizieren von Instanzen der Klasse wie bei einem Array |
| Ereignisse | Benachrichtigungen, die von der Klasse generiert werden können |
| Operatoren | Von der Klasse unterstützte Konvertierungen und Ausdrucksoperatoren |
| Konstruktoren | Zum Initialisieren der Klasse selbst oder von Instanzen der Klasse erforderliche Aktionen |
| Destruktoren | Vor dem endgültigen Verwerfen von Instanzen der Klasse auszuführende Aktionen |
| Typen | In der Klasse deklarierte geschachtelte Typen |

### Barrierefreiheit

Jedem Member einer Klasse ist eine Form des Zugriffs zugeordnet, die bestimmt, von welchen Bereichen des Programmtexts aus auf den Member zugegriffen werden kann. Es gibt fünf mögliche Formen des Zugriffs. Diese sind in der folgenden Tabelle zusammengefasst.

|  |  |
| --- | --- |
| **Barrierefreiheit** | **Bedeutung** |
| public | Unbeschränkter Zugriff |
| protected | Zugriff ist auf diese Klasse oder von dieser Klasse abgeleitete Klassen beschränkt |
| internal | Zugriff ist auf dieses Programm beschränkt |
| protected internal | Zugriff ist auf dieses Programm oder von dieser Klasse abgeleitete Klassen beschränkt |
| private | Zugriff ist auf diese Klasse beschränkt |

### Typparameter

In einer Klassendefinition kann ein Satz von Typparametern angegeben werden, indem dem Klassennamen eine Liste von Typparameternamen in spitzen Klammern nachgestellt wird. Die Typparameter können im Text der Klassendeklarationen verwendet werden, um die Klassenmember zu definieren. Im folgenden Beispiel haben die Typparameter von Pair die Bezeichnungen TFirst und TSecond:

public class Pair<TFirst,TSecond>  
{  
 public TFirst First;

public TSecond Second;  
}

Ein Klassentyp, der für das Akzeptieren von Typparametern deklariert wurde, wird als generischer Klassentyp bezeichnet. Struktur-, Schnittstellen- und Delegattypen können ebenfalls generisch sein.

Bei der Verwendung der generischen Klasse müssen für die einzelnen Typparameter Typargumente angegeben werden.

Pair<int,string> pair = new Pair<int,string> { First = 1, Second = “two” };  
int i = pair.First; // TFirst is int  
string s = pair.Second; // TSecond is string

Ein generischer Typ, für den Typargumente angegeben werden, wie Pair<int,string> oben, wird als konstruierter Typ bezeichnet.

### Basisklassen

In einer Klassendeklaration kann eine Basisklasse angegeben werden, indem dem Klassennamen und den Typparametern ein Doppelpunkt und der Name der Basisklasse nachgestellt wird. Wenn keine Basisklasse angegeben wird, entspricht dies einem Ableiten vom Typ object. Im folgenden Beispiel ist Point die Basisklasse von Point3D und object die Basisklasse von Point:

public class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

public class Point3D: Point  
{  
 public int z;

public Point3D(int x, int y, int z): base(x, y) {  
 this.z = z;  
 }  
}

Eine Klasse erbt die Member ihrer Basisklasse. Vererbung bedeutet, dass eine Klasse implizit alle Member ihres Basisklassentyps enthält. Davon ausgenommen sind lediglich die Instanzkonstruktoren und die statische Konstruktoren sowie die Destruktoren der Basisklasse. Sie kann neue Member zu den geerbten hinzufügen, jedoch nicht die Definition eines geerbten Members entfernen. Im vorausgehenden Beispiel erbt Point3D das x-Feld und das y-Feld von Point, und jede Point3D-Instanz enthält die drei Felder x, y und z.

Für einen Klassentypen ist eine implizite Konvertierung in alle seiner Basisklassentypen vorhanden. Daher kann die Variable eines Klassentyps auf eine Instanz dieser Klasse oder eine Instanz einer beliebigen abgeleiteten Klasse verweisen. Ausgehend von den oben stehenden Klassendeklarationen kann eine Variable vom Typ Point beispielsweise auf einen Point ebenso wie auf einen Point3D verweisen:

Point a = new Point(10, 20);  
Point b = new Point3D(10, 20, 30);

### Felder

Felder sind Variablen, die einer Klasse oder der Instanz einer Klasse zugeordnet sind.

Statische Felder werden mit dem static-Modifizierer deklariert. Ein statisches Feld kennzeichnet genau einen Speicherort. Unabhängig von der Anzahl der Instanzen einer Klasse gibt es immer nur eine Kopie eines statischen Feldes.

Ein Feld, das ohne den static-Modifizierer deklariert wird, ist als Instanzfeld definiert. Jede Instanz einer Klasse enthält eine separate Kopie aller Instanzfelder der Klasse.

Im folgenden Beispiel verfügt jede Instanz der Color-Klasse über eine eigene Kopie der Instanzfelder r, g und b. Die statischen Felder Black, White, Red, Green und Blue sind hingegen nur einmal vorhanden:

public class Color  
{  
 public static readonly Color Black = new Color(0, 0, 0);  
 public static readonly Color White = new Color(255, 255, 255);  
 public static readonly Color Red = new Color(255, 0, 0);  
 public static readonly Color Green = new Color(0, 255, 0);  
 public static readonly Color Blue = new Color(0, 0, 255);

private byte r, g, b;

public Color(byte r, byte g, byte b) {  
 this.r = r;  
 this.g = g;  
 this.b = b;  
 }  
}

Wie im vorhergehenden Beispiel veranschaulicht wird, können mithilfe des readonly-Modifizierers schreibgeschützte Felder deklariert werden. Zuweisungen zu einem readonly-Feld können nur in der Deklaration des Felds oder in einem Konstruktor in derselben Klasse vorgenommen werden.

### Methoden

Methoden sind Member, die eine Berechnung oder Aktion implementieren, die von einem Objekt oder einer Klasse ausgeführt werden kann. Auf statische Methoden wird über die Klasse zugegriffen. Auf Instanzmethoden wird über Instanzen der Klasse zugegriffen.

Methoden verfügen über eine (ggf. leere) Liste von Parametern, die Werte oder Variablenverweise darstellen, die an die Methode übergeben werden, sowie über einen Rückgabetyp, der den Typ des von dieser Methode berechneten und zurückgegebenen Werts angibt. Wenn eine Methode keinen Wert zurückgibt, ist ihr Rückgabetyp void.

Methoden verfügen wie Typen über einen Satz von Typparametern, für die beim Aufruf der Methode Typargumente angegeben werden müssen. Anders als Typen können die Typargumente häufig aus den Argumenten eines Methodenaufrufs abgeleitet werden und müssen nicht explizit angegeben werden.

Die Signatur einer Methode muss innerhalb der Klasse, in der die Methode deklariert wird, eindeutig sein. Die Signatur einer Methode besteht aus dem Namen der Methode, der Anzahl der Typparameter sowie der Anzahl, den Modifizierern und den Typen der Parameter. Die Signatur einer Methode beinhaltet nicht den Rückgabetyp.

#### Parameter

Mit Parametern können Werte und Variablenverweise an Methoden übergeben werden. Die Parameter einer Methode erhalten ihre eigentlichen Werte über die Argumente, die beim Aufruf der Methode angegeben werden. Es gibt vier Arten von Parametern: Werteparameter, Verweisparameter, Ausgabeparameter und Parameterarrays.

Werteparameter werden zum Übergeben von Eingabeparametern verwendet. Ein Werteparameter entspricht einer lokalen Variablen, die ihren Anfangswert von dem Argument erhält, das für diesen Parameter übergeben wurde. Änderungen am Werteparameter haben keine Auswirkungen auf das für diesen Parameter übergebene Argument.

Wertparameter können optional sein, indem ein Standardwert angegeben wird, sodass entsprechende Argumente ausgelassen werden können.

Verweisparameter werden zum Übergeben von Eingabe- und Ausgabeparametern verwendet. Bei dem für einen Verweisparameter übergebenen Argument muss es sich um eine Variable handeln, und während der Ausführung der Methode beziehen sich Verweisparameter und Argumentvariable auf denselben Speicherort. Verweisparameter werden mit dem ref-Modifizierer deklariert. Im folgenden Beispiel wird die Verwendung von ref-Parametern veranschaulicht:

using System;

class Test  
{  
 static void Swap(ref int x, ref int y) {  
 int temp = x;  
 x = y;  
 y = temp;  
 }

static void Main() {  
 int i = 1, j = 2;  
 Swap(ref i, ref j);  
 Console.WriteLine("{0} {1}", i, j); // Outputs "2 1"  
 }  
}

Ausgabeparameter werden zum Übergeben von Ausgabeparametern verwendet. Ein Ausgabeparameter ähnelt einem Verweisparameter mit dem Unterschied, dass der Anfangswert des vom Aufrufer angegebenen Arguments irrelevant ist. Ausgabeparameter werden mit dem out-Modifizierer deklariert. Im folgenden Beispiel wird die Verwendung von out-Parametern veranschaulicht:

using System;

class Test  
{  
 static void Divide(int x, int y, out int result, out int remainder) {  
 result = x / y;  
 remainder = x % y;  
 }

static void Main() {  
 int res, rem;  
 Divide(10, 3, out res, out rem);  
 Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"  
 }  
}

Parameterarrays erlauben es, einer Methode eine variable Anzahl von Argumenten zu übergeben. Parameterarrays werden mit dem params-Modifizierer deklariert. Nur der letzte Parameter einer Methode kann ein Parameterarray sein, und der Typ eines Parameterarrays muss ein eindimensionaler Arraytyp sein. Die Write-Methode und die WriteLine-Methode der System.Console-Klasse sind gute Beispiele für die Verwendung von Parameterarrays. Diese sind wie folgt deklariert:

public class Console  
{  
 public static void Write(string fmt, params object[] args) {...}

public static void WriteLine(string fmt, params object[] args) {...}

...  
}

Innerhalb einer Methode, die ein Parameterarray verwendet, verhält sich das Parameterarray genau wie ein normaler Parameter vom Typ eines Arrays. Im Aufruf einer Methode mit einem Parameterarray ist es hingegen möglich, entweder ein einzelnes Argument vom Typ des Parameterarrays oder eine beliebige Anzahl von Argumenten vom Typ der Elemente des Parameterarrays zu übergeben. Bei Letzterem wird automatisch eine Arrayinstanz erstellt und mit den angegebenen Argumenten initialisiert. Dieses Beispiel

Console.WriteLine("x={0} y={1} z={2}", x, y, z);

entspricht dem folgenden Code:

string s = "x={0} y={1} z={2}";  
object[] args = new object[3];  
args[0] = x;  
args[1] = y;  
args[2] = z;  
Console.WriteLine(s, args);

#### Methodentext und lokale Variablen

Im Methodentext werden die Anweisungen angegeben, die beim Aufruf der Methode ausgeführt werden sollen.

Im Methodentext können Variablen deklariert werden, die nur beim Aufrufen dieser Methode gelten sollen. Solche Variablen werden als lokale Variablen bezeichnet. In der Deklaration einer lokalen Variablen werden ein Typname, ein Variablenname und ggf. ein Anfangswert angegeben. Im folgenden Beispiel werden die lokale Variable i mit dem Anfangswert 0 (null) und die lokale Variable j ohne Anfangswert deklariert.

using System;

class Squares  
{  
 static void Main() {  
 int i = 0;  
 int j;  
 while (i < 10) {  
 j = i \* i;  
 Console.WriteLine("{0} x {0} = {1}", i, j);  
 i = i + 1;  
 }  
 }  
}

In C# müssen lokale Variablen definitiv zugewiesen werden, bevor ihr Wert abgerufen werden kann. Wenn beispielsweise bei der vorgenannten Deklaration von i kein Anfangswert angegeben wird, gibt der Compiler bei späteren Verwendungen von i einen Fehler aus, weil es an diesen Positionen im Programm noch keine definitive Zuweisung für i gegeben hat.

Mithilfe von return-Anweisungen können Methoden die Steuerung an den Aufrufer zurückgeben. In Methoden, die void zurückgeben, kann für return-Anweisungen kein Ausdruck angegeben werden. In Methoden, die nicht void zurückgeben, müssen return-Anweisungen einen Ausdruck enthalten, mit dem der Rückgabewert berechnet wird.

#### Statische Methoden und Instanzmethoden

Mit dem static-Modifizierer deklarierte Methoden sind statische Methoden. Statische Methoden werden nicht für eine bestimmte Instanz ausgeführt und können nur direkt auf statische Member zugreifen.

Ohne den static-Modifizierer deklarierte Methoden sind Instanzmethoden. Instanzmethoden werden für eine bestimmte Instanz ausgeführt und können auf statische Member und Instanzmember zugreifen. Auf die Instanz, für die eine Instanzmethode aufgerufen wurde, kann explizit als this zugegriffen werden. In einer statischen Methode auf this zu verweisen, führt zu einem Fehler.

Die folgende Entity-Klasse besitzt statische Member und Instanzmember.

class Entity  
{  
 static int nextSerialNo;

int serialNo;

public Entity() {  
 serialNo = nextSerialNo++;  
 }

public int GetSerialNo() {  
 return serialNo;  
 }

public static int GetNextSerialNo() {  
 return nextSerialNo;  
 }

public static void SetNextSerialNo(int value) {  
 nextSerialNo = value;  
 }  
}

Jede Entity-Instanz enthält eine Seriennummer (neben anderen Informationen, die hier nicht aufgeführt werden). Der Entity-Konstruktor (der einer Instanzmethode entspricht) initialisiert die neue Instanz mit der nächsten verfügbaren Seriennummer. Da es sich bei dem Konstruktor um einen Instanzmember handelt, kann in ihm auf das serialNo-Instanzfeld ebenso wie auf das statische nextSerialNo-Feld zugegriffen werden.

Die statische GetNextSerialNo-Methode und die statische SetNextSerialNo-Methode können auf das statische nextSerialNo-Feld zugreifen, ein direkter Zugriff auf das serialNo-Instanzfeld würde jedoch zu einem Fehler führen.

Im folgenden Beispiel wird die Verwendung der Entity-Klasse veranschaulicht.

using System;

class Test  
{  
 static void Main() {  
 Entity.SetNextSerialNo(1000);

Entity e1 = new Entity();  
 Entity e2 = new Entity();

Console.WriteLine(e1.GetSerialNo()); // Outputs "1000"  
 Console.WriteLine(e2.GetSerialNo()); // Outputs "1001"  
 Console.WriteLine(Entity.GetNextSerialNo()); // Outputs "1002"  
 }  
}

Beachten Sie, dass die statische SetNextSerialNo-Methode und die statische GetNextSerialNo-Methode für die Klasse, die GetSerialNo-Instanzmethode hingegen für Instanzen der Klasse aufgerufen wird.

#### Virtuelle und abstrakte Methoden sowie Überschreibungsmethoden

Wenn die Deklaration einer Instanzmethode einen virtual-Modifizierer enthält, wird diese Methode als virtuelle Methode bezeichnet. Wenn kein virtual-Modifizierer vorhanden ist, wird die Methode als nicht virtuelle Methode bezeichnet.

Beim Aufruf einer virtuellen Methode bestimmt der Laufzeittyp der Instanz, für die der Aufruf ausgeführt wird, die tatsächlich aufzurufende Methodenimplementierung. Bei einem Aufruf einer nicht virtuellen Methode ist der Kompilierungstyp der Instanz der entscheidende Faktor.

Virtuelle Methoden können in abgeleiteten Klassen überschrieben werden. Wenn in der Deklaration einer Instanzmethode ein override-Modifizierer enthalten ist, überschreibt die Methode eine vererbte virtuelle Methode mit der gleichen Methode. Während eine virtuelle Methodendeklaration eine neue Methode einführt, spezialisiert eine Überschreibungsmethodendeklaration eine vorhandene vererbte virtuelle Methode, indem eine neue Implementierung dieser Methode zur Verfügung gestellt wird.

Eine abstrakte Methode ist eine virtuelle Methode ohne Implementierung. Abstrakte Methoden werden mit dem abstract-Modifizierer deklariert und sind nur in einer Klasse zulässig, die ebenfalls als abstract deklariert ist. Abstrakte Methoden müssen in allen abgeleiteten Klassen, die nicht abstrakt sind, überschrieben werden.

Im folgenden Beispiel werden die abstrakte Expression-Klasse, die Elemente eines Ausdrucks darstellt, und die drei abgeleiteten Klassen Constant, VariableReference und Operation deklariert, die Ausdruckselemente für Konstanten, Variablenverweise und arithmetische Operationen implementieren. (Dies ähnelt den im Abschnitt §4.6 beschriebenen Ausdrucksbaumstrukturtypen, darf mit diesen aber nicht verwechselt werden).

using System;  
using System.Collections;

public abstract class Expression  
{  
 public abstract double Evaluate(Hashtable vars);  
}

public class Constant: Expression  
{  
 double value;

public Constant(double value) {  
 this.value = value;  
 }

public override double Evaluate(Hashtable vars) {  
 return value;  
 }  
}

public class VariableReference: Expression  
{  
 string name;

public VariableReference(string name) {  
 this.name = name;  
 }

public override double Evaluate(Hashtable vars) {  
 object value = vars[name];  
 if (value == null) {  
 throw new Exception("Unknown variable: " + name);  
 }  
 return Convert.ToDouble(value);  
 }  
}

public class Operation: Expression  
{  
 Expression left;  
 char op;  
 Expression right;

public Operation(Expression left, char op, Expression right) {  
 this.left = left;  
 this.op = op;  
 this.right = right;  
 }

public override double Evaluate(Hashtable vars) {  
 double x = left.Evaluate(vars);  
 double y = right.Evaluate(vars);  
 switch (op) {  
 case '+': return x + y;  
 case '-': return x - y;  
 case '\*': return x \* y;  
 case '/': return x / y;  
 }  
 throw new Exception("Unknown operator");  
 }  
}

Mit den genannten vier Klassen können Modelle arithmetischer Ausdrücke erstellt werden. Unter Verwendung von Instanzen dieser Klassen kann z. B. der Ausdruck x + 3 wie folgt dargestellt werden.

Expression e = new Operation(  
 new VariableReference("x"),  
 '+',  
 new Constant(3));

Die Evaluate-Methode einer Expression-Instanz wird aufgerufen, um den angegebenen Ausdruck auszuwerten und einen double-Wert zu generieren. Die Methode akzeptiert als Argument eine Hashtable mit Variablennamen (als Schlüssel der Einträge) und Werten (als Werte der Einträge). Die Evaluate-Methode ist eine virtuelle abstrakte Methode, d. h., nicht abstrakte Klassen müssen diese Methode überschreiben, um die eigentliche Implementierung bereitzustellen.

Die Evaluate-Implementierung von Constant gibt einfach die gespeicherte Konstante zurück. Die VariableReference-Implementierung liest den Variablennamen aus der Hashtabelle aus und gibt den resultierenden Wert zurück. In der Operation-Implementierung wird zunächst der linke und rechte Operand ausgewertet (durch rekursiven Aufruf der entsprechenden Evaluate-Methode) und dann die angegebene arithmetische Operation ausgeführt.

Im folgenden Programm werden die Expression-Klassen verwendet, um den Ausdruck x \* (y + 2) für verschiedene Werte von x und y auszuwerten.

using System;  
using System.Collections;

class Test  
{  
 static void Main() {

Expression e = new Operation(  
 new VariableReference("x"),  
 '\*',  
 new Operation(  
 new VariableReference("y"),  
 '+',  
 new Constant(2)  
 )  
 );

Hashtable vars = new Hashtable();

vars["x"] = 3;  
 vars["y"] = 5;  
 Console.WriteLine(e.Evaluate(vars)); // Outputs "21"

vars["x"] = 1.5;  
 vars["y"] = 9;  
 Console.WriteLine(e.Evaluate(vars)); // Outputs "16.5"  
 }  
}

#### Methodenüberladungen

Methodenüberladungen erlauben mehrere Methoden in der gleichen Klasse mit dem gleichen Namen, sofern jede dieser Methode eine eindeutige Signatur besitzt. Beim Kompilieren eines Aufrufs einer überladenen Methode verwendet der Compiler die Überladungsauflösung, um die spezielle aufzurufende Methode zu bestimmen. Bei der Überladungsauflösung wird die Methode gesucht, die am besten mit den Argumenten übereinstimmt. Wenn keine eindeutige Übereinstimmung gefunden wird, wird ein Fehler ausgegeben. Im folgenden Beispiel wird die Funktionsweise der Überladungsauflösung veranschaulicht. Die Kommentare zu den einzelnen Aufrufen in der Main-Methode geben die tatsächlich aufgerufenen Methoden an.

class Test  
{  
 static void F() {  
 Console.WriteLine("F()");  
 }

static void F(object x) {  
 Console.WriteLine("F(object)");  
 }

static void F(int x) {  
 Console.WriteLine("F(int)");  
 }

static void F(double x) {  
 Console.WriteLine("F(double)");  
 }

static void F<T>(T x) {  
 Console.WriteLine("F<T>(T)");  
 }

static void F(double x, double y) {  
 Console.WriteLine("F(double, double)");  
 }

static void Main() {  
 F(); // Invokes F()  
 F(1); // Invokes F(int)  
 F(1.0); // Invokes F(double)  
 F("abc"); // Invokes F(object)  
 F((double)1); // Invokes F(double)  
 F((object)1); // Invokes F(object)  
 F<int>(1); // Invokes F<T>(T)  
 F(1, 1); // Invokes F(double, double) }  
}

Im Beispiel wird auch gezeigt, wie eine bestimmte Methode in jedem Fall aufgerufen werden kann, indem die Argumente in die korrekten Parametertypen umgewandelt werden und/oder indem explizit Typargumente angegeben werden.

### Weitere Funktionsmember

Member, die ausführbaren Code enthalten, werden unter dem Begriff Funktionsmember einer Klasse zusammengefasst. Die im vorausgehenden Abschnitt beschriebenen Methoden stellen die wichtigste Art von Funktionsmembern dar. In diesem Abschnitt werden weitere Arten von Funktionsmembern beschrieben, die von C# unterstützt werden: Konstruktoren, Eigenschaften, Indexer, Ereignisse, Operationen und Destruktoren.

Die folgenden Tabelle zeigt eine generische Klasse mit dem Namen List<T>, die eine erweiterbare Liste von Objekten implementiert. Die Klasse enthält mehrere Beispiele für die üblichsten Arten von Funktionsmembern.

|  |  |
| --- | --- |
| public class List<T> { | |
| const int defaultCapacity = 4; | Konstante |
| T[] items;  int count; | Felder |
| public List(int capacity = defaultCapacity) {  items = new T[capacity];  } | Konstruktoren |
| public int Count {  get { return count; }  }  public int Capacity {  get {  return items.Length;  }  set {  if (value < count) value = count;  if (value != items.Length) {  T[] newItems = new T[value];  Array.Copy(items, 0, newItems, 0, count);  items = newItems;  }  }  } | Eigenschaften |

|  |  |
| --- | --- |
| public T this[int index] {  get {  return items[index];  }  set {  items[index] = value;  OnChanged();  }  } | Indexer |
| public void Add(T item) {  if (count == Capacity) Capacity = count \* 2;  items[count] = item;  count++;  OnChanged();  }  protected virtual void OnChanged() {  if (Changed != null) Changed(this, EventArgs.Empty);  }  public override bool Equals(object other) {  return Equals(this, other as List<T>);  }  static bool Equals(List<T> a, List<T> b) {  if (a == null) return b == null;  if (b == null || a.count != b.count) return false;  for (int i = 0; i < a.count; i++) {  if (!object.Equals(a.items[i], b.items[i])) {  return false;  }  }  return true;  } | Methoden |
| public event EventHandler Changed; | Ereignis |
| public static bool operator ==(List<T> a, List<T> b) {  return Equals(a, b);  }  public static bool operator !=(List<T> a, List<T> b) {  return !Equals(a, b);  } | Operatoren |
| } | |

#### Konstruktoren

C# unterstützt Instanzen- und statische Konstruktoren. Ein Instanzkonstruktor ist ein Member, der die Aktionen implementiert, die zum Initialisieren der Instanz einer Klasse erforderlich sind. Ein statischer Konstruktor ist ein Member, der die Aktionen implementiert, die zum Initialisieren der Klasse selbst erforderlich sind, wenn sie zum ersten Mal geladen wird.

Konstruktoren werden wie Methoden ohne Rückgabetyp und mit dem Namen der Klasse deklariert, in der sie enthalten sind. Wenn die Deklaration eines Konstruktors einen static-Modifizierer enthält, wird mit ihr ein statischer Konstruktor deklariert. Andernfalls wird ein Instanzkonstruktor deklariert.

Instanzkonstruktoren können überladen werden. Die List<T>-Klasse deklariert z. B. zwei Instanzkonstruktoren, einen ohne Parameter und einen mit einem int-Parameter. Instanzkonstruktoren werden mit dem new-Operator aufgerufen. Mit den folgenden Anweisungen werden zwei List<string>-Instanzen zugewiesen, die jeweils einen der Konstruktoren der List-Klasse verwenden.

List<string> list1 = new List<string>();  
List<string> list2 = new List<string>(10);

Im Gegensatz zu anderen Membern werden Instanzkonstruktoren nicht vererbt, und eine Klasse verfügt nur über die in der Klasse selbst deklarierten Instanzkonstruktoren. Wenn für eine Klasse kein Instanzkonstruktor angegeben ist, wird automatisch ein leerer, parameterloser Instanzkonstruktor bereitgestellt.

#### Eigenschaften

Eigenschaften sind eine logische Weiterentwicklung von Feldern. Beide sind benannte Member mit verknüpften Typen. Außerdem ist die Syntax für den Zugriff auf Felder und Eigenschaften identisch. Im Unterschied zu Feldern kennzeichnen Eigenschaften jedoch keine Speicherorte. Stattdessen besitzen Eigenschaften Accessoren, die die Anweisungen angeben, die beim Lesen bzw. Schreiben ihrer Werte auszuführen sind.

Eigenschaften werden wie Felder deklariert mit dem Unterschied, dass die Deklaration nicht mit einem Semikolon endet, sondern mit einem get-Accessor und/oder einem set-Accessor, die zwischen den Trennzeichen { und } stehen. Eine Eigenschaft, die einen get-Accessor und einen set-Accessor besitzt, ist eine Eigenschaft mit Lese-/Schreibzugriff, eine Eigenschaft, die nur über einen get-Accessor verfügt, ist eine schreibgeschützte Eigenschaft, und eine Eigenschaft, die nur einen set-Accessor besitzt, ist eine lesegeschützte Eigenschaft.

Ein get-Accessor entspricht einer parameterlosen Methode mit einem Rückgabewert vom Typ der Eigenschaft. Wenn auf eine Eigenschaft in einem Ausdruck verwiesen wird, wird der get-Accessor der Eigenschaft aufgerufen, um den Wert der Eigenschaft zu berechnen. Dies gilt nicht, wenn der Accessor das Ziel einer Zuweisung ist.

Ein set-Accessor entspricht einer Methode mit einem einzigen Parameter mit dem Namen value und ohne Rückgabetyp. Wenn auf eine Eigenschaft als Ziel einer Zuweisung oder als Operanden für ++ oder -- verwiesen wird, wird der set-Accessor mit einem Argument aufgerufen, das den neuen Wert bereitstellt.

In der List<T>-Klasse werden die beiden Eigenschaften Count und Capacity deklariert, ersteres als schreibgeschützt, letzteres mit Lese-/Schreibzugriff. Im folgenden Beispiel werden diese Eigenschaften verwendet.

List<string> names = new List<string>();  
names.Capacity = 100; // Invokes set accessor  
int i = names.Count; // Invokes get accessor  
int j = names.Capacity; // Invokes get accessor

Wie bei Feldern und Methoden unterstützt C# auch bei Eigenschaften Instanzeigenschaften und statische Eigenschaften. Statische Eigenschaften werden mit dem static-Modifizierer und Instanzeigenschaften ohne diesen Modifizierer deklariert.

Die Accessoren einer Eigenschaft können virtuell sein. Wenn die Eigenschaftendeklaration einen der Modifizierer virtual, abstract oder override enthält, gilt dieser für die Accessoren der Eigenschaft.

#### Indexer

Indexer sind Member, die es ermöglichen, Objekte wie Arrays zu indizieren. Indexer werden wie Eigenschaften deklariert, mit dem Unterschied, dass der Name des Members this gefolgt von einer Parameterliste zwischen den Trennzeichen [ und ] ist. Die Parameter stehen in den Accessoren des Indexers zur Verfügung. Wie Eigenschaften können Indexer Lese-/Schreibzugriff besitzen, schreibgeschützt oder lesegeschützt sein. Außerdem können die Accessoren eines Indexers virtuell sein.

Die List-Klasse deklariert einen einzelnen Indexer mit Lese-/Schreibzugriff, der einen int-Parameter akzeptiert. Der Indexer ermöglicht ein Indizieren von List-Instanzen mit int-Werten. Beispiel:

List<string> names = new List<string>();  
names.Add("Liz");  
names.Add("Martha");  
names.Add("Beth");  
for (int i = 0; i < names.Count; i++) {  
 string s = names[i];  
 names[i] = s.ToUpper();  
}

Indexer können überladen werden, d. h., eine Klasse kann mehrere Indexer deklarieren, sofern sie sich in Anzahl oder Typ ihrer jeweiligen Parameter unterscheiden.

#### Ereignisse

Ereignisse sind Member, die es Klassen und Objekten ermöglichen, Benachrichtigungen auszugeben. Ereignisse werden wie Felder deklariert, mit dem Unterschied, dass in der Deklaration das Schlüsselwort event enthalten ist und der Typ ein Delegattyp sein muss.

Innerhalb einer Klasse, die einen Ereignismember deklariert, verhält sich das Ereignis wie ein Feld von einem Delegattyp (vorausgesetzt, dass das Ereignis nicht abstrakt ist und keine Accessoren deklariert). Im Feld wird ein Verweis auf einen Delegaten gespeichert, der die Ereignishandler darstellt, die dem Ereignis hinzugefügt wurden. Wenn keine Ereignishandler vorhanden sind, hat das Feld den Wert null.

Die List<T>-Klasse deklariert einen einzelnen Ereignismember mit dem Namen Changed, der angibt, dass der Liste ein neues Element hinzugefügt wurde. Das Changed-Ereignis wird von der virtuellen OnChanged-Methode ausgelöst, das als Erstes überprüft, ob das Ereignis null ist (d. h., ob keine Ereignishandler vorhanden sind). Da das Auslösen eines Ereignisses dem Aufrufen des Delegaten entspricht, der von dem Ereignis dargestellt wird, sind keine speziellen Sprachkonstrukte zum Auslösen von Ereignissen erforderlich.

Clients reagieren in Ereignishandlern auf Ereignisse. Ereignishandler werden mit dem +=-Operator angefügt und mit dem -=-Operator entfernt. Im folgenden Beispiel wird ein Ereignishandler an das Changed-Ereignis einer List<string> angefügt.

using System;

class Test  
{  
 static int changeCount;

static void ListChanged(object sender, EventArgs e) {  
 changeCount++;  
 }

static void Main() {  
 List<string> names = new List<string>();  
 names.Changed += new EventHandler(ListChanged);  
 names.Add("Liz");  
 names.Add("Martha");  
 names.Add("Beth");  
 Console.WriteLine(changeCount); // Outputs "3"  
 }  
}

In umfassenden Szenarios, in denen eine Steuerung des zugrunde liegenden Speichervorgangs eines Ereignisses benötigt wird, können Ereignisdeklarationen explizit add- und remove-Accessoren bereitstellen, die in gewisser Weise mit dem set-Accessor einer Eigenschaft vergleichbar sind.

#### Operatoren

Operatoren sind Member, mit denen definiert wird, was das Anwenden eines bestimmten Ausdrucksoperators auf die Instanzen einer Klasse bedeuten soll. Drei Arten von Operatoren können definiert werden: unäre Operatoren, binäre Operatoren und Konvertierungsoperatoren. Alle Operatoren müssen als public und static deklariert werden.

In der List<T>-Klasse werden die beiden Operatoren operator == und operator != deklariert und so den Ausdrücken, die diese Operatoren auf List-Instanzen anwenden, neue Bedeutungen verliehen. Genauer gesagt, definieren die Operatoren die Gleichheit zweier List<T>-Instanzen als den Vergleich aller enthaltenen Objekte mithilfe der jeweiligen Equals-Methoden. Im folgenden Beispiel wird der ==-Operator zum Vergleichen zweier List<int>-Instanzen verwendet.

using System;

class Test  
{  
 static void Main() {  
 List<int> a = new List<int>();  
 a.Add(1);  
 a.Add(2);  
 List<int> b = new List<int>();  
 b.Add(1);  
 b.Add(2);  
 Console.WriteLine(a == b); // Outputs "True"   
 b.Add(3);  
 Console.WriteLine(a == b); // Outputs "False"  
 }  
}

Die erste Console.WriteLine-Anweisung gibt True aus, da die beiden Listen die gleiche Anzahl von Objekten mit den gleichen Werten in derselben Reihenfolge enthalten. Wenn operator == in List<T> nicht definiert wird, gibt die erste Console.WriteLine-Anweisung False aus, da a und b auf unterschiedliche List<int>-Instanzen verweisen.

#### Destruktoren

Destruktoren sind Member, die die Aktionen implementieren, die zum Zerstören einer Klasseninstanz erforderlich sind. Destruktoren können weder Parameter noch Zugriffsmodifizierer besitzen, und sie können nicht explizit aufgerufen werden. Der Destruktor einer Instanz wird automatisch während der Ausführung der Garbage Collection aufgerufen.

Der Garbage Collector hat einen großen Spielraum bei der Auslegung der Kriterien, die zum Freigeben von Objekten und Ausführen von Destruktoren führen. Insbesondere ist der Zeitpunkt eines Destruktorenaufrufs nicht zwingend vorgegeben, und Destruktoren können in beliebigen Threads ausgeführt werden. Unter anderem aus den genannten Gründen sollten Klassen nur dann Destruktoren implementieren, wenn andere Lösungswege nicht praktikabel sind.

Die using-Anweisung bietet eine bessere Lösung für die Objektzerstörung.

## Strukturen

Wie Klassen sind Strukturen Datenstrukturen, die Datenmember und Funktionsmember enthalten können. Der Unterschied besteht darin, dass Strukturen Werttypen sind, für die keine Heapzuordnung erforderlich ist. Variablen eines Strukturtyps speichern die Daten der Struktur selbst, wohingegen Variablen einer Klasse einen Verweis auf ein dynamisch zugeordnetes Objekt speichern. Bei Strukturtypen wird keine benutzerdefinierte Vererbung unterstützt, und alle Strukturtypen werden implizit vom Typ object vererbt.

Strukturen eignen sich besonders für kleine Datenstrukturen, die über Wertsemantik verfügen. Geeignete Beispiele für Strukturen sind komplexe Zahlen, Punkte in einem Koordinatensystem und Schlüsselwertpaare in einem Wörterbuch. Die Verwendung von Strukturen anstelle von Klassen für kleine Datenstrukturen, kann sich deutlich auf die Anzahl der Speicherbelegungen auswirken, die von einer Anwendung ausgeführt werden. Mit dem folgenden Programm wird beispielsweise ein Array von 100 Punkten erstellt und initialisiert. Wenn Point als Klasse implementiert ist, werden 101 separate Objekte instanziiert, eines für das Array und eines für jedes der 100 Elemente.

class Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

class Test  
{  
 static void Main() {  
 Point[] points = new Point[100];  
 for (int i = 0; i < 100; i++) points[i] = new Point(i, i);  
 }  
}

Eine Alternative besteht darin, Point als Struktur zu implementieren.

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Hierbei wird nur ein Objekt instanziiert – das Arrayobjekt. Die Point-Instanzen werden inline im Array gespeichert.

Strukturkonstruktoren werden zwar mit dem new-Operator aufgerufen, dies bedeutet aber nicht, dass eine Speicherbelegung stattfindet. Statt ein Objekt dynamisch zuzuordnen und einen Verweis auf dieses Objekt zurückzugeben, gibt ein Strukturkonstruktor einfach die Struktur selbst zurück (normalerweise an einem temporären Speicherort auf dem Stapel), und dieser Wert wird dann nach Bedarf kopiert.

Bei Klassen besteht die Möglichkeit, dass zwei Variablen auf dasselbe Objekt verweisen. Somit ist es möglich, dass Operationen, die die eine Variable betreffen, auch Auswirkungen auf das von der anderen Variablen referenzierte Objekt haben. Bei Strukturen besitzt jede Variable ihre eigene Kopie der Daten. Operationen beeinflussen daher immer nur eine Variable. Beispielsweise hängt die Ausgabe des folgenden Codefragments davon ab, ob Point eine Klasse oder eine Struktur ist.

Point a = new Point(10, 10);  
Point b = a;  
a.x = 20;  
Console.WriteLine(b.x);

Wenn Pointeine Klasse ist, ist die Ausgabe 20, weil a und b auf das gleiche Objekt verweisen. Wenn Point eine Struktur ist, ist die Ausgabe 10, weil bei der Zuweisung von a zu b eine neue Kopie des Werts erstellt wird und diese Kopie von der folgenden Zuweisung zu a.x nicht betroffen ist.

Im vorhergehenden Beispiel werden zwei Einschränkungen von Strukturen herausgestellt. Erstens ist das Kopieren einer gesamten Struktur i. d. R. weniger effizient als das Kopieren eines Objektverweises, sodass Zuweisungen und Übergaben von Wertparametern bei Strukturen aufwändiger als bei Verweistypen sein können. Zweitens ist es mit Ausnahme des ref-Parameters und des out-Parameters nicht möglich, Verweise auf Strukturen zu erstellen, sodass ihre Verwendung in einer Reihe von Situationen nicht infrage kommt.

## Arrays

Arrays sind Datenstrukturen mit einer Reihe von Variablen, auf die über berechnete Indizes zugegriffen wird. Die in einem Array enthaltenen Variablen, die auch als die Elemente des Arrays bezeichnet werden, sind alle vom selben Typ. Dieser Typ wird als Elementtyp des Arrays bezeichnet.

Arraytypen sind Verweistypen, und bei der Deklaration einer Arrayvariablen wird einfach Speicher für einen Verweis auf eine Arrayinstanz reserviert. Die eigentlichen Arrayinstanzen werden dynamisch zur Laufzeit mithilfe des new-Operators erstellt. Beim new-Vorgang wird die Länge des neuen Arrays angegeben, die nicht mehr geändert werden kann, solange die Instanz besteht. Die Indizes der Elemente eines Arrays liegen zwischen 0 und Length - 1. Der new-Operator initialisiert automatisch die Elemente eines Arrays mit dem entsprechenden Standardwert, z. B. 0 (null) bei numerischen Typen und null für alle Verweistypen.

Im folgenden Beispiel wird ein Array von int-Elementen erstellt, initialisiert und der Inhalt des Arrays dann ausgegeben.

using System;

class Test  
{  
 static void Main() {  
 int[] a = new int[10];  
 for (int i = 0; i < a.Length; i++) {  
 a[i] = i \* i;  
 }  
 for (int i = 0; i < a.Length; i++) {  
 Console.WriteLine("a[{0}] = {1}", i, a[i]);  
 }  
 }  
}

In diesem Beispiel wird ein eindimensionales Array erstellt und verarbeitet. C# unterstützt ebenfalls mehrdimensionale Arrays. Die Anzahl der Dimensionen eines Arraytyps, auch als Rang des Arraytyps bezeichnet, ist eins plus die Anzahl von Kommas, die in den eckigen Klammern des Arraytyps stehen. Im folgenden Beispiel werden ein eindimensionales, ein zweidimensionales und ein dreidimensionales Array zugeordnet.

int[] a1 = new int[10];  
int[,] a2 = new int[10, 5];  
int[,,] a3 = new int[10, 5, 2];

Das a1-Array enthält 10 Elemente, das a2-Array 50 (10 × 5) Elemente und das a3-Array 100 (10 × 5 × 2) Elemente.

Der Elementtyp eines Arrays kann ein beliebiger Typ sein, z. B. auch einen Arraytyp. Arrays mit Elementen vom Typ Array werden zuweilen auch als verzweigte Arrays bezeichnet, weil die Elementarrays unterschiedliche Längen besitzen können. Im folgenden Beispiel wird ein Array von int-Arrays zugeordnet:

int[][] a = new int[3][];  
a[0] = new int[10];  
a[1] = new int[5];  
a[2] = new int[20];

In der ersten Zeile wird ein Array mit drei Elementen erstellt, die jeweils vom Typ int[] sind und den Anfangswert null besitzen. In den folgenden Zeilen werden die drei Elemente mit Verweisen auf einzelne Arrayinstanzen unterschiedlicher Länge initialisiert.

Der new-Operator ermöglicht es, den Anfangswert der Arrayelemente mit einem Arrayinitialisierer anzugeben. Dabei handelt es sich um eine Liste von Ausdrücken zwischen den Trennzeichen { und }. Im folgenden Beispiel wird ein int[] mit drei Elementen zugeordnet und initialisiert.

int[] a = new int[] {1, 2, 3};

Beachten Sie, dass die Länge des Arrays aus der Anzahl der Ausdrücke zwischen { und } abgeleitet wird. Die Deklaration der lokalen Variablen und der Felder kann so verkürzt werden, dass der Arraytyp nicht mehrfach aufgeführt werden muss.

int[] a = {1, 2, 3};

Die beiden vorhergehenden Beispiele entsprechen dem folgendem:

int[] t = new int[3];  
t[0] = 1;  
t[1] = 2;  
t[2] = 3;  
int[] a = t;

## Schnittstellen

Schnittstellen definieren einen "Vertrag", den Klassen und Strukturen implementieren können. Schnittstellen können Methoden, Eigenschaften, Ereignisse und Indexer enthalten. Schnittstellen stellen keine Implementierungen der von ihnen definierte Member bereit, sondern geben lediglich die Member an, die von den Klassen oder Strukturen bereitgestellt werden müssen, die die Schnittstelle implementieren.

Schnittstellen können die Mehrfachvererbung verwenden. Im folgenden Beispiel erbt die Schnittstelle IComboBox von ITextBox und IListBox.

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

interface IComboBox: ITextBox, IListBox {}

Klassen und Strukturen können mehrere Schnittstellen implementieren. Im folgenden Beispiel implementiert die EditBox-Klasse sowohl IControl als auch IDataBound.

interface IDataBound  
{  
 void Bind(Binder b);  
}

public class EditBox: IControl, IDataBound  
{  
 public void Paint() {...}

public void Bind(Binder b) {...}  
}

Wenn eine Klasse oder Struktur eine bestimmte Schnittstelle implementiert, können Instanzen dieser Klasse oder Struktur implizit in diesen Schnittstellentyp konvertiert werden. Beispiel:

EditBox editBox = new EditBox();  
IControl control = editBox;  
IDataBound dataBound = editBox;

In Fällen, in denen die Implementierung einer bestimmten Schnittstelle durch eine Instanz nicht statisch festgelegt ist, können dynamische Typumwandlungen verwendet werden. Beispielsweise wird in den folgenden Anweisungen die dynamische Typumwandlung verwendet, um die Schnittstellenimplementierungen IControl und IDataBound eines Objekts abzurufen. Da der eigentliche Typ des Objekts EditBox ist, sind die Typumwandlungen erfolgreich.

object obj = new EditBox();  
IControl control = (IControl)obj;  
IDataBound dataBound = (IDataBound)obj;

In der genannten EditBox-Klasse sind die Paint-Methode der IControl-Schnittstelle und die Bind-Methode der IDataBound-Schnittstelle als public-Member implementiert. C# unterstützt auch explizite Implementierungen von Schnittstellenmembern, wodurch Member in Klassen oder Strukturen nicht public sein müssen. Eine explizite Implementierung von Schnittstellenmembern wird mithilfe des vollqualifizierten Namens des Schnittstellenmembers angegeben. Beispielweise kann die EditBox-Klasse mithilfe der expliziten Implementierungen von Schnittstellenmembern die IControl.Paint-Methode und die IDataBound.Bind-Methode wie folgt implementieren.

public class EditBox: IControl, IDataBound  
{  
 void IControl.Paint() {...}

void IDataBound.Bind(Binder b) {...}  
}

Der Zugriff auf explizite Schnittstellenmember ist nur über den Schnittstellentyp möglich. Beispielsweise kann die Implementierung von IControl.Paint in der genannten EditBox-Klasse nur nach Konvertieren des EditBox-Verweises in den IControl-Schnittstellentyp aufgerufen werden.

EditBox editBox = new EditBox();  
editBox.Paint(); // Error, no such method  
IControl control = editBox;  
control.Paint(); // Ok

## Enumerationen

Enumerationstypen sind eigenständige Werttypen mit einer Gruppe benannter Konstanten. Im folgenden Beispiel wird ein Enumerationstyp mit dem Namen Color deklariert und verwendet, der die drei Konstantenwerte Red, Green und Blue enthält.

using System;

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

class Test  
{  
 static void PrintColor(Color color) {  
 switch (color) {  
 case Color.Red:  
 Console.WriteLine("Red");  
 break;  
 case Color.Green:  
 Console.WriteLine("Green");  
 break;  
 case Color.Blue:  
 Console.WriteLine("Blue");  
 break;  
 default:  
 Console.WriteLine("Unknown color");  
 break;  
 }  
 }

static void Main() {  
 Color c = Color.Red;  
 PrintColor(c);  
 PrintColor(Color.Blue);  
 }  
}

Jeder Enumerationstyp besitzt einen entsprechenden ganzzahligen Typ, der als zugrunde liegender Typ des Enumerationstyps bezeichnet wird. Bei einem Enumerationstyp ohne explizite Angabe des zugrunde liegenden Typs ist der zugrunde liegende Typ int. Der zugrunde liegende Typ entscheidet über das Speicherformat und den Bereich möglicher Werte eines Enumerationstyps. Die Gruppe von Werten, die ein Enumerationstyp annehmen kann, ist nicht durch die Enumerationsmember begrenzt. Genauer gesagt, kann jeder Wert des zugrunde liegenden Typs einer Enumeration in den Enumerationstyp umgewandelt werden und so einen eigenständigen gültigen Wert dieses Enumerationstyps darstellen.

Im folgenden Beispiel wird ein Enumerationstyp mit dem Namen Alignment und dem zugrunde liegenden Typ sbyte deklariert.

enum Alignment: sbyte  
{  
 Left = -1,  
 Center = 0,  
 Right = 1  
}

Wie das vorhergehende Beispiel veranschaulicht, kann die Deklaration eines Enumerationsmembers einen konstanten Ausdruck enthalten, der den Wert des Members angibt. Der konstante Wert jedes Enumerationsmembers muss im Bereich des zugrunde liegenden Typs der Enumeration liegen. Wenn die Deklaration eines Enumerationsmembers den Wert nicht explizit angibt, wird der Member auf den Wert 0 (null) festgelegt (wenn er der erste Member im Enumerationstyp ist) oder auf den Wert des Enumerationsmembers, der im Code vorausgeht, plus eins.

Enumerationswerte können mithilfe von Typumwandlungen in ganzzahlige Werte und umgekehrt umgewandelt werden. Beispiel:

int i = (int)Color.Blue; // int i = 2;  
Color c = (Color)2; // Color c = Color.Blue;

Der Standardwert eines beliebigen Enumerationstyps ist der ganzzahlige Wert 0 (null), der in den Enumerationstyp konvertiert wird. In den Fällen, in denen Variablen automatisch mit dem Standardwert initialisiert werden, ist dies der Wert, auf den Variablen des Enumerationstyps festgelegt werden. Damit der Standardwert eines Enumerationstyps einfach verfügbar ist, wird der literale Wert 0 implizit in jeden beliebigen Enumerationstyp konvertiert. Somit ist die folgende Anweisung zulässig.

Color c = 0;

## Delegaten

Ein Delegattyp stellt Verweise auf Methoden mit einer bestimmten Parameterliste und einem bestimmten Rückgabetyp dar. Delegaten erlauben es, Methoden als Einheiten zu behandeln, die Variablen zugewiesen und als Parameter übergeben werden können. Somit ähneln Delegaten dem Konzept der Funktionszeiger, wie sie in anderen Programmiersprachen unterstützt werden. Im Gegensatz zu Funktionszeigern sind Delegaten jedoch objektorientiert und typsicher.

Im folgenden Beispiel wird ein Delegattyp mit dem Namen Function deklariert und verwendet.

using System;

delegate double Function(double x);

class Multiplier  
{  
 double factor;

public Multiplier(double factor) {  
 this.factor = factor;  
 }

public double Multiply(double x) {  
 return x \* factor;  
 }  
}

class Test  
{  
 static double Square(double x) {  
 return x \* x;  
 }

static double[] Apply(double[] a, Function f) {  
 double[] result = new double[a.Length];  
 for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);  
 return result;  
 }

static void Main() {  
 double[] a = {0.0, 0.5, 1.0};

double[] squares = Apply(a, Square);

double[] sines = Apply(a, Math.Sin);

Multiplier m = new Multiplier(2.0);  
 double[] doubles = Apply(a, m.Multiply);  
 }  
}

Eine Instanz des Function-Delegattypen kann auf eine beliebige Methode verweisen, die ein double-Argument akzeptiert und einen double-Wert zurückgibt. Die Apply-Methode wendet eine angegebene Function auf die Elemente eines double[] an und gibt ein double[] mit den Ergebnissen zurück. In der Main-Methode wird Apply verwendet, um drei unterschiedliche Funktionen auf einen double[] anzuwenden.

Ein Delegat kann entweder auf eine statische Methode (z. B. Square oder Math.Sin im vorausgehenden Beispiel) oder auf eine Instanzmethode (z. B. m.Multiply im vorausgehenden Beispiel) verweisen. Ein Delegat, der auf eine Instanzmethode verweist, verweist auch auf ein bestimmtes Objekt, und wenn die Instanzmethode über den Delegaten aufgerufen wird, wird dieses Objekt im Aufruf zu this.

Delegaten können auch mit anonymen Funktionen erstellt werden, bei denen es sich um „inline-Methoden“ handelt, die direkt erstellt werden. Anonyme Funktionen haben Zugriff auf die lokalen Variablen der umgebenden Methoden. Dadurch kann das oben stehende Multiplikationsbeispiel noch einfacher ohne eine Multiplier-Klasse geschrieben werden:

double[] doubles = Apply(a, (double x) => x \* 2.0);

Ein interessantes und nützliches Merkmal von Delegaten besteht darin, dass die Klasse der Methode, auf die verwiesen wird, keine Rolle spielt. Es ist lediglich von Bedeutung, dass die referenzierte Methode die gleichen Parameter und den gleichen Rückgabetyp wie der Delegat besitzt.

## Attribute

Typen, Member und andere Einheiten in einem C#-Programm unterstützen Modifizierer, mit denen bestimmte Aspekte ihres Verhaltens gesteuert werden können. Beispielsweise kann der Zugriff auf eine Methode mit den Modifizierern public, protected, internal und private gesteuert werden. In C# ist diese Möglichkeit derart verallgemeinert, dass benutzerdefinierte Typen deklarativer Informationen an Programmeinheiten angefügt und zur Laufzeit abgerufen werden können. Programme legen diese zusätzlichen deklarativen Informationen durch die Definition und Verwendung von Attributen fest.

Im folgenden Beispiel wird ein HelpAttribute-Attribut definiert, das für Programmeinheiten angegeben werden kann, um Verknüpfungen mit der zugeordneten Dokumentation anzugeben.

using System;

public class HelpAttribute: Attribute  
{  
 string url;  
 string topic;

public HelpAttribute(string url) {  
 this.url = url;  
 }

public string Url {   
 get { return url; }  
 }

public string Topic {  
 get { return topic; }  
 set { topic = value; }  
 }  
}

Alle Attributklassen sind von der System.Attribute-Basisklasse abgeleitet, die von .NET Framework bereitgestellt wird. Die Anwendung von Attributen erfolgt über ihren Namen, der zusammen mit sämtlichen Argumenten in eckigen Klammern direkt vor der zugehörigen Deklaration angegeben wird. Wenn der Name eines Attributs auf Attribute endet, kann dieser Teil des Namens beim Verweisen auf das Attribut weggelassen werden. Beispielsweise kann das HelpAttribute wie folgt verwendet werden.

[Help("http://msdn.microsoft.com/.../MyClass.htm")]  
public class Widget  
{  
 [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]  
 public void Display(string text) {}  
}

In diesem Beispiel wird ein HelpAttribute an die Widget-Klasse und ein anderes HelpAttribute an die Display-Methode in der Klasse angefügt werden. Die öffentlichen Konstruktoren einer Attributklasse steuern die Informationen, die beim Anfügen des Attributs an eine Programmeinheit angegeben werden müssen. Zusätzliche Informationen können durch Verweisen auf öffentliche Eigenschaften der Attributklasse mit Lese-/Schreibzugriff (z. B. der oben genannte Verweis auf die Topic-Eigenschaft) bereitgestellt werden.

Im folgenden Beispiel wird das Abrufen von Attributinformationen für eine bestimmte Programmeinheit zur Laufzeit mithilfe der Reflektion veranschaulicht.

using System;  
using System.Reflection;

class Test  
{  
 static void ShowHelp(MemberInfo member) {  
 HelpAttribute a = Attribute.GetCustomAttribute(member,  
 typeof(HelpAttribute)) as HelpAttribute;  
 if (a == null) {  
 Console.WriteLine("No help for {0}", member);  
 }  
 else {  
 Console.WriteLine("Help for {0}:", member);  
 Console.WriteLine(" Url={0}, Topic={1}", a.Url, a.Topic);  
 }  
 }

static void Main() {  
 ShowHelp(typeof(Widget));  
 ShowHelp(typeof(Widget).GetMethod("Display"));  
 }  
}

Wenn ein bestimmtes Attribut über die Reflektion angefordert wird, wird der Konstruktor für die Attributklasse mit den Informationen aufgerufen, die in der Programmquelle angegeben wurden, und die resultierende Attributinstanz wird zurückgegeben. Wenn zusätzliche Informationen über Eigenschaften bereitgestellt werden, werden diese Eigenschaften vor dem Zurückgeben der Attributinstanz auf die angegebenen Werte festgelegt.

# Lexikalische Struktur

## Programme

Ein C#-Programm besteht aus einer oder mehreren Quelldateien, die offiziell als Kompilationseinheit bezeichnet werden (§9.1). Eine Quelldatei ist eine geordnete Abfolge von Unicode-Zeichen. In der Regel besitzen Quelldateien eine 1:1-Übereinstimmung mit Dateien in einem Dateisystem. Diese Übereinstimmung ist jedoch nicht erforderlich. Um die bestmögliche Portierbarkeit sicherzustellen, wird empfohlen, als Codierung der Dateien im Dateisystem die UTF-8-Codierung zu verwenden.

Konzeptuell gesehen, wird ein Programm in drei Arbeitsschritten kompiliert:

1. Transformation, wobei eine Datei aus einem bestimmten Zeichenrepertoire und Codierungsschema in eine Abfolge von Unicode-Zeichen codiert wird.
2. Lexikalische Analyse, wobei ein Stream von Unicode-Eingabezeichen in einen Stream von Token übersetzt wird.
3. Syntaktische Analyse, wobei der Stream von Token in ausführbaren Code übersetzt wird.

## Grammatiken

In dieser Beschreibung wird die Syntax der C#-Programmiersprache unter Verwendung zweier Grammatiken dargestellt. Die lexikalische Grammatik (§2.2.2) definiert, wie mittels Unicode-Zeichen Zeilenabschlusszeichen, Leerraum, Kommentare, Token und Präprozessordirektiven gebildet werden. Die syntaktische Grammatik (§2.2.3) definiert, wie aus den aus der lexikalischen Grammatik resultierenden Token C#-Programme gebildet werden.

### Grammatiknotation

Die lexikalischen und syntaktischen Grammatiken werden mittels Grammatikproduktionen dargestellt. Jede Grammatikproduktion definiert ein Nicht-Terminal-Symbol und die möglichen Erweiterungen dieses Nicht-Terminal-Symbols in Abfolgen von Nicht-Terminal-Symbolen und Terminal-Symbolen. non-terminal-Symbole werden in Grammatikproduktionen kursiv und terminal-Symbole in einer Schriftart mit fester Breite dargestellt.

In der ersten Zeile einer Grammatikproduktion befindet sich der Name des definierten Nicht-Terminal-Symbols, gefolgt von einem Doppelpunkt. Alle nachfolgenden eingerückten Zeilen enthalten eine mögliche Erweiterung des Nicht-Terminals, angegeben als Reihenfolge von Nicht-Terminal-Symbolen oder Terminal-Symbolen. So definiert die Produktion

while-statement:  
while ( boolean-expression ) embedded-statement

beispielsweise ein while-statement, das aus dem Token while gefolgt vom Token "(", einer boolean-expression und dem Token ")" sowie einem embedded-statement besteht.

Wenn es mehr als eine mögliche Erweiterung eines Nicht-Terminal-Symbols gibt, werden die alternativen Varianten in separaten Zeilen angeführt. So definiert die Produktion

statement-list:  
statement  
statement-list statement

eine statement-list, die entweder aus einem statement oder einer statement-list, gefolgt von einem statement, besteht. Mit anderen Worten, die Definition ist rekursiv und gibt an, dass eine Anweisungsliste aus einer oder mehreren Anweisungen besteht.

Das tiefgestellte Suffix "opt" gibt ein optionales Symbol an. Die Produktion:

block:  
{ statement-listopt }

stellt die Kurzform für

block:  
{ }  
{ statement-list }

dar und definiert, dass ein block aus einer optionalen, in die Token "{" und "}" eingeschlossenen statement-list besteht.

Alternativen werden i. d. R. auf separaten Zeilen aufgeführt. In Fällen mit zahlreichen Alternativen kann jedoch auch der Ausdruck „einer von“ vor einer Zeile mit einer Liste von Erweiterungen stehen. Dabei handelt es sich lediglich um eine Kurzform für die Auflistung von Alternativen in separaten Zeilen. So definiert die Produktion

real-type-suffix: one of  
F f D d M m

stellt die Kurzform für

real-type-suffix:  
F  
f  
D  
d  
M  
m

### Lexikalische Grammatik

Die lexikalische Grammatik von C# wird in §2.3, §2.4 und §2.5 vorgestellt. Die Terminal-Symbole der lexikalischen Grammatik sind die Zeichen des Unicode-Zeichensatzes. Die lexikalische Grammatik legt fest, wie Zeichen kombiniert werden, um Token (§2.4), Leerraum (§2.3.3), Kommentare (§2.3.2) und Präprozessordirektiven (§2.5) zu bilden.

Jede Quelldatei in einem C#-Programm muss der input-Produktion der lexikalischen Grammatik (§2.3) entsprechen.

### Syntaktische Grammatik

Die syntaktische Grammatik von C# wird in den Kapiteln und Anhängen nach diesem Kapitel dargestellt. Die Terminal-Symbole der syntaktischen Grammatik sind die von der lexikalischen Grammatik definierten Token. Die syntaktische Grammatik legt fest, wie Token zur Bildung von C#-Programmen kombiniert werden.

Jede Quelldatei in einem C#-Programm muss der compilation-unit-Produktion der syntaktischen Grammatik (§9.1) entsprechen.

## Lexikalische Analyse

Die input-Produktion definiert die lexikalische Struktur einer C#-Quelldatei. Jede Quelldatei in einem C#-Programm muss mit dieser lexikalischen Grammatikproduktion übereinstimmen.

input:  
input-sectionopt

input-section:  
input-section-part  
input-section input-section-part

input-section-part:  
input-elementsopt new-line  
pp-directive

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

Fünf Basiselemente bilden die lexikalische Struktur einer C#-Quelldatei: Zeilenabschlusszeichen (§2.3.1), Leerraum (§2.3.3), Kommentare (§2.3.2), Token (§2.4) und Präprozessordirektiven (§2.5). Von diesen Basiselementen sind in der syntaktischen Grammatik eines C#-Programms nur Token von Bedeutung (§2.2.3).

Die lexikalische Verarbeitung einer C#-Quelldatei besteht in der Reduzierung der Datei auf eine Reihenfolge von Token, die die Eingabe für die syntaktische Analyse darstellen. Zeilenabschlusszeichen, Leerräume und Kommentare können zur Trennung von Token verwendet werden. Präprozessordirektiven können festlegen, dass Abschnitte der Quelldatei übersprungen werden. Diese lexikalischen Elemente haben jedoch keinen Einfluss auf die syntaktische Struktur eines C#-Programms.

Wenn verschiedene lexikalische Grammatikproduktionen mit einer Zeichenfolge in der Quelldatei übereinstimmen, wird bei der lexikalischen Verarbeitung immer das längstmögliche lexikalische Element gebildet. Beispielsweise wird die Zeichenfolge // als Anfang eines einzeiligen Kommentars verarbeitet, da dieses lexikalische Element länger als ein einzelnes /-Token ist.

### Zeilenabschlusszeichen

Zeilenabschlusszeichen unterteilen die Zeichen einer C#-Quelldatei in Zeilen.

new-line:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

Aus Gründen der Kompatibilität mit Quellcodebearbeitungs-Tools, die Dateiendemarker hinzufügen, und um die Anzeige einer Quelldatei als eine Abfolge korrekt abgeschlossener Zeilen zu ermöglichen, werden die folgenden Transformationen in der angegebenen Reihenfolge für alle Quelldateien in einem C#-Programm vorgenommen:

* Wenn das letzte Zeichen der Quelldatei das STRG+Z-Zeichen (U+001A) ist, wird dieses Zeichen gelöscht.
* Wenn diese Quelldatei nicht leer und das letzte Zeichen der Quelldatei kein Wagenrücklauf (U+000D), Zeilenvorschub (U+000A), Zeilentrennzeichen (U+2028) oder Absatztrennzeichen (U+2029) ist, wird am Ende der Quelldatei ein Wagenrücklauf (U+000D hinzugefügt.

### Kommentare

Zwei Formen von Kommentaren werden unterstützt: einzeilige Kommentare und Kommentare mit Trennzeichen. Einzeilige Kommentare beginnen mit den Zeichen // und erstrecken sich bis zum Ende der Quellzeile. Kommentare mit Trennzeichen beginnen mit den Zeichen /\* und enden mit den Zeichen \*/. Kommentare mit Trennzeichen können sich über mehrere Zeilen erstrecken.

comment:  
single-line-comment  
delimited-comment

single-line-comment:  
// input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

delimited-comment:  
/\* delimited-comment-textopt asterisks /

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
/  
asterisksopt not-slash-or-asterisk

asterisks:  
\*  
asterisks \*

not-slash-or-asterisk:  
Any Unicode character except / or \*

Kommentare können nicht geschachtelt werden. Die Zeichenfolgen /\* und \*/ besitzen innerhalb eines //-Kommentars und die Zeichenfolgen // und /\* innerhalb eines Kommentars mit Trennzeichen keine besondere Bedeutung.

Anmerkungen werden innerhalb von Zeichenliteralen und Zeichenfolgenliteralen nicht verarbeitet.

In dem Beispiel

/\* Hello, world program  
 This program writes “hello, world” to the console  
\*/  
class Hello  
{  
 static void Main() {  
 System.Console.WriteLine("hello, world");  
 }  
}

enthält einen Kommentar mit Trennzeichen.

In dem Beispiel

// Hello, world program  
// This program writes “hello, world” to the console  
//  
class Hello // any name will do for this class  
{  
 static void Main() { // this method must be named "Main"  
 System.Console.WriteLine("hello, world");  
 }  
}

enthält mehrere einzeilige Kommentare.

### Leerraum

Der Leerraum ist als ein beliebiges Zeichen mit Unicode-Klasse Zs (welches das Leerzeichen einschließt), das horizontale und vertikale Tabulatorzeichen sowie das Seitenvorschubzeichen definiert.

whitespace:  
Any character with Unicode class Zs  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)

## Token

Es gibt mehrere Arten von Token: Bezeichner, Schlüsselwörter, Literale, Operatoren und Markierungszeichen. Leerräume und Kommentare sind keine Token, obwohl sie als Trennzeichen für Token verwendet werden können.

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

### Escapesequenzen für Unicode-Zeichen

Eine Escapesequenz für Unicode-Zeichen stellt ein Unicode-Zeichen dar. Escapesequenzen für Unicode-Zeichen werden in Bezeichnern (§2.4.2), Zeichenliteralen (§2.4.4.4) und regulären Zeichenfolgenliteralen (§2.4.4.5) verarbeitet. An anderen Stellen wird eine Escapesequenz für Unicode-Zeichen nicht verarbeitet (z. B. bei der Bildung von Operatoren, Markierungszeichen oder Schlüsselwörtern).

unicode-escape-sequence:  
\u hex-digit hex-digit hex-digit hex-digit  
\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

Eine Unicode-Escapesequenz repräsentiert das einzelne aus einer Hexadezimalzahl gebildete Unicode-Zeichen, das auf das Zeichen "\u" oder "\U" folgt. Da C# eine 16-Bit-Codierung von Unicode-Zeichen in Zeichen und Zeichenfolgenwerten verwendet, ist ein Unicode-Zeichen im Bereich zwischen U+10000 und U+10FFFF in einem Zeichenliteral nicht zulässig und wird deshalb mithilfe von zwei Unicode-Ersatzzeichen in einem Zeichenfolgenliteral dargestellt. Unicode-Zeichen mit Codepunkten oberhalb von 0x10FFFF werden nicht unterstützt.

Mehrfache Übersetzungen werden nicht ausgeführt. Das Zeichenfolgenliteral "\u005Cu005C" entspricht beispielsweise "\u005C" und nicht "\". Der Unicode-Wert \u005C steht für das Zeichen "\".

In dem Beispiel

class Class1  
{  
 static void Test(bool \u0066) {  
 char c = '\u0066';  
 if (\u0066)  
 System.Console.WriteLine(c.ToString());  
 }   
}

zeigt mehrere Verwendungsmöglichkeiten von \u0066, der Escapesequenz für den Buchstaben "f". Das Programm ist gleichwertig mit

class Class1  
{  
 static void Test(bool f) {  
 char c = 'f';  
 if (f)  
 System.Console.WriteLine(c.ToString());  
 }   
}

### Bezeichner

Die in diesem Abschnitt angegebenen Regeln für Bezeichner entsprechen genau den vom Unicode-Standard, Annex 31, empfohlenen Regeln mit der Ausnahme, dass der Unterstrich als Anfangszeichen (wie in der C-Programmiersprache), Unicode-Escapesequenzen in Bezeichnern und das @-Zeichen als Präfix zulässig sind, um die Verwendung von Schlüsselwörtern als Bezeichner zu ermöglichen.

identifier:  
available-identifier  
@ identifier-or-keyword

available-identifier:  
An identifier-or-keyword that is not a keyword

identifier-or-keyword:  
identifier-start-character identifier-part-charactersopt

identifier-start-character:  
letter-character  
\_ (the underscore character U+005F)

identifier-part-characters:  
identifier-part-character  
identifier-part-characters identifier-part-character

identifier-part-character:  
letter-character  
decimal-digit-character  
connecting-character  
combining-character  
formatting-character

letter-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl   
A unicode-escape-sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:  
A Unicode character of classes Mn or Mc   
A unicode-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:  
A Unicode character of the class Nd   
A unicode-escape-sequence representing a character of the class Nd

connecting-character:   
A Unicode character of the class Pc  
A unicode-escape-sequence representing a character of the class Pc

formatting-character:   
A Unicode character of the class Cf  
A unicode-escape-sequence representing a character of the class Cf

Informationen zu den genannten Unicode-Zeichenklassen finden Sie in The Unicode Standard, Version 3.0, Abschnitt 4.5.

Beispiele für gültige Bezeichner sind "identifier1", "\_identifier2" und "@if".

Bezeichner in einem konformen Programm müssen das kanonische Format aufweisen, das durch Unicode Normalization Form C definiert und im Unicode Standard Annex 15 festgelegt ist. Das Verhalten bei Auffinden eines nicht in Normalization Form C definierten Bezeichners ist implementierungsabhängig; eine Diagnose ist jedoch nicht erforderlich.

Das Präfix "@" ermöglicht die Verwendung von Schlüsselwörtern als Bezeichner, was für die Implementierung von Schnittstellen zu anderen Programmiersprachen nützlich sein kann. Das @-Zeichen ist kein eigentlicher Bestandteil des Bezeichners, sodass der Bezeichner in anderen Sprachen ohne das Präfix als normaler Bezeichner verwendet werden kann. Bezeichner mit einem @-Präfix werden wörtliche Bezeichner genannt. Die Verwendung des @-Präfixes für Bezeichner, die keine Schlüsselwörter sind, ist zwar zulässig, aus stilistischen Gründen wird jedoch dringend davon abgeraten.

Beispiel:

class @class  
{  
 public static void @static(bool @bool) {  
 if (@bool)  
 System.Console.WriteLine("true");  
 else  
 System.Console.WriteLine("false");  
 }   
}

class Class1  
{  
 static void M() {  
 cl\u0061ss.st\u0061tic(true);  
 }  
}

wird die Klasse "class" mit der statischen Methode "static" definiert, die den Parameter "bool" enthält. Beachten Sie, dass Escapesequenzen für Unicode-Zeichen in Schlüsselwörtern nicht zulässig sind. Das Token "cl\u0061ss" ist deshalb ein Bezeichner und mit dem Bezeichner "@class" identisch.

Zwei Bezeichner werden als identisch betrachtet, wenn sie nach Anwendung der folgenden Transformationen (in der angegebenen Reihenfolge) übereinstimmen:

* Das Präfix "@" wird entfernt (sofern verwendet).
* Jede unicode-escape-sequence wird in das entsprechende Unicode-Zeichen umgewandelt
* Alle formatting-characters werden entfernt.

Bezeichner, die zwei aufeinander folgende Unterstriche enthalten (U+005F), sind für die Verwendung durch die Implementierung reserviert. Eine Implementierung kann beispielsweise erweiterte Schlüsselwörter bereitstellen, die mit zwei Unterstrichen beginnen.

### Schlüsselwörter

Ein Schlüsselwort ist eine bezeichnerähnliche Folge von Zeichen, die reserviert ist und deshalb nur mit dem @-Präfix als Bezeichner verwendet werden kann.

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

An einigen Stellen in der Grammatik besitzen bestimmte Bezeichner eine besondere Bedeutung, sind jedoch keine Schlüsselwörter. Solche Bezeichner werden manchmal als „Kontextschlüsselwörter“ bezeichnet. Innerhalb einer Eigenschaftendeklaration haben der "get"-Bezeichner und der "set"-Bezeichner beispielsweise eine besondere Bedeutung (§10.7.2). An dieser Position sind andere Bezeichner als get oder set nicht zulässig, sodass diese Verwendung nicht mit der Verwendung der Wörter als Bezeichner kollidiert. In anderen Fällen, wie etwa beim Bezeichner "var" in implizit typisierten Variablendeklarationen (§8.5.1), kann ein Kontextschlüsselwort mit deklarierten Namen in Konflikt stehen. In solchen Fällen hat der deklarierte Name Vorrang vor der Verwendung des Bezeichners als Kontextschlüsselwort.

### Literale

Literale sind die Quellcodedarstellung eines Werts.

literal:  
boolean-literal  
integer-literal  
real-literal  
character-literal  
string-literal  
null-literal

#### Boolesche Literale

Es gibt zwei boolesche Literalwerte: true und false.

boolean-literal:  
true  
false

Der Typ eines boolean-literal ist bool.

#### Ganzzahlliterale

Ganzzahlliterale werden zum Schreiben von Werten vom Typ int, uint, long und ulong verwendet. Ganzzahlliterale können zwei Formen haben: dezimal und hexadezimal.

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits integer-type-suffixopt

decimal-digits:  
decimal-digit  
decimal-digits decimal-digit

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of  
U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:  
0x hex-digits integer-type-suffixopt  
0X hex-digits integer-type-suffixopt

hex-digits:  
hex-digit  
hex-digits hex-digit

hex-digit: one of  
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

Der Typ eines Ganzzahlliterals wird folgendermaßen bestimmt:

* Wenn das Literal kein Suffix besitzt, ist es vom ersten der folgenden Typen, der den Literalwert darstellen kann: int, uint, long, ulong.
* Wenn das Literal das Suffix U oder u besitzt, ist es vom ersten der folgenden Typen, der den Literalwert darstellen kann: uint, ulong.
* Wenn das Literal das Suffix L oder l besitzt, ist es vom ersten der folgenden Typen, der den Literalwert darstellen kann: long, ulong.
* Wenn das Literal das Suffix UL, Ul, uL, ul, LU, Lu, lU oder lu, besitzt, ist es vom Typ ulong.

Wenn der durch ein Ganzzahlliteral dargestellte Wert außerhalb des Bereichs des Typs ulong liegt, tritt ein Kompilierungsfehler auf.

Aus stilistischen Gründen ist bei Literalen vom Typ long "L" der Vorrang vor "l" zu geben, da der Buchstabe "l" leicht mit der Ziffer "1" verwechselt werden kann.

Damit die kleinstmöglichen Werte für int und long als dezimale Ganzzahlliterale geschrieben werden können, wurden die beiden folgenden Regeln festgelegt:

* Wenn ein decimal-integer-literal mit dem Wert 2147483648 (231) und ohne integer-type-suffix als Token unmittelbar nach einem unären Minus-Operatortoken (§7.7.2) auftritt, ist das Ergebnis eine Konstante vom Typ int mit dem Wert −2147483648 (−231). In allen anderen Fällen ist ein solches decimal-integer-literal vom Typ uint.
* Wenn ein decimal-integer-literal mit dem Wert 9223372036854775808 (263) ohne ein integer-type-suffix oder mit dem integer-type-suffix L oder l als Token unmittelbar nach einem unären Minus-Operatortoken (§7.7.2) auftritt, ist das Ergebnis eine Konstante vom Typ long mit dem Wert −9223372036854775808 (−263). In allen anderen Fällen ist ein solches decimal-integer-literal vom Typ ulong.

#### Reelle Literale

Reelle Literale werden für Werte der Typen float, double und decimal verwendet.

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt decimal-digits

sign: one of  
+ -

real-type-suffix: one of  
F f D d M m

Wenn kein real-type-suffix angegeben ist, ist der Typ des reellen Literals double. Andernfalls bestimmt das Reelle Zahl-Typsuffix den Typ des reellen Literals folgendermaßen:

* Ein reelles Literal mit dem Suffix F oder f ist vom Typ float. Die Literale 1f, 1.5f, 1e10f und 123.456F sind beispielsweise alle vom Typ float.
* Ein reelles Literal mit dem Suffix D oder d ist vom Typ double. Die Literale 1d, 1.5d, 1e10d und 123.456D sind beispielsweise alle vom Typ double.
* Ein reelles Literal mit dem Suffix M oder m ist vom Typ decimal. Die Literale 1m, 1.5m, 1e10m und 123.456M sind beispielsweise alle vom Typ decimal. Dieses Literal wird in einen decimal-Wert konvertiert, indem der genaue Wert ggf. entsprechend der kaufmännischen Rundung auf den nächsten darstellbaren Wert gerundet wird (§4.1.7). Jede Dezimalstelle aus dem Literal wird beibehalten, es sei denn, der Wert wird gerundet oder ist gleich 0 (das Vorzeichen und die Dezimalstellenanzahl sind im letzteren Fall 0). Folglich wird das Literal 2.900m analysiert, damit die Dezimalzahl mit dem Vorzeichen 0, dem Koeffizienten 2900 und 3 Dezimalstellen gebildet wird.

Wenn das angegebene Literal mit keinem der angegebenen Typen dargestellt werden kann, tritt bei der Kompilierung ein Fehler auf.

Der Wert eines reellen Literals vom Typ float oder double wird durch Rundung nach IEEE-Norm bestimmt.

Beachten Sie, dass bei reellen Literalen nach dem Dezimalzeichen stets Dezimalstellen angegeben werden müssen. Beispielsweise ist 1.3F ein reelles Literal, 1.F hingegen nicht.

#### Zeichenliterale

Ein Zeichenliteral stellt ein einzelnes Zeichen dar und besteht in der Regel aus einem Zeichen in Anführungszeichen, z. B. 'a'.

character-literal:  
' character '

character:  
single-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-character:  
 Any character except ' (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of  
\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:  
\x hex-digit hex-digitopt hex-digitopt hex-digitopt

Bei dem Zeichen nach einem umgekehrten Schrägstrich (\) in einem character muss es sich um eines der folgenden Zeichen handeln: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Andernfalls wird ein Kompilierungsfehler ausgelöst.

Eine hexadezimale Escapesequenz stellt ein einzelnes Unicode-Zeichen dar, wobei der Wert aus "\x" gefolgt von der hexadezimalen Zahl gebildet wird.

Wenn der von einem Zeichenliteral dargestellte Wert größer als U+FFFF ist, tritt ein Kompilierungsfehler auf.

Eine Escapesequenz für Unicode-Zeichen (§2.4.1) in einem Zeichenliteral muss im Bereich zwischen U+0000 und U+FFFF liegen.

Eine einfache Escapesequenz stellt eine Unicode-Zeichencodierung dar (siehe folgende Tabelle).

|  |  |  |
| --- | --- | --- |
| **Escapesequenz** | **Zeichenname** | **Unicode-Codierung** |
| \' | Einfaches Anführungszeichen | 0x0027 |
| \" | Doppeltes Anführungszeichen | 0x0022 |
| \\ | Umgekehrter Schrägstrich | 0x005C |
| \0 | Null | 0x0000 |
| \a | Warnung | 0x0007 |
| \b | Rückschritt | 0x0008 |
| \f | Seitenvorschub | 0x000C |
| \n | Zeilenwechsel | 0x000A |
| \r | Wagenrücklauf | 0x000D |
| \t | Horizontaler Tabulator | 0x0009 |
| \v | Vertikaler Tabulator | 0x000B |

Der Typ eines character-literal ist char.

#### Zeichenfolgenliterale

C# unterstützt zwei Arten von Zeichenfolgenliteralen: reguläre Zeichenfolgenliterale und wörtliche Zeichenfolgenliterale.

Ein reguläres Zeichenfolgenliteral besteht aus null oder mehr Zeichen, eingeschlossen in doppelte Anführungszeichen (z. B. "hello"), und kann einfache Escapesequenzen (z. B. \t für das Tabstoppzeichen), hexadezimale Escapesequenzen und Escapesequenzen für Unicode-Zeichen enthalten.

Ein wörtliches Zeichenfolgenliteral besteht aus einem @-Zeichen, gefolgt von einem doppelten Anführungszeichen, null oder mehr Zeichen und einem schließenden doppelten Anführungszeichen. Ein einfaches Beispiel ist @"hello". Bei einem wörtlichen Zeichenfolgenliteral werden die Zeichen zwischen den Begrenzungszeichen wörtlich interpretiert. Die einzige Ausnahme bildet eine quote-escape-sequence. Genauer gesagt, werden einfache Escapesequenzen, hexadezimale Escapesequenzen und Escapesequenzen für Unicode-Zeichen in wörtlichen Zeichenfolgenliteralen nicht verarbeitet. Ein wörtliches Zeichenfolgenliteral kann sich über mehrere Zeilen erstrecken.

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

regular-string-literal-characters:  
regular-string-literal-character  
regular-string-literal-characters regular-string-literal-character

regular-string-literal-character:  
single-regular-string-literal-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-regular-string-literal-character:  
Any character except " (U+0022), \ (U+005C), and new-line-character

verbatim-string-literal:  
@" verbatim-string-literal-charactersopt "

verbatim-string-literal-characters:  
verbatim-string-literal-character  
verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:  
single-verbatim-string-literal-character  
quote-escape-sequence

single-verbatim-string-literal-character:  
Any character except "

quote-escape-sequence:  
""

Bei dem Zeichen nach einem umgekehrten Schrägstrich (\) in einem regular-string-literal-character muss es sich um eines der folgenden Zeichen handeln: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Andernfalls wird ein Kompilierungsfehler ausgelöst.

In dem Beispiel

string a = "hello, world"; // hello, world  
string b = @"hello, world"; // hello, world

string c = "hello \t world"; // hello world  
string d = @"hello \t world"; // hello \t world

string e = "Joe said \"Hello\" to me"; // Joe said "Hello" to me  
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt"; // \\server\share\file.txt  
string h = @"\\server\share\file.txt"; // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";  
string j = @"one  
two  
three";

zeigt eine Vielzahl von Zeichenfolgenliteralen. Das letzte Zeichenfolgenliteral j ist ein wörtliches Zeichenfolgenliteral, das sich über mehrere Zeilen erstreckt. Die Zeichen zwischen den Anführungszeichen, einschließlich der Leerräume (z. B. Zeilenumbruchzeichen), werden wörtlich beibehalten.

Da eine hexadezimale Escapesequenz eine variable Anzahl von Hexadezimalziffern aufweisen kann, enthält das Zeichenfolgenliteral "\x123" ein einzelnes Zeichen mit dem Hexadezimalwert 123. Um eine Zeichenfolge zu erstellen, die das Zeichen mit dem Hexadezimalwert 12, gefolgt vom Zeichen 3 enthält, könnte stattdessen "\x00123" oder "\x12" + "3" verwendet werden.

Der Typ eines string-literal ist string.

Nicht jedes Zeichenfolgenliteral ergibt notwendigerweise eine neue Zeichenfolgeninstanz. Wenn zwei oder mehr Zeichenfolgenliterale, die entsprechend dem Gleichheitsoperator für Zeichenfolgen (§7.10.7) identisch sind, in demselben Programm auftreten, verweisen diese Zeichenfolgenliterale auf dieselbe Zeichenfolgeninstanz. Beispielsweise ist die Ausgabe von

class Test  
{  
 static void Main() {  
 object a = "hello";  
 object b = "hello";  
 System.Console.WriteLine(a == b);  
 }  
}

True, da sich die beiden Literale auf dieselbe Zeichenfolgeninstanz beziehen.

#### Das NULL-Literal

null-literal:  
null

Das null-literal kann implizit in einen Verweistyp oder einen Typ, der NULL-Werte zulässt, konvertiert werden.

### Operatoren und Markierungszeichen

Es gibt mehrere Arten von Operatoren und Markierungszeichen. Operatoren werden in Ausdrücken zum Beschreiben von Operationen verwendet, die einen oder mehrere Operanden einbeziehen. Der Ausdruck a + b verwendet beispielsweise den +-Operator, um die beiden Operanden a und b zu addieren. Markierungszeichen werden zum Gruppieren und Trennen verwendet.

operator-or-punctuator: one of  
{ } [ ] ( ) . , : ;  
+ - \* / % & | ^ ! ~  
= < > ? ?? :: ++ -- && ||  
-> == != <= >= += -= \*= /= %=  
&= |= ^= << <<= =>

right-shift:  
>|>

right-shift-assignment:  
>|>=

Der senkrechte Strich in der right-shift-Produktion und der right-shift-assignment-Produktion gibt an, dass im Gegensatz zu anderen Produktionen in der syntaktischen Grammatik zwischen den Token in der right-shift-Produktion und der right-shift-assignment-Produktion keinerlei Zeichen (auch kein Leerraum) erlaubt sind. Diese spezielle Eigenschaft dieser Produktionen ermöglicht die ordnungsgemäße Behandlung von type-parameter-lists (§10.1.3).

## Präprozessordirektiven

Die Präprozessordirektive ermöglicht das bedingte Überspringen von Abschnitten der Quelldateien, das Erfassen von Fehler- und Warnungsbedingungen und das Entwerfen von bestimmten Regionen des Quellcodes. Der Ausdruck „Präprozessordirektive“ wird nur aus Gründen der Konsistenz mit den Programmiersprachen C und C++ verwendet. In C# gibt es keinen separaten Präprozessorschritt. Präprozessordirektiven werden während der lexikalischen Analyse verarbeitet.

pp-directive:  
pp-declaration  
pp-conditional  
pp-line  
pp-diagnostic  
pp-region   
pp-pragma

Die folgenden Präprozessordirektiven sind verfügbar:

* #define und #undef, die zum Definieren bzw. zum Aufheben der Definition bedingter Kompilierungssymbole verwendet werden (§2.5.3).
* #if, #elif, #else und #endif, die zum bedingten Überspringen von Quellcodeabschnitten verwendet werden (§2.5.4).
* #line, das zum Steuern der Zeilennummern für die Ausgabe von Fehlern und Warnungen verwendet wird (§2.5.7).
* #error und #warning, die für die Ausgabe von Fehlern bzw. Warnungen verwendet werden (§2.5.5).
* #region und #endregion, die zum expliziten Markieren von Quellcodeabschnitten verwendet werden (§2.5.6).
* #pragma, das zum Angeben optionaler Kontextinformationen für den Compiler verwendet wird (§2.5.8).

Eine Präprozessordirektive nimmt eine immer eine eigene Quellcodezeile ein und beginnt immer mit einem #-Zeichen und dem Namen einer Präprozessordirektiven. Leerraum ist vor dem #-Zeichen sowie zwischen dem #-Zeichen und dem Direktivennamen zulässig.

Eine Quellzeile, die eine der Direktiven #define, #undef, #if, #elif, #else, #endif, #line oder #endregion enthält, kann mit einem einzeiligen Kommentar abgeschlossen werden. Kommentare mit Trennzeichen (Kommentare im Stil /\* \*/) sind in Quellzeilen mit Präprozessordirektiven nicht zulässig.

Präprozessordirektiven sind weder Token noch Teil der syntaktischen Grammatik von C#. Sie können jedoch für den Ein- oder Ausschluss von Tokenabfolgen verwendet werden und haben auf diese Weise auch Auswirkungen auf die Bedeutung eines C#-Programms. Beispiel: Das folgende Programm

#define A  
#undef B

class C  
{  
#if A  
 void F() {}  
#else  
 void G() {}  
#endif

#if B  
 void H() {}  
#else  
 void I() {}  
#endif  
}

erzeugt beim Kompilieren dieselbe Abfolge von Token wie das folgende Programm:

class C  
{  
 void F() {}  
 void I() {}  
}

Die beiden Programme sind zwar lexikalisch sehr unterschiedlich, syntaktisch jedoch identisch.

### Symbole für bedingte Kompilierung

Die von den Direktiven #if, #elif, #else und #endif unterstützten Funktionen für eine bedingte Kompilierung wird mit Präprozessorausdrücken (§2.5.2) und Symbolen für bedingte Kompilierung gesteuert.

conditional-symbol:  
Any identifier-or-keyword except true or false

Ein Symbol für bedingte Kompilierung kann einen von zwei Zuständen aufweisen: definiert oder nicht definiert. Am Anfang der lexikalischen Verarbeitung einer Quelldatei ist das bedingte Kompilierungssymbol nicht definiert, außer es wurde durch einen externen Mechanismus (z. B. durch eine Option des Befehlszeilencompilers) explizit definiert. Bei der Verarbeitung einer #define-Direktive wird das in der Direktive genannte Symbol für bedingte Kompilierung für diese Quelldatei definiert. Die Symboldefinition bleibt so lange gültig, bis eine #undef-Direktive für dasselbe Symbol verarbeitet oder das Ende der Quelldatei erreicht wird. Daraus folgt, dass die Direktiven #define und #undef in einer Quelldatei keine Auswirkung auf die anderen Quelldateien im gleichen Programm haben.

Ein definiertes Symbol für bedingte Kompilierung, auf das in einem Präprozessorausdruck verwiesen wird, hat den booleschen Wert true, und ein nicht definiertes Symbol für bedingte Kompilierung hat den booleschen Wert false. Es ist nicht erforderlich, bedingte Kompilierungssymbole explizit zu deklarieren, bevor in einem Präprozessorausdruck auf sie verwiesen wird. Nicht deklarierte Symbole gelten vielmehr als nicht definiert und besitzen somit den Wert false.

Der Namespace für bedingte Kompilierungssymbole unterscheidet sich von allen anderen benannten Entitäten in einem C#-Programm und ist von diesen getrennt. Auf Symbole für bedingte Kompilierung kann nur in den Direktiven #define und #undef sowie in Präprozessorausdrücken verwiesen werden.

### Präprozessorausdrücke

Präprozessorausdrücke können in den Direktiven #if und #elif auftreten. Die Operatoren !, ==, !=, && und || sind in Präprozessorausdrücken zulässig. Zum Gruppieren können Klammern verwendet werden.

pp-expression:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

pp-unary-expression:  
pp-primary-expression  
! whitespaceopt pp-unary-expression

pp-primary-expression:  
true  
false  
conditional-symbol  
( whitespaceopt pp-expression whitespaceopt )

Ein definiertes Symbol für bedingte Kompilierung, auf das in einem Präprozessorausdruck verwiesen wird, hat den booleschen Wert true, und ein nicht definiertes Symbol für bedingte Kompilierung hat den booleschen Wert false.

Die Auswertung eines Präprozessorausdrucks ergibt immer einen booleschen Wert. Die Regeln für die Auswertung eines Präprozessorausdrucks stimmen mit denen für einen konstanten Ausdruck (§7.19) überein, außer dass die einzigen benutzerdefinierten Entitäten, auf die verwiesen werden kann, Symbole für bedingte Kompilierung sind.

### Deklarationsdirektiven

Die Deklarationsdirektiven werden zum Definieren von bedingten Kompilierungssymbolen bzw. zum Aufheben der Definition verwendet.

pp-declaration:  
whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line  
whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line:  
whitespaceopt single-line-commentopt new-line

Die Verarbeitung einer #define-Direktive bewirkt, dass das angegebene Symbol für bedingte Kompilierung ab der Quellzeile definiert ist, die auf die Direktive folgt. Entsprechend bewirkt die Verarbeitung einer #undef-Direktive, dass die Definition des angegebenen bedingten Kompilierungssymbols ab der Quellzeile aufgehoben ist, die auf die Direktive folgt.

Alle #define-Direktiven und #undef-Direktiven in einer Quelldatei müssen vor dem ersten token (§2.4) auftreten, da andernfalls während der Kompilierung ein Fehler auftritt. Einfach ausgedrückt, müssen die #define-Direktive und die #undef-Direktive in der Quelldatei vor dem eigentlichen Code stehen.

Beispiel:

#define Enterprise

#if Professional || Enterprise  
 #define Advanced  
#endif

namespace Megacorp.Data  
{  
 #if Advanced  
 class PivotTable {...}  
 #endif  
}

ist gültig, da die #define-Direktiven in der Quelldatei vor dem ersten Token (dem namespace-Schlüsselwort) stehen.

Das folgende Beispiel führt zu einem Kompilierungsfehler, weil nach dem eigentlichen Code eine #define-Direktive steht:

#define A  
namespace N  
{  
 #define B  
 #if B  
 class Class1 {}  
 #endif  
}

Mit #define kann ein bereits definiertes Symbol für bedingte Kompilierung definiert werden, ohne dass zuvor ein #undef für das Symbol angegeben wird. Im folgenden Beispiel wird ein bedingtes Kompilierungssymbol A definiert, das anschließend erneut definiert wird.

#define A  
#define A

#undef kann auch die Definition eines nicht definierten Symbols für bedingte Kompilierung aufheben. Im folgenden Beispiel wird das Symbol für bedingte Kompilierung A definiert, und die Definition wird anschließend zweimal aufgehoben. Das zweite #undef hat zwar keine Auswirkung, ist aber zulässig.

#define A  
#undef A  
#undef A

### Direktiven für bedingte Kompilierung

Die bedingten Kompilierungsdirektiven werden zum bedingten Ein- bzw. Ausschließen von Quelldateiabschnitten verwendet.

pp-conditional:  
pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif

pp-if-section:  
whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-sectionopt

pp-elif-sections:  
pp-elif-section  
pp-elif-sections pp-elif-section

pp-elif-section:  
whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-sectionopt

pp-else-section:  
whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

pp-endif:  
whitespaceopt # whitespaceopt endif pp-new-line

conditional-section:  
input-section  
skipped-section

skipped-section:  
skipped-section-part  
skipped-section skipped-section-part

skipped-section-part:  
skipped-charactersopt new-line  
pp-directive

skipped-characters:  
whitespaceopt not-number-sign input-charactersopt

not-number-sign:  
Any input-character except #

Direktiven für bedingte Kompilierung müssen entsprechend der angegebenen Syntax als Gruppen geschrieben werden, die aus einer #if-Direktive, null oder mehr #elif-Direktiven, null oder einer #else-Direktive und einer #endif-Direktive in dieser Reihenfolge bestehen. Zwischen den Direktiven befinden sich bedingte Quellcodeabschnitte. Jeder Abschnitt wird dabei von der unmittelbar vorhergehenden Direktive gesteuert. Ein bedingter Abschnitt kann geschachtelte bedingte Kompilierungsdirektiven enthalten, vorausgesetzt diese Direktiven bilden vollständige Sets.

Ein pp-conditional wählt höchstens eine der enthaltenen conditional-sections für die reguläre lexikalische Verarbeitung aus:

* Die pp-expressions der #if-Direktive und der #elif-Direktive werden in der angegebenen Reihenfolge ausgewertet, bis eine den Wert true zurückgibt. Wenn ein Ausdruck den Wert true zurückgibt, wird die conditional-section der entsprechenden Direktive ausgewählt.
* Wenn alle pp-expressions false zurückgeben und eine #else-Direktive vorhanden ist, wird die conditional-section der #else-Direktive ausgewählt.
* Andernfalls wird kein conditional-section ausgewählt.

Die ausgewählte conditional-section (sofern eine Auswahl getroffen wurde) wird als normale input-section verarbeitet: Der im Abschnitt enthaltene Quellcode muss die Vorgaben der lexikalischen Grammatik einhalten. Aus dem Quellcode des Abschnitts werden Token erstellt. Die Präprozessordirektiven im Abschnitt haben die vorgeschriebenen Auswirkungen.

Die verbleibenden conditional-sections (sofern eine Auswahl getroffen wurde) werden als skipped-sections verarbeitet: Abgesehen von den Präprozessordirektiven muss der in diesen Abschnitten enthaltene Quellcode nicht die Vorgaben der lexikalischen Grammatik einhalten. Aus dem Quellcode des Abschnitts werden keine Token erstellt. Die Präprozessordirektiven in dem Abschnitt müssen lexikalisch korrekt sein, werden jedoch nicht auf eine andere Weise verarbeitet. Innerhalb einer als skipped-section verarbeiteten conditional-section werden alle geschachtelten conditional-sections (die in geschachtelten #if...#endif-Konstrukten und geschachtelten #region...#endregion-Konstrukten enthalten sind) ebenfalls als skipped-sections verarbeitet.

Das folgende Beispiel demonstriert, wie bedingte Kompilierungsdirektiven geschachtelt werden können:

#define Debug // Debugging on  
#undef Trace // Tracing off

class PurchaseTransaction  
{  
 void Commit() {  
 #if Debug  
 CheckConsistency();  
 #if Trace  
 WriteToLog(this.ToString());  
 #endif  
 #endif  
 CommitHelper();  
 }  
}

Abgesehen von den Präprozessordirektiven wird der übersprungene Quellcode keiner lexikalischen Analyse unterzogen. Daher ist das folgende Beispiel trotz des nicht abgeschlossenen Kommentars im #else-Abschnitt gültig:

#define Debug // Debugging on

class PurchaseTransaction  
{  
 void Commit() {  
 #if Debug  
 CheckConsistency();  
 #else  
 /\* Do something else  
 #endif  
 }  
}

Beachten Sie jedoch, dass die Präprozessordirektiven lexikalisch korrekt sein müssen, auch wenn sie sich im übersprungenen Abschnitt des Quellcodes befinden.

Innerhalb von mehrzeiligen Eingabeelementen werden Präprozessordirektiven nicht verarbeitet. Beispiel: Das Programm

class Hello  
{  
 static void Main() {  
 System.Console.WriteLine(@"hello,   
#if Debug  
 world  
#else  
 Nebraska  
#endif  
 ");  
 }  
}

erzeugt die folgende Ausgabe:

hello,  
#if Debug  
 world  
#else  
 Nebraska  
#endif

In speziellen Fällen kann die Gruppe der verarbeiteten Präprozessordirektiven von der Auswertung von pp-expression abhängen. Beispiel:

#if X  
 /\*  
#else  
 /\* \*/ class Q { }  
#endif

erzeugt immer denselben Tokenstream (class Q { }), unabhängig davon, ob X definiert ist. Aufgrund des mehrzeiligen Kommentars werden nur die Direktiven #if und #endif verarbeitet, wenn X definiert ist. Wenn X nicht definiert ist, sind drei Direktiven (#if, #else, #endif) Teil der Direktivengruppe.

### Diagnosedirektiven

Die Diagnosedirektiven werden zum expliziten Erstellen von Fehler- und Warnmeldungen verwendet, die bei der Kompilierung auf dieselbe Art angezeigt werden, wie andere Fehler und Warnungen.

pp-diagnostic:  
whitespaceopt # whitespaceopt error pp-message  
whitespaceopt # whitespaceopt warning pp-message

pp-message:  
new-line  
whitespace input-charactersopt new-line

Beispiel:

#warning Code review needed before check-in

#if Debug && Retail  
 #error A build can't be both debug and retail  
#endif

class Test {...}

erzeugt immer eine Warnung, die darauf hinweist, dass vor dem Einchecken eine Codeüberprüfung erforderlich ist, und einen Kompilierungsfehler, der darauf hinweist, dass ein Build nicht gleichzeitig Debug- und Verkaufsversion sein kann, wenn sowohl das Symbol Debug als auch das Symbol Retail definiert ist. Beachten Sie, dass eine pp-message-message beliebigen Text enthalten kann. Dies bedeutet, dass sie keine regelkonformen Token enthalten muss, wie an dem Apostroph im Wort can’t veranschaulicht wird.

### Region-Direktiven

Die Regionsdirektiven werden zum expliziten Markieren von Quellcodebereichen verwendet.

pp-region:  
pp-start-region conditional-sectionopt pp-end-region

pp-start-region:  
whitespaceopt # whitespaceopt region pp-message

pp-end-region:  
whitespaceopt # whitespaceopt endregion pp-message

Mit einer Region ist keine semantische Bedeutung verknüpft. Sie ist lediglich dafür gedacht, dass Programmierer oder automatisierte Tools einen Bereich des Quellcodes markieren können. Die in der #region-Direktive bzw. in der #endregion-Direktive angegebenen Meldungen haben ebenfalls keine semantische Bedeutung. Ihr Zweck besteht lediglich in der Kennzeichnung des Bereichs. Zusammengehörige #region-Direktiven und #endregion-Direktiven können unterschiedliche pp-messages besitzen.

Die lexikalische Verarbeitung einer Region:

#region  
...  
#endregion

entspricht der lexikalischen Verarbeitung einer bedingten Kompilierungsdirektive der Form:

#if true  
...  
#endif

### Line-Direktiven

Zeilendirektiven können zum Ändern der Zeilenanzahl und der Quelldateinamen verwendet werden, die der Compiler in Form von Warnungen und Fehlermeldungen ausgibt und die von Infoattributen des Aufrufers verwendet werden (§17.4.4).

Zeilendirektiven werden sehr häufig in Metaprogrammierungstools verwendet, die C#-Quellcode aus einer anderen Texteingabe erzeugen.

pp-line:  
whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator:  
decimal-digits whitespace file-name   
decimal-digits  
default  
hidden

file-name:  
" file-name-characters "

file-name-characters:  
file-name-character  
file-name-characters file-name-character

file-name-character:  
Any input-character except "

Wenn keine #line-Direktiven vorhanden sind, werden in der Compilerausgabe die tatsächlichen Zeilennummern und Quelldateinamen angegeben. Wenn eine #line-Direktive mit einem line-indicator verarbeitet wird, der nicht default ist, behandelt der Compiler die Zeile nach der Direktive, als ob diese die angegebene Zeilennummer (und ggf. den angegeben Dateinamen) hätte.

Eine #line default-Direktive hebt die Wirkung aller vorherigen #line-Direktiven auf. Der Compiler meldet für alle folgenden Zeilen die tatsächlichen Zeilennummern, als ob keine #line-Direktiven verarbeitet worden wären.

Eine #line hidden-Direktive hat zwar keine Auswirkung auf die Datei und die in Fehlermeldungen angegebenen Zeilennummern, wohl aber auf das Debuggen auf Quellebene. Beim Debuggen sind für alle Zeilen zwischen einer #line hidden-Direktive und der folgenden #line-Direktive (sofern es sich nicht um die #line hidden-Direktive handelt) keine Zeilennummern verfügbar. Beim schrittweisen Ausführen des Codes im Debugger werden diese Zeilen gänzlich übersprungen.

Beachten Sie, dass sich der file-name von einem regulären Zeichenfolgenliteral darin unterscheidet, dass Escapesequenzen nicht verarbeitet werden. Das \-Zeichen bezeichnet in einem file-name lediglich einen normalen umgekehrten Schrägstrich.

### pragma-Direktiven

Mithilfe der #pragma-Vorverarbeitungsdirektive werden für den Compiler optionale Kontextinformationen angegeben. Die in einer #pragma-Direktive angegebenen Informationen ändern nie die Programmsemantik.

pp-pragma:  
whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line

pragma-body:  
pragma-warning-body

In C# werden #pragma-Direktiven zum Steuern von Compilerwarnungen bereitgestellt. Künftige Versionen der Sprache können weitere #pragma-Direktiven enthalten. Um die Interoperabilität mit anderen C#-Compilern zu gewährleisten, werden vom Microsoft C#-Compiler bei unbekannten #pragma-Direktiven keine Kompilierungsfehler ausgegeben. Bei solchen Direktiven werden jedoch Warnungen generiert.

#### pragma-Warnung

Mit der #pragma warning-Direktive werden während der Kompilierung des nachfolgenden Programmtexts alle oder ausgewählte Warnmeldungen deaktiviert bzw. wiederhergestellt.

pragma-warning-body:  
warning whitespace warning-action  
warning whitespace warning-action whitespace warning-list

warning-action:  
disable  
restore

warning-list:  
decimal-digits  
warning-list whitespaceopt , whitespaceopt decimal-digits

Eine #pragma warning-Direktive, in der die Liste der Warnungen ausgelassen wird, wirkt sich auf alle Warnungen aus. Eine #pragma warning-Direktive, die eine Liste von Warnungen enthält, wirkt sich nur auf die Warnungen aus, die in der Liste angegeben sind.

Durch eine #pragma warning disable-Direktive werden alle oder ausgewählte Warnungen deaktiviert.

Mit einer #pragma warning restore-Direktive werden alle oder die angegebenen Warnungen in dem Zustand wiederhergestellt, der zu Beginn der Kompilationseinheit gültig war. Beachten Sie Folgendes: Wenn eine bestimmte Warnung extern deaktiviert wurde, wird die betreffende Warnung durch eine #pragma warning restore-Direktive (für alle Warnungen oder für die betreffende Warnung) nicht erneut aktiviert.

Im folgenden Beispiel wird veranschaulicht, wie mithilfe von #pragma warning und der Warnungsnummer vom Microsoft C#-Compiler die Warnung deaktiviert wird, die ausgegeben wird, wenn auf veraltete Member verwiesen wird.

using System;

class Program  
{  
 [Obsolete]  
 static void Foo() {}

static void Main() {  
#pragma warning disable 612  
 Foo();  
#pragma warning restore 612  
 }  
}

# Grundlegende Konzepte

## Anwendungsstart

Eine Assembly, die über einen Einstiegspunkt verfügt, wird als Anwendung bezeichnet. Beim Ausführen einer Anwendung wird eine neue Anwendungsdomäne erstellt. Auf einem Computer können gleichzeitig mehrere verschiedene Instanziierungen einer Anwendung vorhanden sein, von denen jede ihre eigene Anwendungsdomäne hat.

Eine Anwendungsdomäne ermöglicht die Anwendungsisolation, indem sie als ein Container für den Anwendungsstatus fungiert. Eine Anwendungsdomäne fungiert als Container und Begrenzung der in der Anwendung definierten Typen sowie die von ihr verwendeten Klassenbibliotheken. Die in eine Anwendungsdomäne geladenen Typen unterscheiden sich von den entsprechenden in eine andere Anwendungsdomäne geladenen Typen; Objektinstanzen werden zwischen Anwendungsdomänen nicht direkt freigegeben. Jede Anwendungsdomäne verfügt beispielsweise über ihre eigenen statischen Variablen für diese Typen, und ein statischer Konstruktor für einen Typ wird meistens einmal pro Anwendungsdomäne ausgeführt. Implementierungen können nach Bedarf spezifische Richtlinien oder Mechanismen für die Erstellung und Zerstörung von Anwendungsdomänen enthalten.

Der Anwendungsstart erfolgt, wenn die Ausführungsumgebung eine ausgewiesene Methode aufruft, die als Einstiegspunkt der Anwendung bezeichnet wird. Diese Einstiegspunktmethode hat immer den Namen Main und kann eine der folgenden Signaturen besitzen:

static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}

Das heißt, der Einstiegspunkt kann optional einen int-Wert zurückgeben. Dieser Rückgabewert wird bei der Anwendungsbeendigung verwendet (§3.2).

Der Einsprungpunkt kann optional über einen formalen Parameter verfügen. Der Parameter kann einen beliebigen Namen besitzen, muss jedoch vom Typ string[] sein. Wenn der formale Parameter vorhanden ist, erstellt und übergibt die Ausführungsumgebung ein string[]-Argument mit den Befehlszeilenargumenten, die beim Starten der Anwendung angegeben wurden. Das string[]-Argument ist nie NULL, kann aber, wenn keine Befehlszeilenargumente angegeben wurden, die Länge 0 (null) aufweisen.

Da C# das Überladen von Methoden unterstützt, kann eine Klasse oder Struktur mehrere Definitionen einer Methode enthalten, solange jede eine unterschiedliche Signatur besitzt. In einem einzelnen Programm sollte jedoch keine Klasse oder Struktur mehr als eine Methode mit dem Namen Main besitzen, da diese als Einstiegspunkt für die Anwendung verwendet wird. Andere überladene Versionen von Main sind zulässig, vorausgesetzt, sie haben mehrere Parameter oder der einzige Parameter ist nicht vom Typ string[].

Eine Anwendung kann aus mehreren Klassen oder Strukturen bestehen. Es ist möglich, dass zwei oder mehr dieser Klassen oder Strukturen eine Methode mit dem Namen Main enthalten, die aufgrund ihrer Definition als Einstiegspunkt für die Anwendung infrage kommen. In solchen Fällen muss ein externer Mechanismus (z. B. eine Befehlszeilencompiler-Option) verwendet werden, um eine dieser Main-Methoden als Einstiegspunkt auszuwählen.

In C# müssen alle Methoden als ein Member einer Klasse oder Struktur definiert sein. In der Regel wird der für eine Methode deklarierte Zugriff (§3.5.1) durch die in ihrer Deklaration angegebenen Zugriffsmodifizierer (§10.3.5) bestimmt. Entsprechend wird der für einen Typ deklarierte Zugriff durch die in seiner Deklaration angegebenen Zugriffsmodifizierer bestimmt. Damit die angegebene Methode eines angegebenen Typs aufgerufen werden kann, muss sowohl auf den Typ als auch auf den Member zugegriffen werden können. Der Anwendungseinsprungpunkt ist jedoch ein Sonderfall. Speziell die Ausführungsumgebung kann auf den Anwendungseinsprungpunkt zugreifen, unabhängig vom deklarierten Zugriff und unabhängig vom deklarierten Zugriff der einschließenden Typdeklarationen.

Die Methode für den Anwendungseinsprungpunkt darf nicht in einer generischen Klassendeklaration vorkommen.

In allen anderen Punkten verhalten sich Einsprungpunktmethoden wie alle anderen Methoden.

## Anwendungsbeendigung

Durch die Anwendungsbeendigung wird die Steuerung an die Ausführungsumgebung zurückgegeben.

Wenn der Rückgabetyp der Einstiegspunktmethode der Anwendung int ist, wird der zurückgegebene Wert als Beendigungszustandscode der Anwendung verwendet. Zweck dieses Codes ist es, die Ausführungsumgebung über den Erfolg oder das Fehlschlagen zu benachrichtigen.

Wenn der Rückgabetyp der Einstiegspunktmethode void ist, wird der Beendigungszustandscode 0 zurückgegeben, sobald die rechte geschweifte Klammer (}) am Ende der Einstiegspunktmethode erreicht wird oder eine return-Anweisung ohne Ausdruck ausgeführt wird.

Vor Beendigung einer Anwendung werden die Destruktoren für alle Objekte aufgerufen, für die noch keine Garbage Collection ausgeführt wurde, sofern diese Bereinigung nicht (z. B. durch einen Aufruf der Bibliotheksmethode GC.SuppressFinalize) unterdrückt wurde.

## Deklarationen

Deklarationen in einem C#-Programm definieren die konstituierenden Elemente des Programms. C#-Programme werden mithilfe von Namespaces (§9) organisiert, die Typdeklarationen und geschachtelte Namespacedeklarationen enthalten können. Typdeklarationen (§9.6) werden zum Definieren von Klassen (§10), Strukturen (§10.14), Schnittstellen (§13), Enumerationen (§14) und Delegaten (§15) verwendet. Die in einer Typdeklaration zulässigen Membertypen sind von der Form der Typdeklaration abhängig. Klassendeklarationen können beispielsweise Deklarationen für Konstanten (§10.4), Felder (§10.5), Methoden (§10.6), Eigenschaften (§10.7), Ereignisse (§10.8), Indexer (§10.9), Operatoren (§10.10), Instanzkonstruktoren (§10.11), statische Konstruktoren (§10.12), Destruktoren (§10.13) und geschachtelte Typen (§10.3.8) enthalten.

Durch eine Deklaration wird ein Name im zugehörigen Deklarationsabschnitt definiert. Außer bei überladenen Membern (§3.6) wird ein Kompilierungsfehler verursacht, wenn zwei oder mehr Deklarationen vorhanden sind, durch die gleichnamige Member im Deklarationsabschnitt eingeführt werden. Ein Deklarationsabschnitt kann nie verschiedene Arten von gleichnamigen Membern enthalten. Es ist beispielsweise nicht möglich, dass ein Deklarationsabschnitt ein Feld und eine Methode mit demselben Namen enthält.

Im Folgenden werden mehrere unterschiedliche Typen von Deklarationsabschnitten beschrieben.

* Für alle Quelldateien eines Programms gilt, dass namespace-member-declarations ohne einschließende namespace-declaration Member eines einzigen Sammeldeklarationsabschnitts sind, der als globaler Deklarationsabschnitt bezeichnet wird.
* In allen Quelldateien eines Programms sind namespace-member-declarations innerhalb von namespace-declarations, die denselben vollqualifizierten Namespacenamen besitzen, Member eines einzigen Sammeldeklarationsabschnitts.
* Alle Klassen-, Struktur- und Schnittstellendeklarationen erstellen einen neuen Deklarationsabschnitt. Namen werden in diesen Deklarationsabschnitt durch class-member-declarations, struct-member-declarations, interface-member-declarations oder type-parameters eingeführt. Mit Ausnahme von überladenen Instanzkonstruktordeklarationen und statischen Konstruktordeklarationen kann eine Klasse oder Struktur keine Memberdeklaration mit demselben Namen wie die Klasse oder Struktur einführen. Eine Klasse, Struktur oder Schnittstelle lässt die Deklaration überladener Methoden und Indexer zu. Außerdem lässt eine Klasse oder Struktur die Deklaration überladener Instanzkonstruktoren und Operatoren zu. Eine Klasse, Struktur oder Schnittstelle kann beispielsweise mehrere Methodendeklarationen mit demselben Namen enthalten, vorausgesetzt, diese Methodendeklarationen haben unterschiedliche Signaturen (§3.6). Beachten Sie, dass Basisklassen nicht zum Deklarationsabschnitt einer Klasse und Basisschnittstellen nicht zum Deklarationsabschnitt einer Schnittstelle beitragen. Demnach kann eine abgeleitete Klasse oder Schnittstelle einen Member mit demselben Namen wie einen vererbten Member deklarieren. Ein solcher Member wird vom geerbten Member verborgen.
* Alle Delegatdeklarationen erstellen einen neuen Deklarationsabschnitt. Namen werden in diesen Deklarationsabschnitt durch formale Parameter (fixed-parameters und parameter-arrays) und type-parameters eingeführt.
* Alle Enumerationsdeklarationen erstellen einen neuen Deklarationsabschnitt. Namen werden über enum-member-declarations in diesen Deklarationsabschnitt eingeführt.
* Jede Methodendeklaration, Indexerdeklaration, Operatordeklaration, Instanzkonstruktordeklaration und anonyme Funktion erstellt einen neuen Deklarationsabschnitt, der als Deklarationsabschnitt für lokale Variablen bezeichnet wird. Namen werden in diesen Deklarationsabschnitt durch formale Parameter (fixed-parameters und parameter-arrays) und type-parameters eingeführt. Der Text des Funktionsmembers oder der anonymen Funktion, sofern vorhanden, wird als im Deklarationsabschnitt für lokale Variablen geschachtelt betrachtet. Ebenso stellt es einen Fehler dar, wenn ein Deklarationsabschnitt für lokale Variablen und ein geschachtelter Deklarationsabschnitt für lokale Variablen Elemente mit dem gleichen Namen enthalten. Demnach kann innerhalb eines geschachtelten Deklarationsabschnitts keine lokale Variable deklariert werden, die den gleichen Namen wie eine lokale Variable im umschließenden Deklarationsabschnitt besitzt. Zwei Deklarationsabschnitte können Elemente mit dem gleichen Namen enthalten, sofern keiner der Deklarationsabschnitte im anderen enthalten ist.
* Jeder block bzw. switch-block, sowie eine *for*-, *foreach*- und *using*-Anweisung erstellt einen Deklarationsabschnitt für lokale Variablen und lokale Konstanten . Namen werden in diesen Deklarationsabschnitt durch local-variable-declarations und local-constant-declarations eingeführt. Beachten Sie, dass Blocks, die innerhalb eines Texts eines Funktionsmembers oder einer anonymen Funktion oder als ein solcher Text auftreten, innerhalb des Deklarationsabschnitts für lokale Variablen, der von diesen Funktionen für ihre Parameter deklariert wird, geschachtelt sind. Demnach ist es ein Fehler, wenn z. B. eine Methode eine lokale Variable und einen Parameter mit demselben Namen aufweist.
* Jeder block bzw. switch-block erstellt einen eigenen Bezeichnungsdeklarationsabschnitt. Namen werden in diesen Deklarationsabschnitt durch labeled-statements eingeführt. Auf die Namen wird mit goto-statements verwiesen. Der Bezeichnungsdeklarationsabschnitt eines Blocks enthält beliebige geschachtelte Blöcke. Demnach kann innerhalb eines geschachtelten Blockes eine Bezeichnung mit demselben Namen wie eine Bezeichnung im umschließenden Block deklariert werden.

Die Textreihenfolge, in der Namen im Text deklariert werden, hat in der Regel keine Bedeutung. Insbesondere ist die Textreihenfolge für die Deklaration und Verwendung von Namespaces, Konstanten, Methoden, Eigenschaften, Ereignissen, Indexern, Operatoren, Instanzkonstruktoren, Destruktoren, statischen Konstruktoren und Typen unerheblich. Die Deklarationsreihenfolge hat in folgender Hinsicht Bedeutung:

* Die Deklarationsreihenfolge für Felddeklarationen und Deklarationen von lokalen Variablen bestimmt die Reihenfolge, in der deren Initialisierungen (sofern vorhanden) ausgeführt werden.
* Lokale Variablen müssen vor ihrer Verwendung definiert werden (§3.7).
* Die Deklarationsreihenfolge für Enumerationsmemberdeklarationen (§14.3) ist von Bedeutung, wenn Werte von constant-expression ausgelassen werden.

Der Deklarationsabschnitt eines Namespaces ist unbeschränkt, und zwei Namespacedeklarationen mit demselben vollqualifizierten Namen ergänzen denselben Deklarationsabschnitt. Beispiel:

namespace Megacorp.Data  
{  
 class Customer  
 {  
 ...  
 }  
}

namespace Megacorp.Data  
{  
 class Order  
 {  
 ...  
 }  
}

tragen die beiden oben stehenden Namespacedeklarationen zu demselben Deklarationsabschnitt bei. In diesem Fall werden dazu zwei Klassen mit denselben vollqualifizierten Namen Megacorp.Data.Customer und Megacorp.Data.Order deklariert. Da die beiden Deklarationen zu demselben Deklarationsabschnitt beitragen, kommt es zu einem Kompilierungsfehler, wenn beide eine Deklaration einer gleichnamigen Klasse enthalten.

Der Deklarationsabschnitt eines Blocks enthält, wie oben angegeben, beliebige geschachtelte Blöcke. Demnach wird im folgenden Beispiel durch die F-Methode und die G-Methode ein Kompilierungsfehler verursacht, da der Name i im äußeren Block deklariert wurde und nicht erneut im inneren Block deklariert werden kann. Die H-Methode und die I-Methode sind jedoch gültig, da in beiden Fällen i in separaten nicht geschachtelten Blöcken deklariert wurde.

class A  
{  
 void F() {  
 int i = 0;  
 if (true) {  
 int i = 1;   
 }  
 }

void G() {  
 if (true) {  
 int i = 0;  
 }  
 int i = 1;   
 }

void H() {  
 if (true) {  
 int i = 0;  
 }  
 if (true) {  
 int i = 1;  
 }  
 }

void I() {  
 for (int i = 0; i < 10; i++)  
 H();  
 for (int i = 0; i < 10; i++)  
 H();  
 }  
}

## Member

Namespaces und Typen besitzen Member. Die Member einer Entität sind i. Allg. durch Verwenden eines qualifizierten Namens verfügbar, der mit einem Verweis auf die Entität beginnt, gefolgt vom Token "." und dem Namen des Members.

Member eines Typs sind entweder in der Typdeklaration deklariert oder wurden von der Basisklasse des Typs vererbt. Wenn ein Typ von einer Basisklasse erbt, werden alle Member der Basisklasse, außer Instanzkonstruktoren, Destruktoren und statischen Konstruktoren, zu Membern des abgeleiteten Typs. Der für einen Basisklassenmember deklarierte Zugriff bestimmt nicht, ob der Member vererbt wird. Die Vererbung erstreckt sich auf alle Member, bei denen es sich nicht um Instanzkonstruktoren, statische Konstruktoren oder Destruktoren handelt. Wenn auf einen vererbten Member in einem abgeleiteten Typ jedoch nicht zugegriffen werden kann, dann entweder aufgrund des für diesen deklarierten Zugriffs (§3.5.1) oder weil er von einer Deklaration im Typ selbst verborgen wird (§3.7.1.2).

### Namespacemember

Namespaces und Typen, die von keinem Namespace umschlossen werden, sind Member des globalen Namespaces. Dies entspricht den Namen, die im globalen Deklarationsabschnitt deklariert sind.

In einem Namespace deklarierte Namespaces und Typen sind Member dieses Namespaces. Dies entspricht den Namen, die im Deklarationsabschnitt des Namespaces deklariert sind.

Namespaces weisen keine Zugriffsbeschränkungen auf. Private, geschützte und interne Namespaces können nicht deklariert werden, und Namespacenamen sind immer öffentlich verfügbar.

### Strukturmember

Member einer Struktur sind die in der Struktur definierten Member sowie die von der direkten Basisklasse der Struktur System.ValueType und der indirekten Basisklasse object vererbten Member.

Die Member eines einfachen Typs entsprechen direkt den Membern des Strukturtyps mit einem Alias vom einfachen Typ:

* Die Member von sbyte sind die Member der System.SByte-Struktur.
* Die Member von byte sind die Member der System.Byte-Struktur.
* Die Member von short sind die Member der System.Int16-Struktur.
* Die Member von ushort sind die Member der System.UInt16-Struktur.
* Die Member von int sind die Member der System.Int32-Struktur.
* Die Member von uint sind die Member der System.UInt32-Struktur.
* Die Member von long sind die Member der System.Int64-Struktur.
* Die Member von ulong sind die Member der System.UInt64-Struktur.
* Die Member von char sind die Member der System.Char-Struktur.
* Die Member von float sind die Member der System.Single-Struktur.
* Die Member von double sind die Member der System.Double-Struktur.
* Die Member von decimal sind die Member der System.Decimal-Struktur.
* Die Member von bool sind die Member der System.Boolean-Struktur.

### Enumerationsmember

Member einer Enumeration sind die in der Enumeration definierten Konstanten sowie die von der direkten Basisklasse der Struktur System.Enum und den indirekten Basisklassen System.ValueType und object vererbten Member.

### Klassenmember

Die Member einer Klasse sind die in der Klasse deklarierten Member sowie die von der Basisklasse vererbten Member (außer bei der Klasse object, die keine Basisklasse besitzt). Die von der Basisklasse geerbten Member schließen die Konstanten, Felder, Methoden, Eigenschaften, Ereignisse, Indexer, Operatoren und Typen der Basisklasse ein. Ausgenommen sind die Instanzkonstruktoren, Destruktoren und statischen Konstruktoren der Basisklasse. Basisklassenmember werden unabhängig von ihrem Zugriff geerbt.

Eine Klassendeklaration kann Deklarationen für Konstanten, Felder, Methoden, Eigenschaften, Ereignisse, Indexer, Operatoren, Instanzkonstruktoren, Destruktoren, statische Konstruktoren und Typen enthalten.

Die Member von object und string entsprechen direkt den Membern der Klassentypen, deren Aliase sie sind:

* Die Member von object sind die Member der System.Object-Klasse.
* Die Member von string sind die Member der System.String-Klasse.

### Schnittstellenmember

Die Member einer Schnittstelle sind die in der Schnittstelle und in allen Basisschnittstellen der Schnittstelle deklarierten Member. Genau genommen, sind die Member in der Klasse object keine Member einer Schnittstelle (§13.2). Die Member in der object-Klasse stehen jedoch über die Membersuche in jedem beliebigen Schnittstellentyp (§7.4) zur Verfügung.

### Arraymember

Die Member eines Arrays sind die von der Klasse System.Array vererbten Member.

### Delegatmember

Die Member eines Delegaten sind die von der Klasse System.Delegate vererbten Member.

## Memberzugriff

Die Deklarationen von Membern ermöglichen die Steuerung des Memberzugriffs. Der Zugriff eines Members wird über den für den Member deklarierten Zugriff (§3.5.1) in Kombination mit dem Zugriff auf den unmittelbar umgebenden Typ (sofern vorhanden) festgelegt.

Wenn der Zugriff auf einen bestimmten Member zulässig ist, wird er als zugreifbar bezeichnet. Entsprechend wird ein bestimmter Member, auf den kein Zugriff möglich ist, als nicht zugreifbar bezeichnet. Der Zugriff auf einen Member ist zulässig, wenn die Textposition, an der der Zugriff erfolgt, sich in der Zugriffsdomäne (§3.5.2) des Members befindet.

### Deklarierter Zugriff

Folgende Arten des deklarierten Zugriffs auf einen Member sind möglich:

* Öffentlich. Dieser Zugriff wird durch Hinzufügen eines public-Modifizierers zur Memberdeklaration ausgewählt. Einfach ausgedrückt, steht public für unbegrenzten Zugriff.
* Geschützt. Dieser Zugriff wird durch Hinzufügen eines protected-Modifizierers zur Memberdeklaration ausgewählt. Einfach ausgedrückt, steht protected für "Zugriff ist auf die umgebende Klasse und auf Typen beschränkt, die von der umgebenden Klasse abgeleitet sind".
* Intern. Dieser Zugriff wird durch Hinzufügen eines internal-Modifizierers zur Memberdeklaration ausgewählt. Einfach ausgedrückt, steht internal für "Zugriff ist auf dieses Programm beschränkt".
* Geschützt intern (d. h. geschützt oder intern). Dieser Zugriff wird durch Hinzufügen sowohl des protected-Modifizierers als auch des internal-Modifizierers zur Memberdeklaration ausgewählt. Einfach ausgedrückt, steht protected internal für "Zugriff ist auf dieses Programm und auf Typen beschränkt, die von der umgebenden Klasse abgeleitet sind".
* Privat. Dieser Zugriff wird durch Hinzufügen eines private-Modifizierers zur Memberdeklaration ausgewählt. Einfach ausgedrückt, steht private für "Zugriff ist auf den umgebenden Typ beschränkt".

Je nachdem in welchem Kontext die Memberdeklaration erfolgt, sind nur bestimmte Typen deklarierten Zugriffs zulässig. Wenn eine Memberdeklaration keinen Zugriffsmodifizierer enthält, wird anhand des Kontexts der Deklaration der standardmäßig deklarierte Zugriff festgelegt.

* Namespaces sind implizit mit dem Zugriff public deklariert. In Namespacedeklarationen sind keine Zugriffsmodifizierer zulässig.
* In Kompilationseinheiten oder Namespaces deklarierte Typen können mit dem Zugriff public oder internal deklariert werden. Standardmäßig werden sie mit dem Zugriff internal deklariert.
* Klassenmember können einen der fünf Typen für den deklarierten Zugriff besitzen. Standardmäßig werden sie mit dem Zugriff private deklariert. (Ein als Member einer Klasse deklarierter Typ kann jeden der fünf deklarierten Zugriffe besitzen, während ein als Member eines Namespaces deklarierter Typ nur mit dem Zugriff public oder internal deklariert werden kann.)
* Strukturmember können mit dem Zugriff public, internal oder private deklariert werden. Standardmäßig werden sie mit dem Zugriff private deklariert, da Strukturen implizit versiegelt sind. Für in eine Struktur eingeführte Strukturmember (also nicht von dieser Struktur geerbte Strukturmember) kann nicht der Zugriff protected oder protected internal deklariert werden. (Ein als Member einer Struktur deklarierter Typ kann den Zugriff public, internal oder private besitzen, während ein als Member eines Namespaces deklarierter Typ nur mit dem Zugriff public oder internal deklariert werden kann.)
* Schnittstellenmember werden implizit mit dem Zugriff public deklariert. In Schnittstellenmember-Deklarationen sind keine Zugriffsmodifizierer zulässig.
* Enumerationsmember werden implizit mit dem Zugriff public deklariert. In Enumerationsmemberdeklarationen sind keine Zugriffsmodifizierer zulässig.

### Zugriffsdomänen

Die Zugriffsdomäne eines Members bilden die (möglicherweise nicht zusammenhängenden) Abschnitte des Programmtexts, in denen Zugriff auf die Member zulässig ist. Um eine Memberzugriffsdomäne definieren zu können, wird ein Member als ein Member der obersten Ebene bezeichnet, wenn er nicht innerhalb eines Typs deklariert wird, und als geschachtelt, wenn er innerhalb eines anderen Typs deklariert wird. Außerdem wird der Programmtext eines Programms als der gesamte in allen Quelldateien des Programms enthaltene Programmtext definiert, und der Programmtext eines Typs wird als der gesamte in den type-declarations dieses Typs enthaltene Programmtext definiert (ggf. einschließlich der Typen, die innerhalb des Typs geschachtelt sind).

Die Zugriffsdomäne eines vordefinierten Typs (z. B. object, int oder double) ist nicht begrenzt.

Die Zugriffsdomäne eines in einem Programm P deklarierten ungebundenen Typs T (§4.4.3) der obersten Ebene wird folgendermaßen definiert:

* Wenn T mit dem Zugriff public deklariert ist, ist die Zugriffsdomäne von T der Programmtext von P sowie aller auf P verweisenden Programme.
* Wenn die deklarierte Zugriffsart von T den Wert internal hat, entspricht die Zugriffsdomäne von T dem Programmtext von P.

Aus diesen Definitionen folgt, dass die Zugriffsdomäne eines ungebundenen Typs der obersten Ebene immer mindestens den Programmtext des Programms beinhaltet, in dem dieser Typ deklariert ist.

Die Zugriffsdomäne für einen konstruierten Typ T<A1, ...,AN> ist die Schnittmenge der Zugriffsdomäne des ungebundenen generischen Typs T und der Zugriffsdomäne der Typargumente A1, ...,AN.

Die Zugriffsdomäne eines geschachtelten Members M, der in einem Typ T innerhalb eines Programms P deklariert ist, wird folgendermaßen definiert (wobei M selbst ein Typ sein kann):

* Wenn die deklarierte Zugriffsart von M den Wert public hat, entspricht die Zugriffsdomäne von M der von T.
* Wenn M mit dem Zugriff protected internal deklariert ist, steht D für die Vereinigung des Programmtexts von P und des Programmtexts aller von T abgeleiteten Typen, die außerhalb von P deklariert wurden. Die Zugriffsdomäne von M ist die Schnittmenge der Zugriffsdomäne von T mit D.
* Wenn M mit dem Zugriff protected deklariert ist, steht D für die Vereinigung des Programmtexts von T und des Programmtexts aller von T abgeleiteten Typen. Die Zugriffsdomäne von M ist die Schnittmenge der Zugriffsdomäne von T mit D.
* Wenn die deklarierte Zugriffsart von M den Wert internal hat, entspricht die Zugriffsdomäne von M der Schnittmenge zwischen der Zugriffsdomäne von T und dem Programmtext von P.
* Wenn die deklarierte Zugriffsart von M den Wert private hat, entspricht die Zugriffsdomäne von M dem Programmtext von T.

Aus diesen Definitionen folgt, dass die Zugriffsdomäne eines geschachtelten Members immer mindestens den Programmtext des Typs beinhaltet, in welchem der Member deklariert ist. Des Weiteren kann daraus abgeleitet werden, dass die Zugriffsdomäne eines Members nie mehr als die Zugriffsdomäne des Typs einschließt, in dem der Member deklariert ist.

Beim Zugriff auf einen Typ oder Member M werden die folgenden Schritte ausgewertet, um sicherzustellen, dass der Zugriff zulässig ist:

* Wenn M innerhalb eines Typs deklariert wird (im Gegensatz zu einer Kompilationseinheit oder einem Namespace), tritt ein Kompilierungsfehler auf, wenn auf diesen Typ nicht zugegriffen werden kann.
* Wenn M als public deklariert ist, ist der Zugriff zulässig.
* Wenn M andernfalls als protected internal deklariert ist, ist der Zugriff zulässig, wenn er innerhalb des Programms erfolgt, in der M deklariert ist, oder wenn er innerhalb einer Klasse erfolgt, die von der Klasse abgeleitet ist, in der M deklariert ist, und wenn der Zugriff über den abgeleiteten Klassentyp (§3.5.3) ausgeführt wird.
* Wenn M andernfalls als protected deklariert ist, ist der Zugriff zulässig, wenn er innerhalb der Klasse erfolgt, in der M deklariert ist, oder wenn er innerhalb einer Klasse erfolgt, die von der Klasse abgeleitet ist, in der M deklariert ist, und wenn der Zugriff über den abgeleiteten Klassentyp (§3.5.3) ausgeführt wird.
* Wenn M andernfalls als internal deklariert ist, ist der Zugriff zulässig, wenn er innerhalb des Programms erfolgt, in dem M deklariert ist.
* Wenn M andernfalls als private deklariert ist, ist der Zugriff zulässig, wenn er innerhalb des Typs erfolgt, in dem M deklariert ist.
* In allen anderen Fällen kann auf den Typ oder Member nicht zugegriffen werden, und es tritt ein Kompilierungsfehler auf.

In dem Beispiel

public class A  
{  
 public static int X;  
 internal static int Y;  
 private static int Z;  
}

internal class B  
{  
 public static int X;  
 internal static int Y;  
 private static int Z;

public class C  
 {  
 public static int X;  
 internal static int Y;  
 private static int Z;  
 }

private class D  
 {  
 public static int X;  
 internal static int Y;  
 private static int Z;  
 }  
}

haben die Klassen und Member die folgenden Zugriffsdomänen:

* Die Zugriffsdomäne von A und A.X ist unbegrenzt.
* Die Zugriffsdomäne von A.Y, B, B.X, B.Y, B.C, B.C.X und B.C.Y ist der Programmtext des enthaltenden Programms.
* Die Zugriffsdomäne von A.Z ist der Programmtext von A.
* Die Zugriffsdomäne von B.Z und B.D ist der Programmtext von B, einschließlich des Programmtexts von B.C und B.D.
* Die Zugriffsdomäne von B.C.Z ist der Programmtext von B.C.
* Die Zugriffsdomäne von B.D.X und B.D.Y ist der Programmtext von B, einschließlich des Programmtexts von B.C und B.D.
* Die Zugriffsdomäne von B.D.Z ist der Programmtext von B.D.

Anhand des Beispiels ist zu erkennen, dass die Zugriffsdomäne eines Members niemals größer als die eines enthaltenden Typs ist. Obwohl beispielsweise alle X-Member über einen als öffentlich deklarierten Zugriff verfügen, besitzen alle mit Ausnahme von A.X durch einen umgebenden Typ eingeschränkte Zugriffsdomänen.

Alle Member einer Basisklasse mit Ausnahme von Instanzkonstruktoren, Destruktoren und statische Konstruktoren werden von abgeleiteten Typen geerbt (siehe §3.4). Dies schließt auch private Member einer Basisklasse ein. Die Zugriffsdomäne eines privaten Members beinhaltet nur den Programmtext des Typs, in dem der Member deklariert ist. In dem Beispiel

class A  
{  
 int x;

static void F(B b) {  
 b.x = 1; // Ok  
 }  
}

class B: A  
{  
 static void F(B b) {  
 b.x = 1; // Error, x not accessible  
 }  
}

erbt die B-Klasse den privaten Member x von der A-Klasse. Da der Member privat ist, kann nur innerhalb von class-body von A darauf zugegriffen werden. Daher ist der Zugriff auf b.x in der A.F-Methode erfolgreich, in der B.F-Methode tritt jedoch ein Fehler auf.

### Geschützter Zugriff für Instanzmember

Wenn der Zugriff auf einen als protected deklarierten Instanzmember außerhalb des Programmtexts der Klasse erfolgt, in der er deklariert ist, bzw. wenn der Zugriff auf einen als protected internal deklarierten Member außerhalb des Programmtexts des Programms erfolgt, in dem er deklariert ist, muss der Zugriff in einer Klassendeklaration ausgeführt werden, die von der Klasse erbt, in der er deklariert ist. Außerdem muss der Zugriff über eine Instanz dieses abgeleiteten Klassentyps oder eines Klassentyps, der aus dieser konstruiert wurde, erfolgen. Mit dieser Einschränkung wird verhindert, dass eine abgeleitete Klasse auf geschützte Member anderer abgeleiteter Klassen zugreift, auch wenn die Member von derselben Basisklasse vererbt wurden.

Angenommen, B ist eine Basisklasse, die einen geschützten Instanzmember M deklariert, und D ist eine Klasse, die von B abgeleitet ist. Innerhalb des class-body von D kommen für den Zugriff auf M folgende Formen infrage:

* Ein nicht qualifizierter type-name oder eine nicht qualifizierte primary-expression der Form M.
* Eine primary-expression der Form E.M, sofern E vom Typ T oder einer von T abgeleiteten Klasse ist, wobei T der Klassentyp D oder ein aus D konstruierter Klassentyp ist.
* Eine primary-expression der Form base.M.

Zusätzlich zu diesen Zugriffsformen kann eine abgeleitete Klasse auf einen geschützten Instanzkonstruktor einer Basisklasse in einem constructor-initializer (§10.11.1) zugreifen.

In dem Beispiel

public class A  
{  
 protected int x;

static void F(A a, B b) {  
 a.x = 1; // Ok  
 b.x = 1; // Ok  
 }  
}

public class B: A  
{  
 static void F(A a, B b) {  
 a.x = 1; // Error, must access through instance of B  
 b.x = 1; // Ok  
 }  
}

ist es innerhalb von A möglich, auf x sowohl über Instanzen von A als auch über Instanzen von B zuzugreifen, da in beiden Fällen der Zugriff entweder über eine Instanz von A oder über eine von A abgeleitete Klasse erfolgt. Es ist jedoch nicht möglich, innerhalb von B auf x über eine Instanz von A zuzugreifen, da A nicht von B abgeleitet ist.

In dem Beispiel

class C<T>  
{  
 protected T x;  
}

class D<T>: C<T>  
{  
 static void F() {  
 D<T> dt = new D<T>();  
 D<int> di = new D<int>();  
 D<string> ds = new D<string>();  
 dt.x = default(T);  
 di.x = 123;  
 ds.x = "test";  
 }  
}

sind die drei Zuweisungen zu x zulässig, da sie alle über Instanzen von Klassentypen erfolgen, die aus dem generischen Typ konstruiert wurden.

### Zugriffsbeschränkungen

Verschiedene Konstrukte in der C#-Sprache erfordern einen Typ, der mindestens so zugreifbar ist wie ein Member oder ein anderer Typ. Von einem Typ T heißt es, dass er mindestens so verfügbar wie ein Member oder Typ M ist, wenn die Zugriffsdomäne von T eine Obermenge der Zugriffsdomäne von M ist. Anders ausgedrückt: T ist mindestens so verfügbar wie M, wenn T in allen Kontexten verfügbar ist, in denen M verfügbar ist.

Es bestehen die folgenden Zugriffsbeschränkungen:

* Die direkte Basisklasse eines Klassentyps muss mindestens so zugreifbar sein wie der Klassentyp selbst.
* Die expliziten Basisschnittstellen eines Schnittstellentyps müssen mindestens so zugreifbar sein wie der Schnittstellentyp selbst.
* Der Rückgabetyp und die Parametertypen eines Delegattyps müssen mindestens so zugreifbar sein wie der Delegattyp selbst.
* Der Typ einer Konstante muss mindestens so zugreifbar sein wie die Konstante selbst.
* Der Typ eines Feldes muss mindestens so zugreifbar sein wie das Feld selbst.
* Der Rückgabetyp und die Parametertypen einer Methode müssen mindestens so zugreifbar sein wie die Methode selbst.
* Der Typ einer Eigenschaft muss mindestens so zugreifbar sein wie die Eigenschaft selbst.
* Der Typ eines Ereignisses muss mindestens so zugreifbar sein wie das Ereignis selbst.
* Der Typ und die Parametertypen eines Indexers müssen mindestens so zugreifbar sein wie der Indexer selbst.
* Der Rückgabetyp und die Parametertypen eines Operators müssen mindestens so zugreifbar sein wie der Operator selbst.
* Die Parametertypen eines Instanzkonstruktors müssen mindestens so verfügbar sein wie der Instanzkonstruktor selbst.

In dem Beispiel

class A {...}

public class B: A {...}

verursacht die B-Klasse einen Kompilierungsfehler, da A nicht mindestens so zugreifbar ist wie B.

Analog gilt im Beispiel

class A {...}

public class B  
{  
 A F() {...}

internal A G() {...}

public A H() {...}  
}

dass die H-Methode in B einen Kompilierungsfehler verursacht, da der Rückgabetyp A nicht mindestens so zugreifbar ist wie die Methode.

## Signaturen und Überladung

Methoden, Instanzkonstruktoren, Indexer und Operatoren werden durch ihre Signaturen gekennzeichnet:

* Die Signatur einer Methode besteht aus dem Namen der Methode, der Anzahl von Typparametern sowie dem Typ und der Art (Wert, Verweis oder Ausgabe) aller ihrer formalen Parameter von links nach rechts. Zu diesem Zweck werden sämtliche Typparameter der Methode, die im Typ eines formalen Parameters enthalten sind, nicht anhand ihres Namens, sondern anhand ihrer Ordinalposition in der Typargumentliste der Methode identifiziert. Die Signatur einer Methode enthält ausdrücklich weder den Rückgabetyp noch den params-Modifizierer, der für den am weitesten rechts stehenden Parameter angegeben werden kann, noch die optionalen Typparametereinschränkungen.
* Die Signatur eines Instanzkonstruktors besteht aus dem Typ und der Art (Wert, Verweis oder Ausgabe) aller seiner formalen Parameter von links nach rechts. Die Signatur eines Instanzkonstruktors enthält ausdrücklich nicht den params-Modifizierer, der für den am weitesten rechts stehenden Parameter angegeben werden kann.
* Die Signatur eines Indexers besteht aus dem Typ aller seiner formalen Parameter von links nach rechts. Die Signatur eines Indexers enthält ausdrücklich weder den Elementtyp noch den params-Modifizierer, der für den am weitesten rechts stehenden Parameter angegeben werden kann.
* Die Signatur eines Operators besteht aus dem Namen des Operators und dem Typ aller seiner formalen Parameter von links nach rechts. Die Signatur eines Operators beinhaltet ausdrücklich nicht den Ergebnistyp.

Signaturen sind der Mechanismus, der das Überladen von Membern in Klassen, Strukturen und Schnittstellen ermöglicht:

* Das Überladen von Methoden ermöglicht es einer Klasse, Struktur oder Schnittstelle, mehrere Methoden mit demselben Namen zu deklarieren, vorausgesetzt, sie besitzen eindeutige Signaturen innerhalb dieser Klasse, Struktur oder Schnittstelle.
* Das Überladen von Instanzkonstruktoren ermöglicht es einer Klasse oder Struktur, mehrere Instanzkonstruktoren zu deklarieren, vorausgesetzt, sie besitzen eindeutige Signaturen innerhalb dieser Klasse oder Struktur.
* Das Überladen von Indexern ermöglicht es einer Klasse, Struktur oder Schnittstelle, mehrere Indexer zu deklarieren, vorausgesetzt, sie besitzen eindeutige Signaturen innerhalb dieser Klasse, Struktur oder Schnittstelle.
* Das Überladen von Operatoren ermöglicht es einer Klasse oder Struktur, mehrere Operatoren mit demselben Namen zu deklarieren, vorausgesetzt, sie besitzen eindeutige Signaturen innerhalb dieser Klasse oder Struktur.

Obwohl der out-Parametermodifizierer und der ref-Parametermodifizierer als Teil einer Signatur gelten, können sich in ein und demselben Typ deklarierte Member nicht allein in ref und out in ihrer Signatur unterscheiden. Wenn zwei Member im gleichen Typ mit Signaturen deklariert werden, die identisch wären, wenn alle Parameter mit out-Modifizierer in Parameter mit ref-Modifizierer geändert würden, tritt ein Kompilierungsfehler auf. In anderen Einsatzbereichen des Signaturabgleichs (z. B. beim Verdecken oder Überschreiben), werden ref und out als Teil der Signatur und nicht als gleichwertig betrachtet. (Diese Einschränkung ist erforderlich, damit C#-Programme problemlos für die Ausführung in der Common Language Infrastructure (CLI) übersetzt werden können, die keine Möglichkeit dafür vorsieht, Methoden zu definieren, die sich lediglich in ref und out unterscheiden.)

Für die Zwecke von Signaturen werden die Typen object und dynamic als identisch angesehen. In einem einzelnen Typ deklarierte Member können sich daher in der Signatur nicht nur durch object und dynamic unterscheiden.

Das folgende Beispiel zeigt eine Reihe überladener Methodendeklarationen mit ihren jeweiligen Signaturen.

interface ITest  
{  
 void F(); // F()

void F(int x); // F(int)

void F(ref int x); // F(ref int)

void F(out int x); // F(out int) error

void F(int x, int y); // F(int, int)

int F(string s); // F(string)

int F(int x); // F(int) error

void F(string[] a); // F(string[])

void F(params string[] a); // F(string[]) error  
}

Alle ref-Parametermodifizierer und out-Parametermodifizierer (§10.6.1) sind Teil einer Signatur. F(int) und F(ref int) sind folglich eindeutige Signaturen. Allerdings können F(ref int) und F(out int) nicht in der gleichen Schnittstelle deklariert werden, da sich ihre Signaturen lediglich in ref und out unterscheiden. Der Rückgabetyp und der params-Modifizierer sind außerdem nicht Teil der Signatur. Somit ist keine Überladung allein auf Grundlage des Rückgabetyps oder des Angebens oder Weglassens des params-Modifizierers möglich. Die Deklarationen der oben genannten Methoden F(int) und F(params string[]) erzeugen daher einen Kompilierungsfehler.

## Gültigkeitsbereiche

Der Gültigkeitsbereich eines Namens ist der Bereich des Programmtexts, in dem es möglich ist, auf die durch den Namen deklarierte Entität zu verweisen, ohne den qualifizierten Namen zu verwenden. Gültigkeitsbereiche können geschachtelt werden, und ein innerer Gültigkeitsbereich kann die Bedeutung eines Namens aus einem äußeren Gültigkeitsbereich neu deklarieren. (Hierdurch wird die aus §3.3 folgende Einschränkung allerdings nicht aufgehaben, die besagt, dass in einem geschachtelten Block keine lokale Variable deklariert werden kann, die den gleichen Namen wie eine lokale Variable in einem umgebenden Block hat.) Der Name des äußeren Gültigkeitsbereichs wird dann im Programmtextbereich des inneren Gültigkeitsbereichs als verborgen bezeichnet, und der Zugriff auf den äußeren Namen ist nur durch Verwendung des qualifizierten Namens möglich.

* Der Gültigkeitsbereich eines durch eine namespace-member-declaration (§9.5) ohne einschließende namespace-declaration deklarierten Namespacemembers ist der gesamte Programmtext.
* Der Gültigkeitsbereich eines Namespacemembers, der durch eine namespace-member-declaration innerhalb einer namespace-declaration deklariert wurde und den vollqualifizierten Namen N hat, ist der namespace-body jeder namespace-declaration, deren vollqualifizierter Name N ist oder mit N, gefolgt von einem Punkt, beginnt.
* Der Gültigkeitsbereich eines durch extern-alias-directive definierten Namens erstreckt sich über die using-directives, global-attributes und namespace-member-declarations seiner unmittelbar enthaltenden Kompilationseinheit bzw. des unmittelbar enthaltenden Namespacetexts. Eine extern-alias-directive trägt zum zugrunde liegenden Deklarationsabschnitt keine neuen Member bei. Mit anderen Worten: Eine extern-alias-directive ist nicht transitiv. Stattdessen wirkt sie sich nur auf die Kompilationseinheit bzw. den Namespacetext aus, in der bzw. dem sie auftritt.
* Der Gültigkeitsbereich eines mit einer using-directive (§9.4) definierten oder importierten Namens erstreckt sich über die namespace-member-declarations der compilation-unit bzw. des namespace-body, in dem die using-directive auftritt. Eine using-directive stellt 0 (null) oder mehr Namespace- oder Typnamen innerhalb einer bestimmten compilation-unit bzw. eines namespace-body zur Verfügung, erweitert den zugrunde liegenden Deklarationsabschnitt aber nicht um neue Member. Eine using-directive ist also nicht transitiv, sondern betrifft nur die compilation-unit bzw. den namespace-body, in der bzw. dem sie auftritt.
* Der Gültigkeitsbereich eines durch eine type-parameter-list in einer class-declaration (§10.1) deklarierten Typparameters umfasst die class-base, die type-parameter-constraints-clauses und den class-body der class-declaration.
* Der Gültigkeitsbereich eines durch eine type-parameter-list in einer struct-declaration (§11.1) deklarierten Typparameters umfasst die struct-interfaces, die type-parameter-constraints-clauses und den struct-body der struct-declaration.
* Der Gültigkeitsbereich eines durch eine type-parameter-list in einer interface-declaration (§13.1) deklarierten Typparameters umfasst die interface-base, die type-parameter-constraints-clauses und den interface-body der interface-declaration.
* Der Gültigkeitsbereich eines durch eine type-parameter-list in einer delegate-declaration (§15.1) deklarierten Typparameters umfasst den return-type, die formal-parameter-list und die type-parameter-constraints-clauses der delegate-declaration.
* Der Gültigkeitsbereich eines durch eine class-member-declaration (§10.1.6) deklarierten Members ist der class-body, in dem die Deklaration erfolgt. Außerdem wird der Gültigkeitsbereich eines Klassenmembers auf den class-body jener abgeleiteten Klassen ausgedehnt, die in der Zugriffsdomäne (§3.5.2) des Members enthalten sind.
* Der Gültigkeitsbereich eines durch eine struct-member-declaration (§11.2) deklarierten Members ist der struct-body, in dem die Deklaration erfolgt.
* Der Gültigkeitsbereich eines durch eine enum-member-declaration (§14.3) deklarierten Members ist der enum-body, in dem die Deklaration erfolgt.
* Der Gültigkeitsbereich eines in einer method-declaration (§10.6) deklarierten Parameters ist der method-body dieser method-declaration.
* Der Gültigkeitsbereich eines in einer indexer-declaration (§10.9) deklarierten Parameters sind die accessor-declarations dieser indexer-declaration.
* Der Gültigkeitsbereich eines in einer operator-declaration (§10.10) deklarierten Parameters ist der block dieser operator-declaration.
* Der Gültigkeitsbereich eines in einer constructor-declaration (§10.11) deklarierten Parameters ist der constructor-initializer sowie der block dieser constructor-declaration.
* Der Gültigkeitsbereich eines in einer lambda-expression (§7.15) deklarierten Parameters ist der lambda-expression-body der lambda-expression
* Der Gültigkeitsbereich eines in einer anonymous-method-expression (§7.15) deklarierten Parameters ist der block der anonymous-method-expression.
* Der Gültigkeitsbereich einer in einem labeled-statement (§8.4) deklarierten Bezeichnung ist der block, in dem die Deklaration erfolgt.
* Der Gültigkeitsbereich einer durch eine local-variable-declaration (§8.5.1) deklarierten lokalen Variablen ist der Block, in dem die Deklaration erfolgt.
* Der Gültigkeitsbereich einer in einem switch-block einer switch-Anweisung (§8.7.2) deklarierten lokalen Variablen ist der switch-block.
* Der Gültigkeitsbereich einer in einem for-initializer einer for-Anweisung (§8.8.3) deklarierten lokalen Variablen ist der for-initializer, die for-condition, der for-iterator, und das enthaltene statement der for-Anweisung.
* Der Gültigkeitsbereich einer in einer local-constant-declaration (§8.5.2) deklarierten lokalen Konstanten ist der Block, in dem die Deklaration erfolgt. Wenn an einer Textposition vor dem entsprechenden constant-declarator auf eine lokale Konstante verwiesen wird, tritt ein Kompilierungsfehler auf.
* Der Gültigkeitsbereich einer als Teil einer foreach-statement, using-statement, lock-statement oder einer query-expression deklarierten Variable wird durch die Erweiterung des entsprechenden Konstrukts bestimmt.

Innerhalb des Gültigkeitsbereichs eines Namespace-, Klassen-, Struktur- oder Enumerationsmembers ist es möglich, auf den Member an einer Textposition zu verweisen, die vor der Deklaration des Members liegt. Beispiel:

class A  
{  
 void F() {  
 i = 1;  
 }

int i = 0;  
}

In diesem Beispiel kann in F bereits vor der Deklaration von i darauf verwiesen werden.

Innerhalb des Gültigkeitsbereichs einer lokalen Variablen verursacht ein Verweis auf die lokale Variable an einer Textposition, die vor dem local-variable-declarator der lokalen Variablen liegt, einen Kompilierungsfehler. Beispiel:

class A  
{  
 int i = 0;

void F() {  
 i = 1; // Error, use precedes declaration  
 int i;  
 i = 2;  
 }

void G() {  
 int j = (j = 1); // Valid  
 }

void H() {  
 int a = 1, b = ++a; // Valid  
 }  
}

In der oben dargestellten F-Methode verweist die erste Zuweisung zu i ausdrücklich nicht auf das im äußeren Gültigkeitsbereich deklarierte Feld. Stattdessen verweist sie auf die lokale Variable und verursacht einen Kompilierungsfehler, da die Zuweisung im Text vor der Deklaration der Variablen liegt. In der G-Methode ist die Verwendung von j im Initialisierer für die Deklaration von j zulässig, da die Verwendung nicht vor dem local-variable-declarator liegt. In der H-Methode verweist ein nachfolgender local-variable-declarator ordnungsgemäß auf eine lokale Variable, die durch einen früheren local-variable-declarator innerhalb derselben local-variable-declaration deklariert wurde.

Die Regeln für den Gültigkeitsbereich lokaler Variablen wurden festgelegt, um zu garantieren, dass die Bedeutung eines in einem Ausdruckskontext verwendeten Namens innerhalb eines Blockes immer identisch ist. Wenn sich der Gültigkeitsbereich einer lokalen Variablen nur von ihrer Deklaration bis zum Ende des Blockes erstrecken würde, beträfe die erste Zuweisung im vorangehenden Beispiel die Instanzvariable und die zweite Zuweisung die lokale Variable. Bei späteren Neuanordnungen der Anweisungen im Block könnte dies zu Kompilierungsfehlern führen.

Die Bedeutung eines Namens kann sich in einem Block auf Grund des Kontextes, in dem der Name verwendet wird, unterscheiden. In dem Beispiel

using System;

class A {}

class Test  
{  
 static void Main() {  
 string A = "hello, world";  
 string s = A; // expression context

Type t = typeof(A); // type context

Console.WriteLine(s); // writes "hello, world"  
 Console.WriteLine(t); // writes "A"  
 }  
}

wird der Name A in einem Ausdruckskontext verwendet, um auf die lokale Variable A zu verweisen, und in einem Typkontext, um auf die Klasse A zu verweisen.

### Verdecken von Namen

Der Gültigkeitsbereich einer Entität umfasst in der Regel mehr Programmtext als der Deklarationsabschnitt der Entität. Insbesondere kann der Gültigkeitsbereich einer Entität Deklarationen beinhalten, die neue Deklarationsabschnitte mit Entitäten desselben Namens einführen. Derartige Deklarationen führen dazu, dass die ursprüngliche Entität verborgen wird. Entsprechend wird eine Entität als sichtbar bezeichnet, wenn sie nicht verborgen ist.

Das Verdecken von Namen tritt bei der Überlappung von Gültigkeitsbereichen durch Schachtelung und Vererbung auf. Die Merkmale der zwei Arten von Verdeckungen werden in den folgenden Abschnitten erklärt.

#### Verdecken durch Schachtelung

Das Verdecken von Namen durch Schachtelung kann als Ergebnis der Schachtelung von Namespaces oder von Typen innerhalb von Namespaces, als Ergebnis der Schachtelung von Typen innerhalb von Klassen oder Strukturen und als Ergebnis der Deklaration von Parametern und lokalen Variablen auftreten.

In dem Beispiel

class A  
{  
 int i = 0;

void F() {  
 int i = 1;  
 }

void G() {  
 i = 1;  
 }  
}

wird die Instanzvariable i innerhalb der F-Methode durch die lokale Variable i verborgen. Innerhalb der G-Methode bezieht sich i jedoch immer noch auf die Instanzvariable.

Wenn ein Name in einem inneren Gültigkeitsbereich einen Namen in einem äußeren Gültigkeitsbereich ausblendet, werden auch sämtliche überladene Vorkommen dieses Namens ausgeblendet. In dem Beispiel

class Outer  
{  
 static void F(int i) {}

static void F(string s) {}

class Inner  
 {  
 void G() {  
 F(1); // Invokes Outer.Inner.F  
 F("Hello"); // Error  
 }

static void F(long l) {}  
 }  
}

ruft der Aufruf F(1) das in Inner deklarierte F auf, da alle äußeren Vorkommen von F durch die innere Deklaration verborgen sind. Aus demselben Grund führt der Aufruf F("Hello") zu einem Kompilierungsfehler.

#### Verdecken durch Vererbung

Das Verdecken von Namen durch Vererbung tritt auf, wenn Klassen oder Strukturen Namen neu deklarieren, die von der Basisklasse geerbt wurden. Dieser Typ der Verdeckung von Namen nimmt eine der folgenden Formen an:

* Durch in eine Klasse oder Struktur eingeführte Konstanten, Felder, Eigenschaften, Ereignisse oder Typen werden alle Basisklassenelemente mit demselben Namen verdeckt.
* Durch in eine Klasse oder Struktur eingeführte Methoden werden alle Nicht-Methodenbasisklassenmember mit demselben Namen und alle Basisklassenmethoden mit derselben Signatur (Methodenname und Parameteranzahl, Modifizierer und Typen) verdeckt.
* Durch in eine Klasse oder Struktur eingeführte Indexer werden alle Basisklassenindexer mit derselben Signatur (Parameterzahl und Typen) verdeckt.

Die Regeln für Operatordeklarationen (§10.10) verhindern, dass abgeleitete Klassen einen Operator mit derselben Signatur wie die eines Operators in einer Basisklasse deklarieren. Folglich verdecken sich Operatoren nie gegenseitig.

Im Gegensatz zum Verdecken eines Namens aus einem äußeren Gültigkeitsbereich führt das Ausblenden eines verfügbaren Namens aus einem geerbten Gültigkeitsbereich zur Ausgabe einer Warnung. In dem Beispiel

class Base  
{  
 public void F() {}  
}

class Derived: Base  
{  
 public void F() {} // Warning, hiding an inherited name  
}

verursacht die Deklaration von F in Derived die Ausgabe einer Warnung. Das Verdecken eines geerbten Namens ist ausdrücklich zulässig, da nur so die getrennte Entwicklung von Basisklassen gewährleistet ist. Die oben dargestellte Situation hätte beispielsweise eintreten können, wenn eine spätere Version von Base eine F-Methode einführt, die in der vorherigen Version der Klasse nicht vorhanden war. Wenn die oben dargestellte Situation nicht zulässig wäre, würden alle Änderungen an einer Basisklasse in einer anderen Version der Klassenbibliothek möglicherweise dazu führen, dass die abgeleiteten Klassen ungültig sind.

Die durch das Verdecken eines vererbten Namens ausgelöste Warnung kann verhindert werden, indem der new-Modifizierer verwendet wird:

class Base  
{  
 public void F() {}  
}

class Derived: Base  
{  
 new public void F() {}  
}

Der new-Modifizierer gibt an, dass F in Derived "neu" ist und somit tatsächlich den vererbten Member verdecken soll.

Die Deklaration eines neuen Members verdeckt einen geerbten Member nur innerhalb des Gültigkeitsbereichs des neuen Members.

class Base  
{  
 public static void F() {}  
}

class Derived: Base  
{  
 new private static void F() {} // Hides Base.F in Derived only  
}

class MoreDerived: Derived  
{  
 static void G() { F(); } // Invokes Base.F  
}

Im oben angeführten Beispiel verdeckt die Deklaration von F in Derived das F, das von Base vererbt wurde. Da das neue F in Derived privaten Zugriff besitzt, erstreckt sich der Gültigkeitsbereich nicht auf MoreDerived. Folglich ist der Aufruf F() in MoreDerived.G zulässig und führt zum Aufruf von Base.F.

## Namespace- und Typnamen

In einem C#-Programm müssen in verschiedenen Kontexten ein namespace-name oder ein type-name angegeben werden.

namespace-name:  
namespace-or-type-name

type-name:  
namespace-or-type-name

namespace-or-type-name:  
identifier type-argument-listopt  
namespace-or-type-name . identifier type-argument-listoptqualified-alias-member

Ein namespace-name ist ein namespace-or-type-name, der auf einen Namespace verweist. Entsprechend der im Folgenden beschriebenen Auflösung muss der namespace-or-type-name eines namespace-name auf einen Namespace verweisen, da ansonsten ein Kompilierungsfehler auftritt. Ein namespace-name darf keine Typargumente (§4.4.1) enthalten (nur Typen können Typargumente besitzen).

Ein type-name ist ein namespace-or-type-name, der auf einen Typ verweist. Entsprechend der im Folgenden beschriebenen Auflösung muss der namespace-or-type-name eines type-name auf einen Typ verweisen, da ansonsten ein Kompilierungsfehler auftritt.

Wenn der namespace-or-type-name ein qualified-alias-member ist, finden Sie in §9.7 eine Beschreibung seiner Bedeutung. Andernfalls besitzt ein namespace-or-type-name eine von vier Formen:

* I
* I<A1, ..., AK>
* N.I
* N.I<A1, ..., AK>

wobei I ein einzelner Bezeichner, N ein namespace-or-type-name und <A1, ..., AK> eine optionale type-argument-list ist. Wenn keine type-argument-list angegeben ist, wird K als 0 (null) betrachtet.

Die Bedeutung eines namespace-or-type-name wird folgendermaßen bestimmt:

* Wenn der namespace-or-type-name die Form I aufweist oder die Form I<A1, ..., AK>:
* Wenn K (null) ist und der namespace-or-type-name innerhalb einer generischen Methodendeklaration steht (§10.6), und wenn diese Deklaration einen Typparameter (§10.1.3) mit dem Namen I enthält, dann verweist der namespace-or-type-name auf diesen Typparameter.
* Wenn andernfalls der namespace-or-type-name innerhalb einer Typdeklaration steht, gilt für jeden Instanztyp T (§10.3.1) vom Instanztyp dieser Typdeklaration über die Instanztypen aller umschließenden Klassen- oder Strukturdeklarationen (sofern vorhanden):
* Wenn K 0 (null) ist und die Deklaration von T einen Typparameter mit dem Namen I enthält, dann verweist der namespace-or-type-name auf diesen Typparameter.
* Wenn andernfalls der namespace-or-type-name im Text der Typdeklaration steht und T oder einer seiner Basistypen einen verschachtelten zugreifbaren Typ enthält, der den Namen I und K Typparameter aufweist, dann verweist der namespace-or-type-name auf diesen mit den angegebenen Typargumenten konstruierten Typ. Wenn mehrere entsprechende Typen vorhanden sind, wird der stärker abgeleitete Typ ausgewählt. Beachten Sie, dass Member, die keine Typen sind (Konstanten, Felder, Methoden, Eigenschaften, Indexer und Operatoren, Instanzkonstruktoren, Destruktoren und statische Konstruktoren), und Typmember mit abweichender Anzahl von Typparametern bei der Bestimmung der Bedeutung eines namespace-or-type-name ignoriert werden.
* Wenn die vorherigen Schritte zu keinem Ergebnis führen, werden für jeden Namespace N ab dem Namespace, in dem der namespace-or-type-name vorkommt, über jeden einschließenden Namespace (sofern vorhanden) bis zum globalen Namespace die folgenden Schritte ausgewertet, bis eine Entität gefunden wird:
* Wenn K 0 (null) ist und I der Name eines Namespaces in N ist, gilt Folgendes:
* Wenn die Position, an der der namespace-or-type-name auftritt, von einer Namespacedeklaration für N eingeschlossen ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive enthält, die den Namen I einem Namespace oder Typ zuordnet, dann ist der namespace-or-type-name mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls verweist der namespace-or-type-name auf den Namespace mit dem Namen I in N.
* Wenn N andernfalls einen zugreifbaren Typ mit dem Namen I und K Typparametern enthält, dann gilt Folgendes:
* Wenn K 0 (null) ist und die Position, an der der namespace-or-type-name auftritt, von einer Namespacedeklaration für N eingeschlossen ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive enthält, die den Namen I einem Namespace oder Typ zuordnet, dann ist der namespace-or-type-name mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls verweist der namespace-or-type-name auf den mit den angegebenen Typargumenten konstruierten Typ.
* Wenn andernfalls die Position, an der der namespace-or-type-name auftritt, in eine Namespacedeklaration für N eingeschlossen ist, dann gilt Folgendes:
* Wenn K 0 (null) ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive enthält, die den Namen I mit einem importierten Namespace oder Typ verknüpft, verweist der namespace-or-type-name auf diesen Namespace oder Typ.
* Wenn andernfalls die über die using-namespace-directives der Namespacedeklaration importierten Namespaces genau einen Typ mit dem Namen I und K Typparametern enthalten, dann verweist der namespace-or-type-name auf den mit den angegebenen Typargumenten konstruierten Typ.
* Wenn andernfalls die über die using-namespace-directives der Namespacedeklaration importierten Namespaces mehr als einen Typ mit dem Namen I und K Typparametern enthalten, dann ist der namespace-or-type-name mehrdeutig, und es tritt ein Fehler auf.
* Andernfalls ist der namespace-or-type-name nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Andernfalls weist der namespace-or-type-name die Form N.I oder die Form N.I<A1, ..., auf AK>. N wird zuerst als namespace-or-type-name aufgelöst. Wenn die Auflösung von N nicht erfolgreich ist, tritt ein Kompilierungsfehler auf. Andernfalls werden N.I oder N.I<A1, ..., AK> wie folgt aufgelöst:
* Wenn K 0 (null) ist, N auf einen Namespace verweist und N einen geschachtelten Namespace mit dem Namen I enthält, dann verweist der namespace-or-type-name auf diesen geschachtelten Namespace.
* Wenn N andernfalls auf einen Namespace verweist und N einen zugreifbaren Typ mit dem Namen I und K Typparametern enthält, dann verweist der namespace-or-type-name auf den mit den angegeben Typargumenten konstruierten Typ.
* Wenn N andernfalls auf einen (ggf. konstruierten) Klassen- oder Strukturtyp verweist und N oder eine der Basisklassen einen geschachtelten zugreifbaren Typ mit dem Namen I und K Typparametern enthält, dann verweist der namespace-or-type-name auf den mit den angegebenen Typargumenten konstruierten Typ. Wenn mehrere entsprechende Typen vorhanden sind, wird der stärker abgeleitete Typ ausgewählt. Beachten Sie, dass wenn die Bedeutung von N.I als Teil des Auflösens der Basisklassenspezifikation von N bestimmt wird, die direkte Basisklasse von N als Objekt betrachtet wird (§10.1.4.1).
* Andernfalls ist N.I ein ungültiger namespace-or-type-name, und es tritt ein Kompilierungsfehler auf.

Ein namespace-or-type-name darf unter einer der folgenden Bedingungen auf eine statische Klasse (§10.1.1.3) verweisen:

* Der namespace-or-type-name ist das T in einem namespace-or-type-name der Form T.I oder
* Der namespace-or-type-name ist das T in einer typeof-expression (§7.5.11) der Form typeof(T).

### Vollqualifizierte Namen

Jeder Namespace und Typ besitzt einen vollqualifizierten Namen, der ihn von allen anderen Namespaces oder Typen eindeutig unterscheidet. Der vollqualifizierte Name eines Namespaces oder Typs N wird folgendermaßen bestimmt:

* Wenn N ein Member des globalen Namespaces ist, lautet sein vollqualifizierter Name N.
* Andernfalls lautet sein vollqualifizierter Name S.N, wobei S der vollqualifizierte Name des Namespaces oder Typs ist, in dem N deklariert ist.

D. h., der vollqualifizierte Name von N stellt den vollständigen hierarchischen Pfad der zu N führenden Bezeichner dar, beginnend beim globalen Namespace. Da jeder Member eines Namespaces oder Typs einen eindeutigen Namen besitzen muss, ist auch der voll gekennzeichnete Name eines Namespaces oder Typs immer eindeutig.

Das folgende Beispiel zeigt verschiedene Namespace- oder Typdeklarationen zusammen mit deren verknüpften voll gekennzeichneten Namen.

class A {} // A

namespace X // X  
{  
 class B // X.B  
 {  
 class C {} // X.B.C  
 }

namespace Y // X.Y  
 {  
 class D {} // X.Y.D  
 }  
}

namespace X.Y // X.Y  
{  
 class E {} // X.Y.E  
}

## Automatische Speicherverwaltung

C# verwendet eine automatische Speicherverwaltung, wodurch der Entwickler keine manuelle Reservierung und keine Freigabe des durch Objekte belegten Speichers mehr vornehmen muss. Die Richtlinien der automatischen Speicherverwaltung werden von einem Garbage Collector implementiert. Der über die Speicherverwaltung gesteuerte Lebenszyklus eines Objekts sieht folgendermaßen aus:

1. Bei der Erstellung wird Speicher für das Objekt belegt, der Konstruktor wird ausgeführt, und das Objekt wird als aktiv angesehen.
2. Falls zukünftig ausgeführter Code mit Ausnahme ausgeführter Destruktoren nicht mehr auf das Objekt bzw. einen Teil davon zugreifen kann, wird das Objekt als nicht mehr benutzt angesehen und für die Zerstörung freigegeben. Der C#-Compiler und der Garbage Collector können anhand einer Codeanalyse feststellen, welche Objektverweise in Zukunft u. U. noch verwendet werden. Wenn es sich bei einer lokalen Variablen, die sich innerhalb des Gültigkeitsbereichs befindet, um den einzig vorhandenen Verweis auf ein Objekt handelt, in einer späteren Ausführung ab dem aktuellen Ausführungspunkt der Prozedur jedoch nie auf diese lokale Variable verwiesen wird, kann der Garbage Collector das Objekt (optional) so behandeln, als wäre es nicht mehr in Gebrauch.
3. Sobald das Objekt für die Zerstörung freigegeben ist, wird nach Ablauf einer nicht näher bestimmten Zeitspanne der Destruktor (§10.13) (sofern vorhanden) für das Objekt ausgeführt. Unter normalen Umständen wird der Destruktor für das Objekt nur einmal ausgeführt, obwohl implementierungsspezifische APIs es möglicherweise zulassen, dass dieses Verhalten überschrieben wird.
4. Falls bei einer späteren Ausführung, einschließlich ausgeführter Destruktoren, nicht auf das Objekt bzw. einen Teil davon zugegriffen werden kann, nachdem der Destruktor für ein Objekt ausgeführt wurde, wird das Objekt als nicht verfügbar angesehen und für die Auflistung freigegeben.
5. Einige Zeit, nachdem das Objekt für die Auflistung freigegeben wurde, wird der dem Objekt zugeordnete Speicher vom Garbage Collector wieder freigegeben.

Der Garbage Collector verwaltet die Informationen über die Objektverwendung und trifft anhand dieser Informationen Entscheidungen über die Speicherverwaltung, z. B. wo ein neu erstelltes Objekt im Speicher abgelegt und wann ein Objekt verschoben wird bzw. wann ein Objekt nicht mehr in Gebrauch oder nicht verfügbar ist.

C# wurde so wie andere Sprachen entworfen, die vom Vorhandensein eines Garbage Collectors ausgehen, d. h., der Garbage Collector kann eine große Anzahl von Speicherverwaltungsrichtlinien implementieren. In C# ist es beispielsweise nicht erforderlich, dass sofort bei der Freigabe von Objekten Destruktoren ausgeführt oder Objekte aufgelistet werden bzw. dass Destruktoren in einer bestimmten Reihenfolge oder in einem bestimmten Thread ausgeführt werden.

Das Verhalten des Garbage Collectors kann bis zu einem gewissen Grad über statische Methoden für die System.GC-Klasse gesteuert werden. Über diese Klasse kann angefordert werden, dass eine Auflistung stattfindet und dass Destruktoren ausgeführt (bzw. nicht ausgeführt) werden usw.

Da der Garbage Collector seine Entscheidungen über die Auflistung von Objekten und das Ausführen von Destruktoren sehr flexibel treffen kann, können sich die Ausgabedaten einer anderen Implementierung von der Ausgabe des folgenden Codes unterscheiden. Das Programm

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("Destruct instance of A");  
 }  
}

class B  
{  
 object Ref;

public B(object o) {  
 Ref = o;  
 }

~B() {  
 Console.WriteLine("Destruct instance of B");  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B(new A());  
 b = null;  
 GC.Collect();  
 GC.WaitForPendingFinalizers();  
 }  
}

erstellt eine Instanz der Klasse A und eine Instanz der Klasse B. Diese Objekte sind für die Garbage Collection freigegeben, wenn der Variablen b der Wert null zugewiesen wird, da danach kein vom Benutzer verfasster Code mehr auf sie zugreifen kann. Ausgegeben werden könnte

Destruct instance of A  
Destruct instance of B

oder

Destruct instance of B  
Destruct instance of A

da seitens der Sprache keine Einschränkungen hinsichtlich der Reihenfolge bestehen, in der die Objekte an die Garbage Collection übergeben werden.

In komplexen Fällen kann die Unterscheidung zwischen „für Zerstörung freigegeben“ und „für die Garbage Collection freigegeben“ von Bedeutung sein. Beispiel:

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("Destruct instance of A");  
 }

public void F() {  
 Console.WriteLine("A.F");  
 Test.RefA = this;  
 }  
}

class B  
{  
 public A Ref;

~B() {  
 Console.WriteLine("Destruct instance of B");  
 Ref.F();  
 }  
}

class Test  
{  
 public static A RefA;  
 public static B RefB;

static void Main() {  
 RefB = new B();  
 RefA = new A();  
 RefB.Ref = RefA;  
 RefB = null;  
 RefA = null;

// A and B now eligible for destruction  
 GC.Collect();  
 GC.WaitForPendingFinalizers();

// B now eligible for collection, but A is not  
 if (RefA != null)  
 Console.WriteLine("RefA is not null");  
 }  
}

Wenn der Garbage Collector im vorangehenden Programm entscheidet, dass der Destruktor von A vor dem Destruktor von B ausgeführt wird, könnte das Programm folgende Daten ausgeben:

Destruct instance of A  
Destruct instance of B  
A.F  
RefA is not null

Beachten Sie, dass die Methoden von A (in diesem Fall F) immer noch von einem anderen Destruktor aufgerufen werden können, obwohl die Instanz von A nicht verwendet und der Destruktor von A ausgeführt wurde. Das Ausführen eines Destruktors kann außerdem dazu führen, dass ein Objekt vom Mainlineprogramm erneut nicht verwendet werden kann. In diesem Fall führt das Ausführen des Destruktors von B dazu, dass der aktive Verweis Test.RefA auf eine Instanz von A, die zuvor nicht in Gebrauch war, zugreifen kann. Nach dem Aufruf von WaitForPendingFinalizers ist die Instanz von B für die Garbage Collection freigegeben. Für die Instanz von A gilt dies aufgrund des Verweises Test.RefA jedoch nicht.

Um Verwirrung und unerwartetes Verhalten zu vermeiden, empfiehlt es sich generell, dass Destruktoren nur die in den objekteigenen Feldern gespeicherten Daten bereinigen und keine Aktionen für referenzierte Objekte oder statische Felder ausführen.

Als Alternative zur Verwendung von Destruktoren kann die Klasse auch die System.IDisposable-Schnittstelle implementieren. Dadurch kann der Client eines Objekts bestimmen, wann die Ressourcen des Objekts freigegeben werden. Dies erfolgt i. d. R. durch Zugriff auf das Objekt als Ressource in einer using-Anweisung (§8.13).

## Ausführungsreihenfolge

Die Ausführung eines C#-Programms geht so vonstatten, dass die Nebeneffekte der einzelnen ausgeführten Threads an kritischen Ausführungspunkten unverändert bleiben. Ein Nebeneffekt wird als Lese- oder Schreibzugriff auf ein flüchtiges Feld, als Schreibzugriff auf eine nicht flüchtige Variable, als Schreibzugriff auf eine externe Ressource und als Auslösen einer Ausnahme definiert. Die kritischen Ausführungspunkte, an denen die Reihenfolge dieser Nebeneffekte beibehalten werden muss, stellen Verweise auf flüchtige Felder (§10.5.3), lock-Anweisungen (§8.12) sowie Threaderstellung und Threadbeendigung dar. In der Ausführungsumgebung kann die Ausführungsreihenfolge eines C#-Programms, abgesehen von folgenden Einschränkungen, beliebig geändert werden:

* Die Datenabhängigkeit bleibt innerhalb eines Ausführungsthreads erhalten. Der Wert der einzelnen Variablen wird also so berechnet, als würden alle Anweisungen im Thread in der ursprünglichen Programmreihenfolge ausgeführt werden.
* Die Regeln für die Reihenfolge der Initialisierung werden beibehalten (§10.5.4 und §10.5.5).
* Die Reihenfolge der Nebeneffekte wird im Hinblick auf die flüchtigen Lese- und Schreibzugriffe (§10.5.3) beibehalten. Darüber hinaus müssen Ausdrücke in der Ausführungsumgebung nicht teilweise ausgewertet werden, falls abgeleitet werden kann, dass der Wert des Ausdrucks nicht in Gebrauch ist und dass keine Nebeneffekte erzeugt werden (einschließlich solcher, die beim Aufrufen einer Methode oder beim Zugriff auf ein flüchtiges Feld auftreten). Wenn die Programmausführung durch ein asynchrones Ereignis (z. B. eine durch einen anderen Thread ausgelöste Ausnahme) unterbrochen wird, kann nicht gewährleistet werden, dass die erkennbaren Nebeneffekte in der ursprünglichen Programmreihenfolge dargestellt werden.

# Typen

Die Typen der Sprache C# sind in zwei Hauptkategorien unterteilt: Werttypen und Verweistypen. Sowohl Werttypen als auch Verweistypen können generische Typen sein, die einen oder mehrere Typparameter akzeptieren. Typparameter können Werttypen und Verweistypen angeben.

type:  
value-type  
reference-type   
type-parameter

Eine dritte Kategorie von Typen und Zeigern ist nur im nicht gesicherten Code verfügbar. Dies wird unter §18.2 genauer beschrieben.

Werttypen unterscheiden sich von Verweistypen darin, dass die Variablen der Werttypen ihre Daten direkt enthalten, während die Variablen der Verweistypen Verweise auf ihre Daten speichern. Letztere werden als Objekte bezeichnet. Bei Verweistypen können zwei Variablen auf dasselbe Objekt verweisen. Somit ist es möglich, dass die eine Variable betreffenden Operationen auch Auswirkungen auf das von der anderen Variablen referenzierte Objekt haben. Bei Werttypen besitzt jede Variable ihre eigene Kopie der Daten. Operationen beeinflussen daher immer nur eine Variable.

Das Typensystem von C# ist dahingehend vereinheitlicht, dass ein Wert beliebigen Typs als ein Objekt behandelt werden kann. Alle Typen in C# sind direkt oder indirekt vom object-Klassentyp abgeleitet, und object ist die oberste Basisklasse für alle Typen. Werte von Verweistypen werden als Objekte behandelt, indem die Werte einfach als object-Typen betrachtet werden. Die Werte der Werttypen werden durch die Ausführung von Boxing- und Unboxing-Vorgängen als Objekte behandelt (§4.3).

## Werttypen

Ein Werttyp ist entweder ein Strukturtyp oder ein Enumerationstyp. C# stellt eine Reihe vordefinierter Strukturtypen bereit, die als einfache Typen bezeichnet werden. Einfache Typen werden durch reservierte Wörter identifiziert.

value-type:  
struct-type  
enum-type

struct-type:  
type-name  
simple-type   
nullable-type

simple-type:  
numeric-type  
bool

numeric-type:  
integral-type  
floating-point-type  
decimal

integral-type:  
sbyte  
byte  
short  
ushort  
int  
uint  
long  
ulong  
char

floating-point-type:  
float  
double

nullable-type:  
non-nullable-value-type ?

non-nullable-value-type:  
type

enum-type:  
type-name

Im Unterschied zu Variablen von Verweistypen können Variablen eines Werttyps den Wert null nur dann enthalten, wenn der Werttyp NULL-Werte zulässt. Es ist zu jedem Werttyp, der keine NULL-Werte zulässt, ein zugehöriger Werttyp vorhanden, der NULL-Werte zulässt und denselben Wertesatz sowie den zusätzlichen null-Wert enthalten kann.

Durch die Zuweisung zu einer Variablen eines Werttyps wird eine Kopie des zugewiesenen Werts erstellt. Bei der Zuweisung zu einer Variablen eines Verweistyps wird hingegen nur der Verweis kopiert und nicht das durch den Verweis gekennzeichnete Objekt.

### Der System.ValueType-Typ

Alle Werttypen werden implizit von der System.ValueType-Klasse vererbt, die ihrerseits von der object-Klasse vererbt wird. Ein Typ kann nicht von einem Werttyp abgeleitet werden. Somit sind Werttypen implizit versiegelt (§10.1.1.2).

Beachten Sie, dass System.ValueType selbst kein value-type ist. Vielmehr handelt es sich um einen class-type, von dem alle value-types automatisch abgeleitet werden.

### Standardkonstruktoren

Alle Werttypen deklarieren implizit einen öffentlichen, parameterlosen Instanzkonstruktor, der als Standardkonstruktor bezeichnet wird. Der Standardkonstruktor gibt eine mit 0 initialisierte Instanz zurück, die den Standardwert für diesen Werttyp darstellt:

* Der Standardwert für alle simple-types ist der von einem aus allen 0-Werten bestehenden Bitmuster erzeugte Wert.
* Für sbyte, byte, short, ushort, int, uint, long und ulong ist der Standardwert 0.
* Der Standardwert für char ist '\x0000'.
* Der Standardwert für float ist 0.0f.
* Der Standardwert für double ist 0.0d.
* Der Standardwert für decimal ist 0.0m.
* Der Standardwert für bool ist false.
* Der Standardwert eines enum-type E ist 0 (null), der in den Typ E konvertiert wird.
* Der Standardwert für einen struct-type wird erzeugt, indem für alle Werttypfelder der Standardwert und für alle Verweistypfelder null festgelegt wird.
* Der Standardwert eines nullable-type ist eine Instanz, für die die HasValue-Eigenschaft false und die Value-Eigenschaft nicht definiert ist. Der Standardwert wird auch als der NULL-Wert des Typs bezeichnet, der NULL-Werte zulässt.

Wie bei allen anderen Instanzkonstruktoren erfolgt der Aufruf des Standardkonstruktors eines Werttyps über den new-Operator. Aus Effizienzgründen ist diese Anforderung nicht dazu gedacht, dass die Implementierung einen Konstruktoraufruf erzeugt. Im folgenden Beispiel werden die Variablen i und j mit 0 initialisiert.

class A  
{  
 void F() {  
 int i = 0;  
 int j = new int();  
 }  
}

Da jeder Werttyp implizit über einen öffentlichen, parameterlosen Konstruktor verfügt, kann ein Strukturtyp keine explizite Deklaration eines parameterlosen Konstruktors enthalten. Ein Strukturtyp kann jedoch parametrisierte Instanzkonstruktoren deklarieren (§11.3.8).

### Strukturtypen

Ein Strukturtyp ist ein Werttyp, der Konstanten, Felder, Methoden, Eigenschaften, Indexer, Operatoren, Instanzkonstruktoren, statische Konstruktoren und geschachtelte Typen deklarieren kann. Die Deklaration von Strukturtypen wird in §11.1 beschrieben.

### Einfache Typen

C# stellt eine Reihe vordefinierter Strukturtypen bereit, die als einfache Typen bezeichnet werden. Die einfachen Typen lassen sich anhand reservierter Wörter erkennen, die jedoch lediglich Aliase vordefinierter Strukturtypen im System-Namespace darstellen (siehe die folgende Tabelle).

|  |  |
| --- | --- |
| **Reserviertes Wort** | **Zugeordneter Typ** |
| sbyte | System.SByte |
| byte | System.Byte |
| short | System.Int16 |
| ushort | System.UInt16 |
| int | System.Int32 |
| uint | System.UInt32 |
| long | System.Int64 |
| ulong | System.UInt64 |
| char | System.Char |
| float | System.Single |
| double | System.Double |
| bool | System.Boolean |
| decimal | System.Decimal |

Da der einfache Typ einen Alias für einen Strukturtyp darstellt, verfügen alle einfachen Typen über Member. Beispielsweise besitzt int die in System.Int32 deklarierten sowie die von System.Object geerbten Member. Somit sind die folgenden Anweisungen zulässig:

int i = int.MaxValue; // System.Int32.MaxValue constant  
string s = i.ToString(); // System.Int32.ToString() instance method  
string t = 123.ToString(); // System.Int32.ToString() instance method

Die einfachen Typen unterscheiden sich von den Strukturtypen darin, dass sie verschiedene zusätzliche Operationen ermöglichen:

* Die meisten einfachen Typen ermöglichen das Erstellen von Werten, indem Literale (literals)(§2.4.4) geschrieben werden. So ist beispielsweise 123 ein Literal vom Typ int und 'a' ein Literal vom Typ char. C# trifft in der Regel keine Vorkehrungen für Literale von Strukturtypen. Nicht standardmäßige Werte anderer Strukturtypen werden letztendlich stets über Instanzkonstruktoren der entsprechenden Strukturtypen erstellt.
* Wenn alle Operanden eines Ausdrucks Konstanten des einfachen Typs sind, kann der Compiler den Ausdruck während der Kompilierung auswerten. Ein solcher Ausdruck wird als konstanter Ausdruck (constant-expression) bezeichnet (§7.19). Ausdrücke mit Operatoren, die von anderen Strukturtypen definiert wurden, werden nicht als konstante Ausdrücke betrachtet.
* Mithilfe von const-Deklarationen können Konstanten der einfachen Typen deklariert werden (§10.4). Konstanten anderer Strukturtypen sind nicht möglich. Mithilfe von static readonly-Feldern wird jedoch ein ähnliches Ergebnis erzeugt.
* Konvertierungen, die einfache Typen verwenden, können in die Auswertung von Konvertierungsoperatoren einbezogen werden, die von anderen Strukturtypen definiert wurden. Ein benutzerdefinierter Konvertierungsoperator kann jedoch nie zum Auswerten eines anderen benutzerdefinierten Operators verwendet werden (§6.4.3).

### Ganzzahlige Typen

C# unterstützt neun Ganzzahltypen: sbyte, byte, short, ushort, int, uint, long, ulong und char. Die Ganzzahltypen haben folgende Größen und Wertebereiche:

* Der sbyte-Typ stellt vorzeichenbehaftete 8-Bit-Ganzzahlen mit Werten zwischen –128 und 127 dar.
* Der byte-Typ stellt vorzeichenlose 8-Bit-Ganzzahlen mit Werten zwischen 0 und 255 dar.
* Der short-Typ stellt vorzeichenbehaftete 16-Bit-Ganzzahlen mit Werten zwischen –32768 und 32767 dar.
* Der ushort-Typ stellt vorzeichenlose 16-Bit-Ganzzahlen mit Werten zwischen 0 und 65535 dar.
* Der int-Typ stellt vorzeichenbehaftete 32-Bit-Ganzzahlen mit Werten zwischen –2147483648 und 2147483647 dar.
* Der uint-Typ stellt vorzeichenlose 32-Bit-Ganzzahlen mit Werten zwischen 0 und 4294967295 dar.
* Der long-Typ stellt vorzeichenbehaftete 64-Bit-Ganzzahlen mit Werten zwischen –9223372036854775808 und 9223372036854775807 dar.
* Der ulong-Typ stellt vorzeichenlose 64-Bit-Ganzzahlen mit Werten zwischen 0 und 18446744073709551615 dar.
* Der char-Typ stellt vorzeichenlose 16-Bit-Ganzzahlen mit Werten zwischen 0 und 65535 dar. Die für den char-Typ möglichen Werte entsprechen dem Unicode-Zeichensatz. Obwohl die Darstellung von char identisch mit der von ushort ist, sind nicht alle Operationen, die für den einen Typ zulässig sind, auch für den anderen zulässig.

Die unären und binären Operatoren der Ganzzahltypen arbeiten immer mit vorzeichenbehafteter 32-Bit-Genauigkeit, vorzeichenloser 32-Bit-Genauigkeit, vorzeichenbehafteter 64-Bit-Genauigkeit oder vorzeichenloser 64-Bit-Genauigkeit:

* Bei den unären Operatoren + und ~ wird der Operand in den Typ T konvertiert, wobei T der erste Typ von int, uint, long und ulong ist, der alle möglichen Werte des Operanden vollständig darstellen kann. Die Operation wird dann mit der Genauigkeit von Typ T ausgeführt, und das Ergebnis ist ebenfalls T.
* Beim unären Operator – wird der Operand in den Typ T konvertiert, wobei T der erste Typ von int und long ist, der alle möglichen Werte des Operanden vollständig darstellen kann. Die Operation wird dann mit der Genauigkeit von Typ T ausgeführt, und das Ergebnis ist ebenfalls T. Der unäre Operator – kann nicht auf Operanden vom Typ ulong angewendet werden.
* Bei den binären Operatoren +, –, \*, /, %, &, ^, |, ==, !=, >, <, >= und <= werden die Operanden in den Typ T konvertiert, wobei T der erste Typ von int, uint, long und ulong ist, der alle möglichen Werte beider Operanden vollständig darstellen kann. Die Operation wird dann mit der Genauigkeit von Typ T ausgeführt, und das Ergebnis ist ebenfalls T (oder bool bei relationalen Operatoren). Es ist bei binären Operatoren nicht möglich, dass ein Operand vom Typ long und der andere vom Typ ulong ist.
* Bei den binären Operatoren << und >> wird der linke Operand in den Typ T konvertiert, wobei T der erste Typ von int, uint, long und ulong ist, der alle möglichen Werte des Operanden vollständig darstellen kann. Die Operation wird dann mit der Genauigkeit von Typ T ausgeführt, und das Ergebnis ist ebenfalls T.

Der char-Typ ist zwar als Ganzzahltyp klassifiziert, unterscheidet sich jedoch in zwei Punkten von den anderen Ganzzahltypen:

* Es finden keine impliziten Konvertierungen aus anderen Datentypen in den char-Typ statt. Obwohl die Typen sbyte, byte und ushort Wertebereiche besitzen, die sich mithilfe des Typs char vollständig darstellen lassen, gibt es keine impliziten Konvertierungen aus sbyte, byte oder ushort in char.
* Konstanten mit dem Typ char müssen als character-literal oder als integer-literal in Verbindung mit einer Typumwandlung in den Typ char geschrieben werden. (char)10 entspricht beispielsweise '\x000A'.

Die Operatoren und Anweisungen checked und unchecked werden zum Steuern der Überlaufprüfung für arithmetische Ganzzahltypoperationen und -konvertierungen verwendet (§7.6.12). In einem checked-Kontext erzeugt ein Überlauf bei der Kompilierung einen Fehler oder löst System.OverflowException aus. In einem unchecked-Kontext werden Überläufe ignoriert, und alle höherwertigen Bits, die nicht in den Zieltyp passen, werden verworfen.

### Gleitkommatypen

C# unterstützt zwei Gleitkommatypen: float und double. Die Typen float und double werden mithilfe der IEEE 754-Formate für einfache Genauigkeit (32-Bit) und doppelte Genauigkeit (64-Bit) dargestellt, wodurch die folgenden Werte möglich sind:

* Positive 0 und negative 0. In den meisten Fällen verhalten sich positive und negative 0 identisch, d. h. wie der einfache Wert 0. Allerdings gibt es bei bestimmten Operationen auch Unterschiede (§7.8.2).
* Positive Unendlichkeit und negative Unendlichkeit. Unendlichkeiten werden durch Operationen wie die Division einer von 0 verschiedenen Zahl durch 0 erzeugt. Beispielsweise ergibt 1.0 / 0.0 positive Unendlichkeit und –1.0 / 0.0 negative Unendlichkeit.
* Der Wert Not-a-Number-Number. Dieser Wert wird häufig mit NaN abgekürzt. NaNs werden durch ungültige Gleitkommaoperationen erzeugt, z. B. die Division von 0 durch 0.
* Die endliche Menge der von 0 verschiedenen Werte der s × m × 2e, wobei s den Wert 1 oder −1 hat und m sowie e durch den entsprechenden Gleitkommatyp bestimmt werden: Für float, 0 < m < 224 und −149 ≤ e ≤ 104 sowie für double, 0 < m < 253 und −1075 ≤ e ≤ 970. Denormalisierte Gleitkommazahlen werden als gültige Werte ungleich 0 betrachtet.

Der float-Typ kann Werte im Bereich von ungefähr 1,5 × 10−45 bis 3,4 × 1038 mit einer Genauigkeit von 7 Stellen darstellen.

Der double-Typ kann Werte im Bereich von ungefähr 5,0 × 10−324 bis 1,7 × 10308 mit einer Genauigkeit von 15-16 Stellen darstellen.

Wenn einer der Operanden eines binären Operators vom Gleitkommatyp ist, muss der andere Operand vom Ganzzahl- oder Gleitkommatyp sein. Die Operation wird folgendermaßen ausgewertet:

* Wenn ein Operand vom Ganzzahltyp ist, wird dieser Operand in den Gleitkommatyp des anderen Operanden konvertiert.
* Wenn einer der Operanden vom Typ double ist, wird der andere Operand in double konvertiert. Die Operation wird mindestens mit dem Wertebereich und der Präzision von double ausgeführt. Der Ergebnistyp ist double (oder bool bei relationalen Operatoren).
* Andernfalls wird die Operation mindestens mit dem Wertebereich und der Genauigkeit von float ausgeführt, und das Ergebnis ist float (oder bool bei relationalen Operatoren).

Die Gleitkommaoperatoren, einschließlich der Zuweisungsoperatoren, erzeugen niemals Ausnahmen. Stattdessen erzeugen Gleitkommaoperationen, wie im Folgenden erklärt, in Ausnahmesituationen 0, Unendlichkeit oder NaN:

* Wenn das Ergebnis einer Gleitkommaoperation für das Zielformat zu klein ist, ist das Ergebnis der Operation positiv oder negativ 0.
* Wenn das Ergebnis einer Gleitkommaoperation für das Zielformat zu groß ist, ist das Ergebnis der Operation positive oder negative Unendlichkeit.
* Wenn eine Gleitkommaoperation ungültig ist, ist das Ergebnis der Operation NaN.
* Wenn mindestens einer der beiden Operanden einer Gleitkommaoperation NaN ist, ist das Ergebnis der Operation NaN.

Gleitkommaoperationen können mit einer höheren Genauigkeit als der Ergebnistyp der Operation ausgeführt werden. Manche Hardwarearchitekturen unterstützen z. B. einen "erweiterten" oder "long double"-Gleitkommatyp mit einem größeren Bereich und einer höheren Genauigkeit als der Typ double und führen alle Gleitkommaoperationen implizit mit diesem höheren Genauigkeitstyp aus. Nur unter großen Leistungsverlusten können solche Hardwarearchitekturen Gleitkommaoperationen mit geringerer Genauigkeit ausführen. Anstatt eine Implementierung zu verlangen, die Einbußen hinsichtlich Leistung und Genauigkeit mit sich bringt, ermöglicht C# die Verwendung eines Typs mit höherer Genauigkeit für alle Gleitkommaoperationen. Der einzige messbare Effekt dieser Variante besteht in der Ausgabe genauerer Ergebnisse. In Ausdrücken der Form x \* y / z, in denen die Multiplikation zu Ergebnissen außerhalb des Bereichs von double, die anschließende Division jedoch zu temporären Ergebnissen innerhalb des Bereichs von double führt, kann die Tatsache, dass der Ausdruck in einem höheren Bereichsformat ausgewertet wird, anstelle einer Unendlichkeit zu endlichen Ergebnissen führen.

### Der decimal-Typ

Der decimal-Typ ist ein 128-Bit-Datentyp, der sich insbesondere für Finanz- und Währungskalkulationen eignet. Der decimal-Typ kann Werte im Bereich von 1,0 × 10−28 bis ungefähr 7,9 × 1028 mit 28-29 signifikanten Stellen darstellen.

Die endliche Menge der Werte mit dem Typ decimal hat die Form (–1)s× c × 10-e, wobei das Vorzeichen s 0 oder 1 lautet, der Koeffizient c durch 0 ≤ c < 296 angegeben wird und die Dezimalstellenanzahl e 0 ≤ e ≤ 28 entspricht. Der decimal-Typ bietet keine Unterstützung für Nullen (0) mit Vorzeichen, unendliche Zahlen oder NaNs (Not a Number). Ein decimal-Wert wird als 96-Bit-Ganzzahl dargestellt, die mit einer Potenz zur Basis 10 skaliert wird. Bei decimal-Werten mit einem absoluten Wert kleiner als 1.0m reicht die Genauigkeit des Werts bis zur 28. Dezimalstelle, jedoch nicht darüber hinaus. Bei decimal-Werten mit einem absoluten Wert größer oder gleich 1.0m ist der Wert bis auf 28 oder 29 Stellen genau. Im Gegensatz zu den Datentypen float und double werden gebrochene Dezimalzahlen (z. B. 0,1) in der decimal-Darstellung präzise angegeben. Bei der float- und double-Darstellung werden solche Zahlen oft als unendliche Brüche angegeben, wodurch die Darstellung anfälliger für Rundungsfehler wird.

Wenn einer der Operanden eines binären Operators vom Typ decimal ist, muss der andere Operand vom Ganzzahltyp oder vom Typ decimal sein. Wenn ein Ganzzahloperand vorhanden ist, wird er vor der Ausführung der Operation in decimal konvertiert.

Das Ergebnis einer Operation mit Werten vom Typ decimal ist mit dem Ergebnis identisch, das sich aus der Berechnung eines genauen Ergebnisses (gleicher Exponent, nach der Definition für den jeweiligen Operator) ergibt, das dann entsprechend gerundet wird. Ergebnisse werden auf den nächsten darstellbaren Wert gerundet. Wenn zwei darstellbare Werte gleich nah sind, wird auf den Wert mit einer geraden Zahl an der am wenigsten signifikanten Ziffernposition gerundet (kaufmännische Rundung). Ein Ergebnis von 0 weist stets das Zeichen 0 und einen Exponenten von 0 auf.

Wenn durch eine arithmetische Dezimaloperation ein Wert kleiner oder gleich 5 × 10-29 absolut erzeugt wird, gibt die Operation als Ergebnis 0 zurück. Wenn eine arithmetische decimal-Operation zu einem Wert führt, der zu groß für das decimal-Format ist, wird eine System.OverflowException ausgelöst.

Der decimal-Typ besitzt eine größere Genauigkeit, aber einen kleineren Wertebereich als die Gleitkommatypen. Folglich können Konvertierungen von Gleitkommatypen in den decimal-Typ Überlaufausnahmen erzeugen und Konvertierungen vom decimal-Typ in die Gleitkommatypen zu einem Verlust an Genauigkeit führen. Aus diesen Gründen sind keine impliziten Konvertierungen zwischen den Gleitkommatypen und dem decimal-Typ vorhanden, und es ist ohne explizite Umwandlungen nicht möglich, Gleitkommatypen und den decimal-Typ im selben Ausdruck zu vereinen.

### Der bool-Typ

Mit dem bool-Typ werden boolesche logische Mengen dargestellt. Die für den bool-Typ zulässigen Werte sind true und false.

Zwischen bool und den anderen Typen sind keine standardmäßigen Konvertierungen vorhanden. Insbesondere unterscheidet sich der bool-Typ deutlich von den Ganzzahltypen. Ein bool-Wert kann nicht anstelle eines ganzzahligen Werts verwendet werden. Diese Einschränkung gilt auch in umgekehrter Richtung.

In den Sprachen C und C++ kann eine ganze Zahl oder Gleitkommazahl mit dem Wert 0 bzw. ein NULL-Zeiger in den booleschen Wert false und eine ganze Zahl oder Gleitkommazahl mit einem Wert ungleich 0 oder ein Nicht-NULL-Zeiger in den booleschen Wert true konvertiert werden. In C# erfolgen solche Konvertierungen über den expliziten Vergleich eines ganzzahligen Werts oder eines Gleitkommawerts mit 0 oder eines Objektverweises mit null.

### Enumerationstypen

Ein Enumerationstyp ist ein eigenständiger Type mit einer benannten Konstante. Jeder Enumerationstyp besitzt einen zugrunde liegenden Typ, der vom Typ byte, sbyte, short, ushort, int, uint, long oder ulong sein muss. Die Wertegruppe des Enumerationstyps ist mit der Wertegruppe des zugrunde liegenden Typs identisch. Die Werte des Enumerationstyps sind nicht auf die Werte der benannten Konstanten beschränkt. Enumerationstypen werden über Enumerationsdeklarationen definiert (§14.1).

### Typen, die NULL-Werte zulassen

Ein Typ, der NULL-Werte zulässt, kann alle Werte seines zugrunde liegenden Typs und zusätzlich den NULL-Wert darstellen. Ein Typ, der NULL-Werte zulässt, wird durch T? dargestellt, wobei T der zugrunde liegende Typ ist. Diese Syntax ist eine Abkürzung für System.Nullable<T>, die beiden Schreibweisen können gleichbedeutend verwendet werden.

Ein Werttyp, der keine NULL-Werte zulässt, ist hingegen jeder Werttyp außer System.Nullable<T> und dessen Abkürzung T? (für jedes T) sowie jeder Typparameter, der auf einen Werttyp, der keine NULL-Werte zulässt, beschränkt ist (d. h. jeder Typparameter mit einer struct-Einschränkung). Der System.Nullable<T>-Typ gibt die Werttypeinschränkung für T (§10.1.5) an, d. h., dass der zugrunde liegende Typ eines Typs, der NULL-Werte zulässt, jeder Werttyp sein kann, der keine NULL-Werte zulässt. Der zugrunde liegende Typ eines Typs, der NULL-Werte zulässt, darf kein Typ, der NULL-Werte zulässt, und kein Verweistyp sein. int?? und string? sind z. B. ungültige Typen.

Eine Instanz eines Typs T?, der NULL-Werte zulässt, verfügt über zwei öffentliche schreibgeschützte Eigenschaften:

* Eine HasValue-Eigenschaft vom Typ bool
* Eine Value-Eigenschaft vom Typ T

Eine Instanz, für die HasValue den Wert true hat, gilt als nicht NULL. Eine Instanz, die keine NULL-Werte zulässt, enthält einen bekannten Wert, und Value gibt diesen Wert zurück.

Eine Instanz, für die HasValue den Wert false hat, gilt als NULL. Eine NULL-Instanz verfügt über einen nicht definierten Wert. Wenn versucht wird, den Value einer NULL-Instanz zu lesen, wird eine System.InvalidOperationException ausgelöst. Der Zugriff auf die Value-Eigenschaft einer Instanz, die NULL-Werte zulässt, wird als Entwrappen bezeichnet.

Neben dem Standardkonstruktor verfügt jeder Typ T?, der NULL-Werte zulässt, über einen öffentlichen Konstruktor, der ein einzelnes Argument vom Typ T akzeptiert. Mit einem Wert x vom Typ T erstellt ein Konstruktoraufruf der Form

new T?(x)

eine Instanz von T?, die keine NULL-Werte zulässt und für die die Value-Eigenschaft x ist. Das Erstellen einer Instanz, die keine NULL-Werte zulässt, eines Typs, der NULL-Werte zulässt, für einen bestimmten Wert wird als Wrapping bezeichnet.

Implizite Konvertierung des null-Literals in T? (§6.1.5) und von T in T? (§6.1.4) sind möglich.

## Verweistypen

Ein Verweistyp ist ein Klassen-, Schnittstellen-, Array- oder Delegattyp.

reference-type:  
class-type  
interface-type  
array-type  
delegate-type

class-type:  
type-name  
object  
dynamic  
string

interface-type:  
type-name

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

delegate-type:  
type-name

Ein Verweistypwert ist ein Verweis auf eine Instanz des Typs, die auch als Objekt bezeichnet wird. Der Spezialwert null ist mit allen Verweistypen kompatibel und zeigt das Nichtvorhandensein einer Instanz an.

### Klassentypen

Ein Klassentyp definiert eine Datenstruktur, in der Datenmember (Konstanten und Felder), Funktionsmember (Methoden, Eigenschaften, Ereignisse, Indexer, Operatoren, Instanzkonstruktoren, Destruktoren und statische Konstruktoren) sowie geschachtelte Typen enthalten sind. Klassentypen unterstützen die Vererbung, sodass Basisklassen durch abgeleitete Klassen erweitert und spezialisiert werden können. Instanzen von Klassentypen werden mithilfe von object-creation-expressions (§7.6.10.1) generiert.

Klassentypen werden in §10 beschrieben.

Bestimmte vordefinierte Klassentypen haben in der Sprache C# eine besondere Bedeutung, die Sie der folgenden Tabelle entnehmen können.

|  |  |
| --- | --- |
| **Klassentyp** | **Beschreibung** |
| System.Object | Die oberste Basisklasse aller anderen Typen. Siehe §4.2.2. |
| System.String | Der Zeichenfolgentyp der Sprache C#. Siehe §4.2.4. |
| System.ValueType | Die Basisklasse aller Werttypen. Siehe §4.1.1. |
| System.Enum | Die Basisklasse aller Enumerationstypen. Siehe §14. |
| System.Array | Die Basisklasse aller Arraytypen. Siehe §12. |
| System.Delegate | Die Basisklasse aller Delegattypen. Siehe §15. |
| System.Exception | Die Basisklasse aller Ausnahmetypen. Siehe §16. |

### Der object-Typ

Der object-Klassentyp ist die oberste Basisklasse für alle anderen Typen. Jeder Typ in C# ist direkt oder indirekt vom object-Klassentyp abgeleitet.

Das object-Schlüsselwort ist lediglich ein Alias für die vordefinierte System.Object-Klasse.

### Der dynamische Typ

Der Typ dynamic kann, ähnlich wie object, auf ein beliebiges Objekt verweisen. Wenn Operatoren auf Ausdrücke vom Typ dynamic angewendet werden, wird ihre Auflösung verschoben, bis das Programm ausgeführt wird. Wenn daher der Operator nicht auf das Objekt, auf das verwiesen wird, angewendet werden darf, wird während der Kompilierung kein Fehler zurückgegeben. Stattdessen wird eine Ausnahme ausgelöst, wenn bei der Auflösung des Operators zur Laufzeit ein Fehler auftritt.

Eine ausführlichere Beschreibung des dynamischen Typs finden Sie unter §4.7, eine Beschreibung der dynamischen Bindung unter §7.2.2.

### Der string-Typ

Der string-Typ ist ein versiegelter Klassentyp, der direkt von object erbt. Mit Instanzen der string-Klasse werden Unicode-Zeichenfolgen dargestellt.

Werte des string-Typs können als Zeichenfolgenliterale (§2.4.4.5) geschrieben werden.

Das string-Schlüsselwort ist lediglich ein Alias für die vordefinierte System.String-Klasse.

### Schnittstellentypen

Eine Schnittstelle definiert einen Vertrag. Eine Klasse oder Struktur, die eine Schnittstelle implementiert, muss ihren Vertrag einhalten. Eine Schnittstelle kann von mehreren Basisschnittstellen erben. Klassen oder Strukturen können jeweils mehrere Schnittstellen implementieren.

Schnittstellentypen werden in §13 beschrieben.

### Arraytypen

Ein Array ist eine Datenstruktur mit 0 oder mehr Variablen, auf die über berechnete Indizes zugegriffen wird. Die in einem Array enthaltenen Variablen, die auch als Arrayelemente bezeichnet werden, sind alle vom selben Typ. Dieser Typ wird als Elementtyp des Arrays bezeichnet.

Arraytypen werden in §12 beschrieben.

### Delegattypen

Ein Delegat ist eine Datenstruktur, die auf eine oder mehrere Methoden verweist. Im Fall von Instanzmethoden verweist sie auch auf die entsprechenden Objektinstanzen.

In C und C++ kommt der Funktionszeiger einem Delegaten am nächsten. Allerdings kann der Funktionszeiger nur auf statische Funktionen verweisen, während ein Delegat sowohl auf statische als auch auf Instanzmethoden verweisen kann. Bei Instanzmethoden speichert der Delegat nicht nur einen Verweis auf den Einsprungpunkt der Methode, sondern auch einen Verweis auf die Objektinstanz, für die die Methode aufgerufen werden soll.

Delegattypen werden in §15 beschrieben.

## Boxing und Unboxing

Die Begriffe „Boxing“ und „Unboxing“ sind für das Typsystem von C# von zentraler Bedeutung. Beide Prozesse bilden die Voraussetzung dafür, dass jeder Wert eines value-type in und aus dem Typ object konvertiert werden kann, und stellen somit eine Brücke zwischen value-types und reference-types dar. Boxing und Unboxing ermöglichen eine vereinheitlichte Sicht auf das Typensystem, da durch sie ein Wert beliebigen Typs letztlich immer als ein Objekt behandelt werden kann.

### Boxingkonvertierungen

Eine Boxingkonvertierung ermöglicht die implizite Konvertierung eines value-type in einen reference-type. Folgende Boxingkonvertierungen sind möglich:

* Aus einem beliebigen value-type in den Typ object.
* Aus einem beliebigen value-type in den Typ System.ValueType.
* Aus einem beliebigen non-nullable-value-type in einen beliebigen interface-type der vom value-type implementiert wird.
* Aus einem beliebigen nullable-type in einen interface-type, der vom zugrunde liegenden Typ des nullable-type implementiert wird.
* Aus einem beliebigen enum-type in den Typ System.Enum.
* Aus einem beliebigen nullable-type mit einem zugrunde liegenden enum-type in den Typ System.Enum.

Beachten Sie, dass eine implizite Konvertierung aus einem Typparameter als Boxingkonvertierung ausgeführt wird, wenn zur Laufzeit eine Konvertierung aus einem Werttyp in einen Verweistyp durchgeführt wird (§6.1.10).

Das Boxing des Werts eines non-nullable-value-type besteht aus der Reservierung einer Objektinstanz und dem Kopieren des non-nullable-value-type-Werts in diese Instanz.

Das Boxing des Werts eines nullable-type führt zu einem NULL-Verweis, wenn es sich dabei um den null-Wert handelt (HasValue ist false), oder zum Ergebnis des Entwrappens und Boxing des zugrunde liegenden Werts.

Der eigentliche Vorgang beim Boxing eines Werts mit non-nullable-value-type lässt sich am besten anhand einer generischen Boxingklasse erklären, deren Verhalten folgender Deklaration ähnelt.

sealed class Box<T>: System.ValueType  
{  
 T value;

public Box(T t) {  
 value = t;  
 }  
}

Das Boxing eines Werts v vom Typ T besteht jetzt aus der Ausführung des Ausdrucks new Box<T>(v) und der Rückgabe der resultierenden Instanz als einen Wert vom Typ object. Folglich würden die Anweisungen

int i = 123;  
object box = i;

vom Konzept her folgenden Anweisungen entsprechen

int i = 123;  
object box = new Box<int>(i);

Eine Boxingklasse, wie die vorangehenden Box<T>-Klasse, ist nicht wirklich existent, und der dynamische Typ eines mittels Boxing gepackten Werts ist kein tatsächlicher Klassentyp. Stattdessen besitzt ein mittels Boxing gepackter Wert vom Typ T den dynamischen Typ T, und eine dynamische Typüberprüfung kann mithilfe des is-Operators einfach auf Typ T verweisen. Beispiel:

int i = 123;  
object box = i;  
if (box is int) {  
 Console.Write("Box contains an int");  
}

führt zur Ausgabe der Zeichenfolge "Box contains an int" auf der Konsole.

Eine Boxingkonvertierung impliziert das Erstellen einer Kopie des mittels Boxing gepackten Werts. Dieser Vorgang unterscheidet sich von der Konvertierung eines reference-type in den Typ object, bei der der Wert weiterhin auf dieselbe Instanz verweist und lediglich als der weniger abgeleitete Typ object betrachtet wird. Bei den Deklarationen

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

geben die folgenden Anweisungen

Point p = new Point(10, 10);  
object box = p;  
p.x = 20;  
Console.Write(((Point)box).x);

den Wert 10 auf der Konsole aus, da bei der mit der Zuweisung von p auf box verbundenen impliziten Boxingoperation der Wert von p kopiert wird. Wenn Point dagegen eine class deklariert hätte, wäre der Wert 20 ausgegeben worden, da p und box auf dieselbe Instanz verweisen würden.

### Unboxingkonvertierungen

Eine Unboxingkonvertierung ermöglicht die explizite Konvertierung eines reference-type in einen value-type. Folgende Unboxingkonvertierungen sind möglich:

* Aus einem beliebigen value-type in den Typ object.
* Aus dem Typ System.ValueType in einen beliebigen value-type.
* Aus einem beliebigen interface-type in einen beliebigen non-nullable-value-type, der den interface-type implementiert.
* Aus einem beliebigen interface-type in einen beliebigen nullable-type, dessen zugrunde liegender Typ den interface-type implementiert.
* Vom Typ System.Enum in einen beliebigen enum-type.
* Aus dem Typ System.Enum in einen beliebigen nullable-type mit einem zugrunde liegenden enum-type.

Beachten Sie, dass eine explizite Konvertierung in einen Typparameter als Unboxingkonvertierung ausgeführt wird, wenn zur Laufzeit eine Konvertierung aus einem Verweistyp in einen Werttyp durchgeführt wird (§6.2.6).

Bei einem Unboxingvorgang in einen non-nullable-value-type wird zuerst überprüft, ob die Objektinstanz ein mittels Boxing gepackter Wert des angegebenen non-nullable-value-type ist, und anschließend wird der Wert aus der Instanz kopiert.

Das Unboxing in einen nullable-type ergibt einen NULL-Wert des nullable-type, wenn der Quelloperand null ist, oder andernfalls das mittels Wrapping gepackte Ergebnis des Unboxing der Objektinstanz in den zugrunde liegenden Typ des nullable-type.

Mit Blick auf die im vorigen Abschnitt beschriebene fiktive Boxingklasse besteht die Unboxingkonvertierung eines Objekts box in einen value-type T in der Ausführung des Ausdrucks ((Box<T>)box).value. Folglich würden die Anweisungen

object box = 123;  
int i = (int)box;

vom Konzept her folgenden Anweisungen entsprechen

object box = new Box<int>(123);  
int i = ((Box<int>)box).value;

Damit eine Unboxingkonvertierung in einen bestimmten non-nullable-value-type zur Laufzeit erfolgreich verläuft, muss der Wert des Quelloperanden ein Verweis auf einen durch Boxing erstellten Wert dieses non-nullable-value-type sein. Wenn der Quelloperand den Wert null hat, wird eine System.NullReferenceException ausgelöst. Wenn der Quelloperand einen Verweis auf ein inkompatibles Objekt darstellt, wird eine System.InvalidCastException ausgelöst.

Damit eine Unboxingkonvertierung in einen bestimmten nullable-type zur Laufzeit erfolgreich verläuft, muss der Wert des Quelloperanden entweder null oder ein Verweis auf einen durch Boxing erstellten Wert des zugrunde liegenden non-nullable-value-type des nullable-type sein. Wenn der Quelloperand einen Verweis auf ein inkompatibles Objekt darstellt, wird eine System.InvalidCastException ausgelöst.

## Konstruierte Typen

Eine generische Typdeklaration selbst gibt einen ungebundenen generischen Typ an, der als Schablone für das Erstellen vieler verschiedener Typen verwendet wird, indem Typargumente angewendet werden. Die Typargumente werden in spitzen Klammern (< und >) geschrieben, direkt gefolgt vom Namen des generischen Typs. Ein mindestens ein Typargument enthaltender Typ wird als konstruierter Typ bezeichnet. Ein konstruierter Typ kann an den meisten Stellen innerhalb der Sprache verwendet werden, an denen ein Typname auftreten kann. Ein ungebundener generischer Typ kann nur in einer typeof-expression (§7.6.11) verwendet werden.

Konstruierte Typen können auch in Ausdrücken als einfache Namen (§7.6.2) oder beim Zugriff auf einen Member (§7.6.4) verwendet werden.

Wenn ein namespace-or-type-name ausgewertet wird, werden nur generische Typen mit der richtigen Anzahl von Typparametern berücksichtigt. Daher können Sie denselben Bezeichner für unterschiedliche Typen verwenden, sofern diese über eine unterschiedliche Anzahl von Typparametern verfügen. Dies ist hilfreich, wenn Sie gleichzeitig generische und nicht generische Klassen in einem Programm verwenden:

namespace Widgets  
{  
 class Queue {...}  
 class Queue<TElement> {...}  
}

namespace MyApplication  
{  
 using Widgets;

class X  
 {  
 Queue q1; // Non-generic Widgets.Queue  
 Queue<int> q2; // Generic Widgets.Queue  
 }  
}

Ein type-name kann einen konstruierten Typ bezeichnen, obwohl er Typparameter nicht direkt angibt. Dies ist möglich, wenn ein Typ in einer generischen Klassendeklaration geschachtelt ist und der Instanztyp der enthaltenden Deklaration implizit für die Namenssuche verwendet wird (§10.3.8.6):

class Outer<T>  
{  
 public class Inner {...}

public Inner i; // Type of i is Outer<T>.Inner  
}

In unsicherem Code kann ein konstruierter Typ nicht als unmanaged-type (§18.2) verwendet werden.

### Typargumente

Jedes Argument in einer Liste von Typargumenten ist einfach ein type.

type-argument-list:  
< type-arguments >

type-arguments:  
type-argument  
type-arguments , type-argument

type-argument:  
type

In unsicherem Code (§18) darf ein type-argument kein Zeigertyp sein. Jedes Typargument muss alle Einschränkungen des zugehörigen Typparameters erfüllen (§10.1.5).

### Offene und geschlossene Typen

Alle Typen können als offene Typen oder geschlossene Typen klassifiziert werden. Ein offener Typ ist ein Typ mit Typparametern. Genauer gesagt:

* Ein Typparameter definiert einen offenen Typ.
* Ein Arraytyp ist dann und nur dann ein offener Typ, wenn sein Elementtyp ein offener Typ ist.
* Ein konstruierter Typ ist dann und nur dann ein offener Typ, wenn eines oder mehrere seiner Typargumente einen offenen Typ besitzen. Ein konstruierter geschachtelter Typ ist dann und nur dann ein offener Typ, wenn eines oder mehrere seiner Typargumente der enthaltenen Typen einen offenen Typ besitzen.

Ein geschlossener Typ ist ein Typ, der kein offener Typ ist.

Zur Laufzeit wird der gesamte Code innerhalb einer generischen Typdeklaration im Kontext eines geschlossen konstruierten Typs ausgeführt, der mithilfe der Typargumente der generischen Deklaration erstellt wurde. Jeder Typparameter im generischen Typ ist an einen bestimmten Laufzeittyp gebunden. Die Verarbeitung aller Anweisungen und Ausdrücke zur Laufzeit tritt immer mit geschlossenen Typen auf, während offene Typen nur bei der Verarbeitung während der Kompilierung auftreten.

Jeder geschlossen konstruierte Typ verfügt über einen eigenen Satz von statischen Variablen, die nicht mit anderen geschlossen konstruierten Typen gemeinsam verwendet werden. Da ein offener Typ zur Laufzeit nicht vorhanden ist, gibt es keine statischen Variablen, die einem offenen Typ zugeordnet sind. Zwei geschlossen konstruierte Typen sind von demselben Typ, wenn Sie aus demselben ungebundenen generischen Typ konstruiert wurden und ihre zugehörigen Typargumente von demselben Typ sind.

### Gebundene und ungebundene Typen

Der Begriff ungebundener Typ bezeichnet einen nicht generischen Typ oder einen ungebundenen generischen Typ. Der Begriff gebundener Typ bezeichnet einen nicht generischen Typ oder einen konstruierten Typ.

Ein ungebundener Typ verweist auf die Entität, die durch eine Typdeklaration deklariert wird. Ein ungebundener generischer Typ ist selbst kein Typ und kann nicht als Typ für eine Variable, ein Argument oder einen Rückgabewert bzw. als Basistyp verwendet werden. Ein Verweis auf einen ungebundenen generischen Typ ist lediglich im typeof-Ausdruck (§7.6.11) möglich.

### Erfüllen von Einschränkungen

Bei Verwendung eines konstruierten Typs oder einer generischen Methode wird das übergebene Typargument zur Kompilierzeit auf die Einschränkungen für den Typparameter hin überprüft, die für den generischen Typ oder die generische Methode (§10.1.5) deklariert wurden. Für jede where-Klausel wird das Typargument A, das dem benannten Typparameter entspricht, wie folgt auf jede Einschränkung hin überprüft:

* Wenn die Einschränkung ein Klassentyp, Schnittstellentyp oder Typparameter ist, sei C diese Einschränkung, bei der alle in der Einschränkung vorkommenden Typparameter durch die übergebenen Typargumente ersetzt werden. Um die Einschränkung zu erfüllen, muss der Typ A auf eine der folgenden Weisen in den Typ C konvertiert werden können:
* Durch Identitätskonvertierung (§6.1.1).
* Durch implizite Verweiskonvertierung (§6.1.6).
* Durch Boxingkonvertierung (§6.1.7) unter der Voraussetzung, dass der Typ A ein Werttyp ist, der keine NULL-Werte zulässt.
* Durch implizite Verweis-, Boxing- oder Typparameterkonvertierung eines Typparameters A in C.
* Wenn es sich bei der Einschränkung um die Verweistypeinschränkung (class) handelt, muss der Typ A eine der folgenden Bedingungen erfüllen:
* A ist ein Schnittstellentyp, Klassentyp, Delegattyp oder Arraytyp. Beachten Sie, dass System.ValueType und System.Enum Verweistypen sind, die diese Einschränkung erfüllen.
* A ist ein Typparameter, von dem bekannt ist, dass er ein Verweistyp ist (§10.1.5).
* Wenn es sich bei der Einschränkung um die Werttypeinschränkung (struct) handelt, muss der Typ A eine der folgenden Bedingungen erfüllen:
* A ist ein Strukturtyp oder Enumerationstyp, aber kein Typ, der NULL-Werte zulässt. Beachten Sie, dass System.ValueType und System.Enum Verweistypen sind, die diese Einschränkung nicht erfüllen.
* A ist ein Typparameter, der die Werttypeinschränkung besitzt (§10.1.5).
* Wenn es sich bei der Einschränkung um die Konstruktoreinschränkung new() handelt, darf der Typ A nicht abstract sein und muss einen öffentlichen Konstruktor ohne Parameter besitzen. Diese Bedingung gilt in folgenden Fällen als erfüllt:
* A ist ein Werttyp, da alle Werttypen einen öffentlichen Standardkonstruktor besitzen (§4.1.2).
* A ist ein Typparameter, der die Konstruktoreinschränkung besitzt (§10.1.5).
* A ist ein Typparameter, der die Werttypeinschränkung besitzt (§10.1.5).
* A ist eine Klasse, die nicht abstract ist und einen explizit deklarierten public-Konstruktor ohne Parameter besitzt.
* A ist nicht abstract und verfügt über einen Standardkonstruktor (§10.11.4).

Wenn mindestens eine der Einschränkungen eines Typparameters von den angegebenen Typargumenten nicht erfüllt wird, tritt ein Kompilierungsfehler auf.

Da Typparameter nicht vererbt werden, werden auch Einschränkungen nicht vererbt. Im folgenden Beispiel muss D die Einschränkung für seinen Typparameter T angeben, damit T die von seiner Basisklasse B<T> aufgestellte Einschränkung erfüllt. Die Klasse E muss hingegen keine Einschränkung angeben, denn List<T> implementiert IEnumerable für alle T.

class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}

## Typparameter

Ein Typparameter ist ein Bezeichner, der einen Werttyp oder Verweistyp festlegt, an den der Parameter zur Laufzeit gebunden wird .

type-parameter:  
identifier

Da ein Typparameter mit vielen verschiedenen tatsächlichen Typargumenten instanziiert werden kann, verfügen Typparameter über andere Operationen und Einschränkungen als andere Typen. Dazu gehören:

* Ein Typparameter kann nicht direkt zum Deklarieren einer Basisklasse (§10.2.4) oder von Schnittstellen verwendet werden (§13.1.3).
* Die Regeln für die Membersuche nach Typparametern sind von den für den Typparameter gültigen Einschränkungen abhängig, sofern solche vorhanden sind. Diese werden in §7.4 genauer beschrieben.
* Die verfügbaren Umwandlungen für einen Typparameter sind von den für den Typparameter gültigen Einschränkungen abhängig, sofern solche vorhanden sind. Diese werden in §6.1.10 und §6.2.6 genauer beschrieben.
* Der Literalwert null kann nur dann in einen von einem Typparameter angegeben Typ konvertiert werden, wenn der Typparameter ein Verweistyp (§6.1.10) ist. Stattdessen kann aber ein default-Ausdruck (§7.6.13) verwendet werden. Außerdem kann ein Wert mit einem durch einen Typparameter angegebenen Typ mithilfe von == und != (§7.10.6) mit null verglichen werden, sofern der Typparameter keine Einschränkung für den Werttyp besitzt.
* Ein new-Ausdruck (§7.6.10.1) kann nur dann mit einem Typparameter verwendet werden, wenn dieser eine constructor-constraint oder die Einschränkung für den Werttyp (§10.1.5) besitzt.
* Typparameter können nicht innerhalb eines Attributs verwendet werden.
* Typparameter können nicht beim Memberzugriff (§7.6.4) oder in einem Typnamen (§3.8) zum Bezeichnen eines statischen Members oder eines geschachtelten Typs verwendet werden.
* In unsicherem Code kann ein Typparameter nicht als unmanaged-type (§18.2) verwendet werden.

Als Typ sind Typparameter ein reines Konstrukt zur Kompilierungszeit. Zur Laufzeit wird jeder Typparameter an einen Laufzeittyp gebunden, der durch Bereitstellung eines Typarguments in der generischen Typdeklaration angegeben wurde. Daher ist der Typ einer mit einem Typparameter deklarierten Variablen zur Laufzeit ein geschlossen konstruierter Typ (§4.4.2). Bei der Ausführung aller Anweisungen und Ausdrücke für Typparameter zur Laufzeit wird der tatsächliche Typ verwendet, der als Typargument für den Parameter bereitgestellt wurde.

## Ausdrucksbaumstrukturtypen

Ausdrucksbaumstrukturen ermöglichen die Darstellung von Lambda-Ausdrücken als Datenstrukturen anstelle von ausführbarem Code. Ausdrucksbaumstrukturen sind Werte von Ausdrucksbaumstrukturtypen der Form System.Linq.Expressions.Expression<D>, wobei D ein beliebiger Delegattyp ist. Im verbleibenden Teil dieser Spezifikation wird auf diese Typen mithilfe der Abkürzung Expression<D> verwiesen.

Wenn eine Konvertierung von einem Lambda-Ausdruck in einen Delegattyp D vorhanden ist, existiert auch eine Konvertierung in den Ausdrucksbaumstrukturtyp Expression<D>. Während bei der Konvertierung eines Lambda-Ausdrucks in einen Delegattyp ein Delegat erstellt wird, der auf ausführbaren Code für den Lambda-Ausdruck verweist, wird bei der Konvertierung in einen Ausdrucksbaumstrukturtyp eine Darstellung der Ausdrucksbaumstruktur des Lambda-Ausdrucks erstellt.

Es handelt sich bei Ausdrucksbaumstrukturen um effektive speicherinterne Datendarstellungen von Lambda-Ausdrücken, die die Struktur des Lambda-Ausdrucks übersichtlich und direkt widerspiegeln.

Ebenso wie der Delegattyp D verfügt Expression<D> über Parameter und Rückgabetypen, die mit denen von D identisch sind.

Im folgenden Beispiel wird ein Lambda-Ausdruck sowohl als ausführbarer Code als auch als Ausdrucksbaumstruktur dargestellt. Da eine Konvertierung in Func<int,int> vorhanden ist, ist auch eine Konvertierung in Expression<Func<int,int>> möglich:

Func<int,int> del = x => x + 1; // Code

Expression<Func<int,int>> exp = x => x + 1; // Data

Anschließend an diese Zuweisungen verweist der Delegat del auf eine Methode, die x + 1 zurückgibt, und die Ausdrucksbaumstruktur exp auf eine Datenstruktur, die den Ausdruck x => x + 1 beschreibt.

Die genaue Definition des generischen Typs Expression<D> sowie die Regeln für das Konstruieren einer Ausdrucksbaumstruktur für den Fall, dass ein Lambda-Ausdruck in einen Ausdrucksbaumstrukturtyp konvertiert wird, gehen über den Umfang dieser Spezifikation hinaus.

Folgende zwei Hinweise sind zu berücksichtigen:

* Nicht alle Lambda-Ausdrücke können in Ausdrucksbaumstrukturen konvertiert werden. Lambda-Ausdrücke mit Anweisungstexten und Lambda-Ausdrücke mit Zuweisungsausdrücken können z. B. nicht dargestellt werden. In diesen Fällen ist zwar eine Konvertierung vorhanden, die allerdings zur Kompilierzeit zu einem Fehler führt. Diese Ausnahmen werden in §6.5 genauer beschrieben.
* Expression<D> stellt die Instanzmethode Compile bereit, die einen Delegaten vom Typ D erstellt:

Func<int,int> del2 = exp.Compile();

Der durch die Ausdrucksbaumstruktur dargestellte Code kann durch einen Aufruf dieses Delegaten ausgeführt werden. Durch die oben stehenden Definitionen ergibt sich daraus, dass del und del2 identisch sind, sodass die folgenden beiden Anweisungen dieselben Auswirkungen haben:

int i1 = del(1);

int i2 = del2(1);

Nach dem Ausführen dieses Codes haben i1 und i2 jeweils den Wert 2.

## Der dynamische Typ

Der Typ dynamic hat eine besondere Bedeutung in C#. Sein Zweck besteht darin, dynamische Bindungen zu ermöglichen, was detailliert unter §7.2.2 beschrieben ist.

dynamic wird als mit object identisch angesehen, außer in folgenden Fällen:

* Operationen für Ausdrücke vom Typ dynamic können dynamisch gebunden werden (§7.2.2).
* Der Typrückschluss (§7.5.2) zieht dynamic gegenüber object vor, wenn beides Kandidaten sind.

Aufgrund dieser Äquivalenz trifft Folgendes zu:

* Es gibt eine implizite Identitätskonvertierung zwischen object und dynamic sowie zwischen konstruierten Typen, die identisch sind, beim Ersetzen von dynamic durch object.
* Implizite und explizite Konvertierung von und in object gelten auch für dynamic.
* Methodensignaturen, die beim Ersetzen von dynamic durch object identisch sind, werden als ein und dieselbe Signatur angesehen.

Der Typ dynamic kann zur Laufzeit nicht von object unterschieden werden.

Ein Ausdruck vom Typ dynamic wird als dynamischer Ausdruck bezeichnet.

# Variablen

Durch Variablen werden Speicherpositionen dargestellt. Jede Variable besitzt einen Typ, der festlegt, welche Werte in der Variablen gespeichert werden können. C# ist eine typsichere Sprache. Der C#-Compiler gewährleistet, dass in Variablen gespeicherte Werte immer vom geeigneten Typ sind. Der Wert einer Variablen kann über eine Zuweisung oder die Verwendung der Operatoren ++ und ‑‑ geändert werden.

Eine Variable muss definitiv zugewiesen werden (§5.3), bevor ihr Wert abgerufen werden kann.

Wie in den folgenden Abschnitten beschrieben, sind Variablen entweder anfänglich zugewiesen oder anfänglich nicht zugewiesen. Eine anfänglich zugewiesene Variable besitzt einen eindeutig definierten Anfangswert und wird immer als definitiv zugewiesen betrachtet. Eine anfänglich nicht zugewiesene Variable besitzt keinen Anfangswert. Damit eine anfänglich nicht zugewiesene Variable an einer bestimmten Position als definitiv zugewiesen betrachtet wird, muss für diese Variable in jedem möglichen zu dieser Position führenden Ausführungspfad eine Zuweisung vorgenommen werden.

## Variablenkategorien

C# definiert sieben Kategorien von Variablen: statische Variablen, Instanzvariablen, Arrayelemente, Werteparameter, Verweisparameter, Ausgabeparameter und lokale Variablen. Im folgenden Abschnitt wird jede dieser Kategorien näher beschrieben.

In dem Beispiel

class A  
{  
 public static int x;  
 int y;

void F(int[] v, int a, ref int b, out int c) {  
 int i = 1;  
 c = a + b++;  
 }  
}

ist x eine statische Variable, y eine Instanzvariable, v[0] ein Arrayelement, a ein Werteparameter, b ein Verweisparameter, c ein Ausgabeparameter und i eine lokale Variable.

### Statische Variablen

Ein mit dem static-Modifizierer deklariertes Feld wird als statische Variable bezeichnet. Eine statische Variable tritt vor Ausführung des statischen Konstruktors (§10.12) für den darin enthaltenen Typ in Erscheinung. Sie hört mit dem Wegfall der zugehörigen Anwendungsdomäne auf zu bestehen.

Der Anfangswert einer statischen Variablen ist der Standardwert (§5.2) des Variablentyps.

Für die definitive Zuweisungsüberprüfung wird eine statische Variable als anfänglich zugewiesen betrachtet.

### Instanzvariablen

Ein ohne static-Modifizierer deklariertes Feld wird als Instanzvariable bezeichnet.

#### Instanzvariablen in Klassen

Eine Instanzvariable einer Klasse tritt in Erscheinung, wenn eine neue Instanz dieser Klasse erstellt wird, und hört auf zu bestehen, wenn keine Verweise auf diese Instanz mehr vorhanden sind und der Destruktor der Instanz (so vorhanden) ausgeführt worden ist.

Der Anfangswert einer Instanzvariablen einer Klasse ist der Standardwert (§5.2) des Variablentyps.

Für die definitive Zuweisungsüberprüfung wird eine Instanzvariable einer Klasse als anfänglich zugewiesen betrachtet.

#### Instanzvariablen in Strukturen

Eine Instanzvariable einer Struktur besitzt genau dieselbe Lebensdauer wie die Strukturvariable, zu der sie gehört. Wenn also eine Variable eines Strukturtyps in Erscheinung tritt bzw. aufhört zu bestehen, gilt gleiches auch für die Instanzvariablen der Struktur.

Der anfängliche Zuweisungsstatus der Instanzvariablen einer Struktur stimmt mit dem der zugehörigen Strukturvariablen überein. Wenn eine Strukturvariable als anfänglich zugewiesen betrachtet wird, gilt folglich gleiches auch für ihre Instanzvariablen. Wird die Strukturvariable als anfänglich nicht zugewiesen betrachtet, sind auch die Instanzvariablen nicht zugewiesen.

### Arrayelemente

Die Elemente eines Arrays treten bei Erstellung einer Arrayinstanz in Erscheinung und hören auf zu bestehen, wenn keine Verweise auf diese Arrayinstanz mehr vorhanden sind.

Der Anfangswert für jedes einzelne Element eines Arrays ist der Standardwert (§5.2) des Arrayelementtyps.

Für die definitive Zuweisungsüberprüfung wird ein Arrayelement als anfänglich zugewiesen betrachtet.

### Werteparameter

Ein ohne die Modifizierer ref oder out deklarierter Parameter wird als Wertparameter bezeichnet.

Ein Wertparameter tritt in Erscheinung, wenn der zum Parameter gehörige Funktionsmember (Methode, Instanzkonstruktor, Accessor oder Operator) oder die anonyme Funktion aufgerufen wird. Er wird mit dem im Aufruf angegebenen Argumentwert initialisiert. Der Wertparameter hört i. d. R. mit Rückgabe des Funktionsmembers oder der anonymen Funktion auf zu bestehen. Wenn allerdings der Werteparameter von einer anonymen Funktion (§7.15) erfasst wird, verlängert sich seine Lebensdauer mindestens bis zu dem Zeitpunkt, zu dem der Delegat oder die Ausdrucksbaumstruktur, der bzw. die von dieser anonymen Funktion erstellt wurde, für die Garbage Collection freigegeben wird.

Für die definitive Zuweisungsüberprüfung wird ein Wertparameter als anfänglich zugewiesen betrachtet.

### Verweisparameter

Ein Parameter, der mit einem ref-Modifizierer deklariert wird, ist ein Verweisparameter.

Ein Verweisparameter erzeugt keine neue Speicherposition, sondern stellt dieselbe Speicherposition wie die Variable dar, die im Aufruf des Funktionsmembers oder der anonymen Funktion als Argument angegeben wird. Folglich entspricht der Wert eines Verweisparameters immer dem Wert der zugrunde liegenden Variablen.

Die folgenden definitiven Zuweisungsregeln gelten für Verweisparameter. Beachten Sie die Unterschiede gegenüber den in §5.1.6 beschriebenen Regeln für Ausgabeparameter.

* Eine Variable muss definitiv zugewiesen werden (§5.3), bevor sie als Verweisparameter im Aufruf eines Funktionsmembers oder eines Delegaten weitergegeben werden kann.
* Ein Verweisparameter innerhalb eines Funktionsmembers oder einer anonymen Funktion wird als anfänglich zugewiesen betrachtet.

Innerhalb einer Instanzmethode oder eines Instanzaccessors eines Strukturtyps verhält sich das this-Schlüsselwort genauso wie ein Verweisparameter des Strukturtyps (§7.6.7).

### Ausgabeparameter

Ein Parameter, der mit einem out-Modifizierer deklariert wird, ist ein Ausgabeparameter.

Ein Ausgabeparameter erzeugt keine neue Speicherposition, sondern stellt dieselbe Speicherposition wie die Variable dar, die im Aufruf des Funktionsmembers oder Delegaten als Argument angegeben wird. Folglich entspricht der Wert eines Ausgabeparameters immer dem der zugrunde liegenden Variablen.

Die folgenden definitiven Zuweisungsregeln gelten für Ausgabeparameter. Beachten Sie die Unterschiede gegenüber den in §5.1.5 beschriebenen Regeln für Verweisparameter.

* Eine Variable muss nicht definitiv zugewiesen werden, bevor sie als Ausgabeparameter im Aufruf eines Funktionsmembers oder Delegaten weitergegeben werden kann.
* Jede Variable, die nach einem normalen Aufruf eines Funktionsmembers oder Delegaten als Ausgabeparameter weitergegeben wurde, wird als zu diesem Ausführungspfad zugewiesen betrachtet.
* Ein Ausgabeparameter innerhalb eines Funktionsmembers oder einer anonymen Funktion wird als anfänglich nicht zugewiesen betrachtet.
* Jeder Ausgabeparameter eines Funktionsmembers oder einer anonymen Funktion muss vor der normalen Rückgabe des Funktionsmembers oder der anonymen Funktion definitiv zugewiesen werden (§5.3).

Innerhalb eines Instanzkonstruktors eines Strukturtyps verhält sich das this-Schlüsselwort genauso wie ein Ausgabeparameter des Strukturtyps (§7.6.7).

### Lokale Variablen

Eine lokale Variable wird durch eine local-variable-declaration, die in einem block, einem for-statement, einem switch-statement oder einem using-statement erfolgen kann, oder durch ein foreach-statement oder eine specific-catch-clause für ein try-statement deklariert.

Die Lebensdauer einer lokalen Variablen entspricht dem Teil der Programmausführung, während dem in jedem Fall Speicher für die Variable reserviert ist. Diese Lebensdauer erstreckt sich mindestens vom Eintritt in den block, das for-statement, das switch-statement, das using-statement, das foreach-statement oder die specific-catch-clause, dem bzw. der es zugeordnet ist, bis zum wie auch immer gearteten Beenden der Ausführung dieses block, for-statement, switch-statement, using-statement, foreach-statement oder dieser specific-catch-clause. (Der Eintritt in einen inneren block oder das Aufrufen einer Methode unterbricht die Ausführung des block, for-statement, switch-statement, using-statement, foreach-statement oder der specific-catch-clause, ohne diese(s) jedoch zu beenden.) Wenn die lokale Variable jedoch durch eine anonyme Funktion (§7.15.5.1) erfasst wird, verlängert sich ihre Lebensdauer mindestens bis zum Freigeben des Delegaten oder der Ausdrucksbaumstruktur, der bzw. die durch die anonyme Funktion erstellt wurde, sowie sämtlicher weiterer Objekte, die auf die erfasste Variable verweisen, für die Garbage Collection.

Wenn der übergeordnete block, das übergeordnete for-statement, switch-statement, using-statement, foreach-statement bzw. die übergeordnete specific-catch-clause rekursiv ausgeführt wird, wird jedes Mal eine neue Instanz der lokalen Variablen erstellt und deren local-variable-initializer, sofern vorhanden, jedes Mal ausgewertet.

Eine durch eine local-variable-declaration eingeführte lokale Variable wird nicht automatisch initialisiert und verfügt daher über keinen Standardwert. Für die definitive Zuweisungsüberprüfung wird eine lokale Variable, die durch eine local-variable-declaration eingeführt wird, als anfänglich nicht zugewiesen betrachtet. Eine local-variable-declaration kann einen local-variable-initializer enthalten. In diesem Fall wird die Variable nur nach dem Initialisierungsausdruck als definitiv zugewiesen betrachtet (§5.3.3.4).

Innerhalb des Gültigkeitsbereichs einer lokalen Variablen, die durch eine local-variable-declaration eingeführt wird, verursacht ein Verweis auf diese lokale Variable an einer Textposition, die vor dem local-variable-declarator liegt, einen Kompilierungsfehler. Wenn die Deklaration der lokalen Variablen implizit ist (§8.5.1), führt das Verweisen auf die Variable in ihrem local-variable-declarator ebenfalls zu einem Fehler.

Eine durch ein foreach-statement oder eine specific-catch-clause eingeführte lokale Variable wird innerhalb ihres gesamten Gültigkeitsbereichs als definitiv zugewiesen betrachtet.

Die tatsächliche Lebensdauer einer lokalen Variablen hängt von der Implementierung ab. Ein Compiler kann beispielsweise statisch festlegen, dass eine lokale Variable in einem Block nur für einen kleinen Teil dieses Bockes verwendet wird. Ausgehend von dieser Analyse könnte der Compiler Code generieren, der dazu führt, dass der Speicher der Variablen eine kürzere Lebensdauer hat als der enthaltene Block.

Der Speicher, auf den eine lokale Verweisvariable verweist, kann unabhängig von der Lebensdauer der lokalen Verweisvariablen (§3.9) freigegeben werden.

## Standardwerte

Die folgenden Variablenkategorien werden automatisch mit ihren Standardwerten initialisiert:

* Statische Variablen.
* Instanzvariablen von Klasseninstanzen.
* Arrayelemente.

Der Standardwert einer Variablen ist vom Typ der Variablen abhängig und wird folgendermaßen festgelegt:

* Bei einer Variablen vom value-type entspricht der Standardwert dem vom Standardkonstruktor des value-type berechneten Wert (§4.1.2).
* Bei einer Variablen vom reference-type ist der Standardwert null.

Die Initialisierung mit Standardwerten wird in der Regel dadurch erreicht, dass der Speicher-Manager oder der Garbage Collector den Speicher in allen Bits mit 0 initialisieren, bevor er für die Verwendung reserviert wird. Aus diesem Grund ist es sinnvoll, die 0 in allen Bits als NULL-Verweis zu verwenden.

## Definitive Zuweisung

Eine Variable wird als definitiv zugewiesen bezeichnet, wenn an einer angegebenen Position im ausführbaren Code eines Funktionsmembers der Compiler anhand einer bestimmten statischen Flussanalyse (§5.3.3) nachweisen kann, dass die Variable automatisch initialisiert wurde oder das Ziel mindestens einer Zuweisung war. Grob gesagt, bestehen für definitive Zuweisungen folgende Regeln:

* Eine anfänglich zugewiesene Variable (§5.3.1) wird immer als definitiv zugewiesen betrachtet.
* Eine anfänglich nicht zugewiesene Variable (§5.3.2) wird an einer angegebenen Position als definitiv zugewiesen betrachtet, wenn alle möglichen zu dieser Position führenden Ausführungspfade mindestens eines der folgenden Elemente enthalten:
* Eine einfache Zuweisung (§7.17.1), in der die Variable der linke Operand ist.
* Ein Aufrufausdruck (§7.6.5) oder Objekterstellungsausdruck (§7.6.10.1), der die Variable als Ausgabeparameter weitergibt.
* Für eine lokale Variable eine lokale Variablendeklaration (§8.5.1), die eine Variableninitialisierung einschließt.

Die diesen Regeln zugrunde liegende formale Spezifikation ist in §5.3.1, §5.3.2 und §5.3.3 beschrieben.

Die definitiven Zuweisungszustände von Instanzvariablen einer struct-type-Variablen werden sowohl einzeln als auch zusammen überwacht. Zusätzlich zu den oben dargestellten Regeln gilt für struct-type-Variablen und ihre Instanzvariablen Folgendes:

* Eine Instanzvariable gilt als definitiv zugewiesen, wenn die sie enthaltende struct-type-Variable als definitiv zugewiesen gilt.
* Eine struct-type-Variable gilt als definitiv zugewiesen, wenn jede ihrer Instanzvariablen als definitiv zugewiesen gilt.

Die definitive Zuweisung gilt in den folgenden Kontexten als Voraussetzung:

* Eine Variable muss an jeder Position definitiv zugewiesen sein, an der ein Wert abgerufen wird. Damit wird sichergestellt, dass keine nicht definierten Werte auftreten. Das Vorkommen einer Variablen in einem Ausdruck wird mit dem Abrufen des Wertes der Variablen gleichgesetzt. Das gilt nicht, wenn
* die Variable der linke Operand einer einfachen Zuweisung ist,
* die Variable als ein Ausgabeparameter weitergegeben wird oder
* die Variable eine struct-type-Variable ist und als linker Operand eines Memberzugriffs auftritt.
* Eine Variable muss an jeder Position definitiv zugewiesen werden, an der sie als Verweisparameter weitergegeben wird. Dadurch wird sichergestellt, dass der aufgerufene Funktionsmember den Verweisparameter als anfänglich zugewiesen betrachten kann.
* Alle Ausgabeparameter eines Funktionsmembers müssen an jeder Position definitiv zugewiesen werden, an die der Funktionsmember zurückkehrt (entweder über eine return-Anweisung oder wenn bei der Ausführung das Ende des Funktionsmembertexts erreicht wird). Dadurch wird sichergestellt, dass Funktionsmember in Ausgabeparametern keine undefinierten Werte zurückgeben. Auf dieser Grundlage kann der Compiler davon ausgehen, dass ein Funktionsmemberaufruf, der eine Variable als Ausgabeparameter verwendet, einer Zuweisung zur Variablen entspricht.
* Die this-Variable des Instanzkonstruktors eines struct-type muss an jeder Position, an die dieser Instanzkonstruktor zurückkehrt, definitiv zugewiesen sein.

### Anfänglich zugewiesene Variablen

Die folgenden Variablenkategorien sind als anfänglich zugewiesen klassifiziert:

* Statische Variablen.
* Instanzvariablen von Klasseninstanzen.
* Instanzvariablen von anfänglich zugewiesenen Strukturvariablen.
* Arrayelemente.
* Werteparameter.
* Verweisparameter
* In einer catch-Klausel oder einer foreach-Anweisung deklarierte Variablen.

### Anfänglich nicht zugewiesene Variablen

Die folgenden Variablenkategorien sind als anfänglich nicht zugewiesen klassifiziert:

* Instanzvariablen von anfänglich nicht zugewiesenen Strukturvariablen.
* Ausgabeparameter einschließlich der this-Variablen von Strukturinstanzkonstruktoren.
* Lokale Variablen, mit Ausnahme derjenigen, die in einer catch-Klausel oder einer foreach-Anweisung deklariert wurden.

### Exakte Regeln zur Ermittlung von definitiven Zuweisungen

Um festzustellen, ob die einzelnen verwendeten Variablen definitiv zugewiesen wurden, durchläuft der Compiler einen Prozess, der mit dem in diesem Abschnitt beschriebenen Prozess vergleichbar ist.

Der Compiler verarbeitet den Text der einzelnen Funktionsmember, die eine oder mehrere anfänglich nicht zugewiesene Variablen aufweisen. Für jede anfänglich nicht zugewiesene Variable v ermittelt der Compiler an jedem der folgenden Punkte im Funktionsmember einen definitiven Zuweisungszustand für v:

* Am Anfang jeder Anweisung
* Am Endpunkt (§8.1) jeder Anweisung
* Auf jedem Bogen, der die Steuerung an eine andere Anweisung oder den Endpunkt einer Anweisung übergibt
* Am Anfang jedes Ausdrucks
* Am Ende jedes Ausdrucks

Der definitive Zuweisungszustand von v kann einen der folgenden Zustände annehmen:

* Definitiv zugewiesen. Dieser Zustand zeigt an, dass in allen möglichen Ablaufsteuerungen zu diesem Punkt v ein Wert zugewiesen wurde.
* Nicht definitiv zugewiesen. Der Zustand einer Variablen, die nicht definitiv zugewiesen ist und am Ende eines Ausdrucks vom Typ bool vorkommt, kann (muss jedoch nicht notwendigerweise) einen der folgenden Unterzustandswerte annehmen:
* Nach true-Ausdruck definitiv zugewiesen. Dieser Zustand gibt an, dass v definitiv zugewiesen ist, wenn der boolesche Ausdruck als true ausgewertet wird, aber nicht zwingend zugewiesen sein muss, wenn der boolesche Ausdruck als false ausgewertet wird.
* Nach false-Ausdruck definitiv zugewiesen. Dieser Zustand gibt an, dass v definitiv zugewiesen ist, wenn der boolesche Ausdruck als false ausgewertet wird, aber nicht zwingend zugewiesen sein muss, wenn der boolesche Ausdruck als true ausgewertet wird.

Die folgenden Regeln legen fest, wie der Zustand einer Variablen v an den einzelnen Positionen ermittelt wird.

#### Allgemeine Regeln für Anweisungen

* v ist am Anfang eines Funktionsmembertextes nicht definitiv zugewiesen.
* v ist am Anfang einer nicht erreichbaren Anweisung definitiv zugewiesen.
* Der Zustand der definitiven Zuweisung von v am Anfang jeder anderen Anweisung wird ermittelt, indem der definitive Zuweisungszustand von v an allen Übergabepunkten der Ablaufsteuerung geprüft wird, die den Anfang dieser Anweisung als Ziel haben. Wenn v an allen diesen Übergabepunkten der Ablaufsteuerung definitiv zugewiesen ist, dann (und nur dann) ist v auch am Anfang der Anweisung definitiv zugewiesen. Die Anzahl der möglichen Übergabepunkte der Ablaufsteuerung wird auf dieselbe Weise ermittelt, wie die Erreichbarkeit von Anweisungen überprüft wird (§8.1).
* Der Zustand der definitiven Zuweisung von v am Endpunkt einer Blockanweisung bzw. einer Anweisung vom Typ checked, unchecked, if, while, do, for, foreach, lock, using oder switch wird ermittelt, indem der definitive Zuweisungszustand von v an allen Übergabepunkten der Ablaufsteuerung geprüft wird, die den Endpunkt dieser Anweisung als Ziel haben. Wenn v an allen diesen Übergabepunkten der Ablaufsteuerung definitiv zugewiesen ist, dann ist v auch am Endpunkt der Anweisung definitiv zugewiesen. Andernfalls ist v am Endpunkt der Anweisung nicht definitiv zugewiesen. Die Anzahl der möglichen Übergabepunkte der Ablaufsteuerung wird auf dieselbe Weise ermittelt, wie die Erreichbarkeit von Anweisungen überprüft wird (§8.1).

#### Blockanweisungen, die checked-Anweisung und die unchecked-Anweisung

Der definitive Zuweisungszustand von v bei der Übergabe der Steuerung an die erste Anweisung aus der Anweisungsliste im Block (bzw. an den Endpunkt des Blocks, falls die Anweisungsliste leer ist) entspricht der definitiven Zuweisungsanweisung von v vor der Blockanweisung bzw. vor der checked-Anweisung oder der unchecked-Anweisung.

#### Ausdrucksanweisungen

Für eine Ausdrucksanweisung stmt mit dem Ausdruck expr gilt Folgendes:

* v weist am Anfang von expr denselben definitiven Zuweisungszustand auf wie am Anfang von stmt.
* Wenn v am Ende von expr definitiv zugewiesen ist, ist die Variable am Endpunkt von stmt definitiv zugewiesen. Andernfalls ist sie am Endpunkt von stmt nicht definitiv zugewiesen.

#### Deklarationsanweisungen

* Wenn stmt eine Deklarationsanweisung ohne Initialisierer ist, dann weist v am Endpunkt von stmt denselben definitiven Zuweisungszustand wie am Anfang von stmt auf.
* Falls stmt eine Deklarationsanweisung mit Initialisierern ist, dann ist der definitive Zuweisungszustand für v so festgelegt, als ob es sich bei stmt um eine Anweisungsliste handelt, die pro Deklaration eine Zuweisungsanweisung mit einem Initialisierer (in der Reihenfolge der Deklaration) enthält.

#### If-Anweisungen

Für eine if-Anweisung stmt der folgenden Form gilt:

if ( expr ) then-stmt else else-stmt

* v weist am Anfang von expr denselben definitiven Zuweisungszustand auf wie am Anfang von stmt.
* Wenn v am Ende von expr definitiv zugewiesen ist, ist die Variable bei der Ablaufsteuerungsübergabe an then-stmt und an else-stmt bzw. an den Endpunkt von stmt definitiv zugewiesen, sofern keine else-Klausel vorhanden ist.
* Wenn v am Ende von expr den Zustand "nach true-Ausdruck definitiv zugewiesen" hat, ist die Variable bei der Ablaufsteuerungsübergabe an then-stmt definitiv und bei der Ablaufsteuerungsübergabe an else-stmt bzw. an den Endpunkt von stmt nicht definitiv zugewiesen, sofern keine else-Klausel vorhanden ist.
* Wenn v am Ende von expr den Zustand "nach false-Ausdruck definitiv zugewiesen" hat, ist die Variable bei der Ablaufsteuerungsübergabe an else-stmt definitiv und bei der Ablaufsteuerungsübergabe an then-stmt nicht definitiv zugewiesen. Sie ist am Endpunkt von stmt ausdrücklich nur dann definitiv zugewiesen, wenn sie am Endpunkt von then-stmt definitiv zugewiesen ist.
* Andernfalls gilt v bei der Übergabe der Ablaufsteuerung an then-stmt oder else-stmt bzw. an den Endpunkt von stmt nicht als definitiv zugewiesen, wenn keine else-Klausel vorhanden ist.

#### Switch-Anweisungen

In einer switch-Anweisung stmt mit einem steuernden Ausdruck expr gilt Folgendes:

* Der definitive Zuweisungszustand von v weist am Anfang von expr denselben Zustand wie v am Anfang von stmt auf.
* Der definitive Zuweisungszustand von v an der Übergabe der Ablaufsteuerung an eine erreichbare switch block-Anweisungsliste ist identisch mit dem definitiven Zuweisungszustand von v am Ende von expr.

#### While-Anweisung

Für eine while-Anweisung stmt der folgenden Form gilt:

while ( expr ) while-body

* v weist am Anfang von expr denselben definitiven Zuweisungszustand auf wie am Anfang von stmt.
* Wenn v am Ende von expr definitiv zugewiesen ist, dann ist die Variable bei der Ablaufsteuerungsübergabe an while-body und an den Endpunkt von stmt definitiv zugewiesen.
* Wenn v am Ende von expr den Zustand "nach true-Ausdruck definitiv zugewiesen" hat, ist die Variable bei der Ablaufsteuerungsübergabe an while-body definitiv und bei der Ablaufsteuerungsübergabe an den Endpunkt von stmt nicht definitiv zugewiesen.
* Wenn v am Ende von expr den Zustand "nach false-Ausdruck definitiv zugewiesen" hat, ist die Variable bei der Ablaufsteuerungsübergabe an den Endpunkt von stmt definitiv und bei der Ablaufsteuerungsübergabe an while-body nicht definitiv zugewiesen.

#### Do-Anweisungen

Für eine do-Anweisung stmt der folgenden Form gilt:

do do-body while ( expr ) ;

* v hat an der Übergabe der Ablaufsteuerung vom Anfang von stmt an do-body denselben definitiven Zuweisungszustand wie am Anfang von stmt.
* v weist am Anfang von expr denselben definitiven Zuweisungszustand auf wie am Endpunkt von do-body.
* Wenn v am Ende von expr definitiv zugewiesen ist, dann ist es auch an der Übergabe der Ablaufsteuerung an den Endpunkt von stmt definitiv zugewiesen.
* Wenn v am Ende von expr den Zustand "nach false-Ausdruck definitiv zugewiesen" hat, dann ist die Variable bei der Ablaufsteuerungsübergabe an den Endpunkt von stmt definitiv zugewiesen.

#### For-Anweisungen

Die Überprüfung des definitiven Zuweisungszustands für eine for-Anweisung der Form:

for ( for-initializer ; for-condition ; for-iterator ) embedded-statement

erfolgt so, als ob die Anweisung wie folgt geschrieben worden wäre:

{  
 for-initializer ;  
 while ( for-condition ) {  
 embedded-statement ;  
 for-iterator ;  
 }  
}

Wenn die for-condition in der for-Anweisung ausgelassen wird, wird die definitive Zuweisung genauso ausgewertet, als wäre for-condition in der vorangehenden Erweiterung durch true ersetzt worden.

#### Break-, continue- und goto-Anweisungen

Der definitive Zuweisungszustand von v bei der Übergabe der Ablaufsteuerung, die durch eine Anweisung vom Typ break, continue oder goto ausgelöst wurde, ist identisch mit dem definitiven Zuweisungszustand von v am Anfang der Anweisung.

#### Throw-Anweisungen

Für eine Anweisung stmt der folgenden Form gilt:

throw expr ;

Der definitive Zuweisungszustand von v am Anfang von expr ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.

#### Return-Anweisungen

Für eine Anweisung stmt der folgenden Form gilt:

return expr ;

* Der definitive Zuweisungszustand von v am Anfang von expr ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.
* Wenn v einen Ausgabeparameter darstellt, muss dieser entweder:
* nach expr
* oder am Ende des finally-Blocks einer try-finally-Anweisung oder einer try-catch-finally-Anweisung, die die return-Anweisung einschließt, definitiv zugewiesen sein.

Für eine Anweisung stmt der folgenden Form gilt:

return ;

* Wenn v einen Ausgabeparameter darstellt, muss dieser entweder:
* vor stmt
* oder am Ende des finally-Blocks einer try-finally-Anweisung oder einer try-catch-finally-Anweisung, die die return-Anweisung einschließt, definitiv zugewiesen sein.

#### Try-catch-Anweisungen

Für eine Anweisung stmt der folgenden Form gilt:

try try-block  
catch(...) catch-block-1  
...  
catch(...) catch-block-n

* Der definitive Zuweisungszustand von v am Anfang von try-block ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.
* Der definitive Zuweisungszustand von v am Anfang von catch-block-i (für beliebiges i) ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.
* Der definitive Zuweisungszustand von v am Endpunkt von stmt wird nur dann definitiv zugewiesen, wenn v am Endpunkt von try-block-block und jedem catch-block-i (für alle i-Werte zwischen 1 und n) definitiv zugewiesen ist.

#### Try-finally-Anweisungen

Für eine try-Anweisung stmt der folgenden Form gilt:

try try-block finally finally-block

* Der definitive Zuweisungszustand von v am Anfang von try-block ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.
* Der definitive Zuweisungszustand von v am Anfang von finally-block ist mit dem definitiven Zuweisungszustand von v am Anfang von stmt identisch.
* Der definitive Zuweisungszustand von v am Endpunkt von stmt lautet "definitiv zugewiesen", wenn mindestens eine der folgenden Bedingungen ausdrücklich zutrifft:
* v ist am Endpunkt von try-block definitiv zugewiesen
* v ist am Endpunkt von finally-block definitiv zugewiesen.

Bei einer Ablaufsteuerungsübergabe (z. B. einer goto-Anweisung), die innerhalb von try-block beginnt und außerhalb von try-block endet, wird v bei dieser Ablaufsteuerungsübergabe ebenfalls als definitiv zugewiesen betrachtet, wenn v am Endpunkt von finally-block definitiv zugewiesen ist. (Dies gilt jedoch nicht ausschließlich. Wenn v bei dieser Ablaufsteuerungsübergabe aus einem anderen Grund definitiv zugewiesen ist, wird die Variable weiterhin als definitiv zugewiesen betrachtet.)

#### Try-catch-finally-Anweisungen

Die Analyse des definitiven Zuweisungszustands für eine try-catch-finally-Anweisung der Form:

try try-block   
catch(...) catch-block-1  
...  
catch(...) catch-block-n  
finally finally-block

wird genauso überprüft, als wäre die Anweisung eine try-finally-Anweisung, die eine try-catch-Anweisung einschließt:

try {  
 try try-block   
 catch(...) catch-block-1  
 ...  
 catch(...) catch-block-n  
}  
finally finally-block

Im folgenden Beispiel wird veranschaulicht, wie die unterschiedlichen Blöcke einer try-Anweisung (§8.10) die definitive Zuweisung beeinflussen.

class A  
{  
 static void F() {  
 int i, j;  
 try {  
 goto LABEL;  
 // neither i nor j definitely assigned  
 i = 1;  
 // i definitely assigned  
 }

catch {  
 // neither i nor j definitely assigned  
 i = 3;  
 // i definitely assigned  
 }

finally {  
 // neither i nor j definitely assigned  
 j = 5;  
 // j definitely assigned  
 }  
 // i and j definitely assigned  
 LABEL:;  
 // j definitely assigned  
  
 }  
}

#### Foreach-Anweisungen

Für eine foreach-Anweisung stmt der folgenden Form gilt:

foreach ( type identifier in expr ) embedded-statement

* Der definitive Zuweisungszustand von v weist am Anfang von expr denselben Zustand wie v am Anfang von stmt auf.
* Der definitive Zuweisungsstatus von v bei der Ablaufsteuerungsübergabe an embedded-statement bzw. an den Endpunkt von stmt entspricht dem Status von v am Ende von expr.

#### Using-Anweisungen

Für eine using-Anweisung stmt der folgenden Form gilt:

using ( resource-acquisition ) embedded-statement

* Der definitive Zuweisungszustand von v weist am Anfang von resource-acquisition denselben Zustand auf wie v am Anfang von stmt.
* Der definitive Zuweisungsstatus von v bei der Ablaufsteuerungsübergabe an embedded-statement entspricht dem Status von v am Ende von resource-acquisition.

#### Lock-Anweisungen

Für eine lock-Anweisung stmt der folgenden Form gilt:

lock ( expr ) embedded-statement

* Der definitive Zuweisungszustand von v weist am Anfang von expr denselben Zustand wie v am Anfang von stmt auf.
* Der definitive Zuweisungsstatus von v bei der Ablaufsteuerungsübergabe an embedded-statement entspricht dem Status von v am Ende von expr.

#### Yield-Anweisungen

Für eine yield return-Anweisung stmt der folgenden Form gilt:

yield return expr ;

* Der definitive Zuweisungszustand von v weist am Anfang von expr denselben Zustand wie v am Anfang von stmt auf.
* Der definitive Zuweisungszustand von v am Ende von stmt entspricht dem Zustand von v am Ende von expr.

Eine yield break-Anweisung hat keine Auswirkungen auf den definitiven Zuweisungszustand.

#### Allgemeine Regeln für einfache Ausdrücke

Die folgende Regel ist für folgende Ausdrücke gültig: Literale (§7.6.1), einfache Namen (§7.6.2), Ausdrücke zum Memberzugriff (§7.6.4), Ausdrücke zum nicht indizierten Basiszugriff (§7.6.8), typeof-Ausdrücke (§7.6.11) und Standardwertausdrücke (§7.6.13).

* Der definitive Zuweisungsstatus von v am Ende eines solchen Ausdrucks ist mit dem definitiven Zuweisungsstatus von v am Anfang des Ausdrucks identisch.

#### Allgemeine Regeln für Ausdrücke mit eingebetteten Ausdrücken

Die folgenden Regeln sind für folgende Ausdrücke gültig: Klammerausdrücke (§7.6.3), Ausdrücke zum Elementzugriff (§7.6.6), Ausdrücke zum indizierten Basiszugriff (§7.6.8), Inkrement- und Dekrementausdrücke (§7.6.9, §7.7.5), Umwandlungsausdrücke (§7.7.6), unäre Ausdrücke +, -, ~, \*, binäre Ausdrücke +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ (§7.8, §7.9, §7.10, §7.11), zusammengesetzte Zuweisungsausdrücke (§7.17.2), checked- und unchecked-Ausdrücke (§7.6.12) sowie Ausdrücke zur Array- und Delegaterstellung (§7.6.10).

Alle diese Ausdrücke haben einen oder mehrere Unterausdrücke, die in einer festen Reihenfolge ohne Bedingung ausgewertet werden. Durch den binären %-Operator wird beispielsweise erst die linke Seite des Operators und dann die rechte Seite ausgewertet. Ein Indizierungsvorgang wertet den indizierten Ausdruck und dann die einzelnen Indexausdrücke von links nach rechts aus. Für einen in dieser Reihenfolge ausgewerteten expr-Ausdruck mit den Unterausdrücken expr1, expr2, ..., exprn gilt:

* Der definitive Zuweisungszustand von v weist am Anfang von expr1 denselben definitiven Zuweisungszustand auf wie am Anfang von expr.
* Der definitive Zuweisungszustand von v weist am Anfang von expri (i größer als eins) denselben definitiven Zuweisungszustand auf wie am Ende von expri-1.
* Der definitive Zuweisungszustand von v weist am Ende von expr denselben definitiven Zuweisungszustand auf wie am Ende von exprn.

#### Aufrufausdrücke und Objekterstellungsausdrücke

Für einen Aufrufausdruck expr der folgenden Form:

primary-expression ( arg1 , arg2 , … , argn )

oder einen Objekterstellungsausdruck der folgenden Form gilt:

new type ( arg1 , arg2 , … , argn )

* Bei einem Aufrufausdruck entspricht der definitive Zuweisungszustand von v vor primary-expression dem Zustand von v vor expr.
* Bei einem Aufrufausdruck entspricht der definitive Zuweisungszustand von v vor arg1 dem Zustand von v nach primary-expression.
* Bei einem Objekterstellungsausdruck entspricht der definitive Zuweisungszustand von v vor arg1 dem Zustand von v vor expr.
* Für alle Argumente argi wird der definitive Zuweisungszustand von v nach argi durch die regulären Ausdrucksregeln bestimmt, wobei alle Modifizierer ref oder out unberücksichtigt bleiben.
* Für alle argi-Argumente (i größer als eins) ist der definitive Zuweisungszustand von v vor argi mit dem Zustand von v nach argi-1 identisch.
* Wenn die Variable v als ein out-Argument (d. h. als ein Argument der Form "out v") an ein beliebiges Argument übergeben wird, dann ist der Zustand von v nach dem Ausdruck expr definitiv zugewiesen. Andernfalls ist der Zustand von v nach dem Ausdruck expr mit dem Zustand von v nach argn identisch.
* Bei Arrayinitialisierern (§7.6.10.4), Objektinitialisierern (§7.6.10.2), Auflistungsinitialisierern (§7.6.10.3) und anonymen Objektinitialisierern (§7.6.10.6) wird der definitive Zuweisungszustand anhand der Erweiterung bestimmt, nach der diese Konstrukte definiert sind.

#### Einfache Zuweisungsausdrücke

Für einen Ausdruck expr der Form w = expr-rhs gilt Folgendes:

* Der definitive Zuweisungsstatus von v vor dem Ausdruck expr-rhs ist mit dem definitiven Zuweisungsstatus von v vor expr identisch.
* Wenn w mit der Variablen v übereinstimmt, dann ist der definitive Zuweisungsstatus von v nach expr definitiv zugewiesen. Andernfalls ist der definitive Zuweisungsstatus von v nach expr mit dem definitiven Zuweisungsstatus von v nach expr-rhs identisch.

#### &&-Ausdrücke

Für einen Ausdruck expr der Form expr-first && expr-second gilt Folgendes:

* Der definitive Zuweisungsstatus von v vor dem Ausdruck expr-first ist mit dem definitiven Zuweisungsstatus von v vor expr identisch.
* Der definitive Zuweisungszustand von v vor expr-second lautet "definitiv zugewiesen", wenn der Zustand von v nach expr-first "definitiv zugewiesen" oder "nach true-Ausdruck definitiv zugewiesen" lautet. Andernfalls ist die Variable nicht definitiv zugewiesen.
* Der definitive Zuweisungsstatus von v nach dem Ausdruck expr wird durch folgende Bedingungen bestimmt:
* Wenn expr-first ein konstanter Ausdruck mit dem Wert false ist, ist der definitive Zuweisungsstatus von v nach expr derselbe Zuweisungsstatus wie v nach expr-first.
* Wenn der Status von v nach expr-first jedoch definitiv zugewiesen ist, ist auch Status von v nach expr definitiv zugewiesen.
* Wenn der Zustand von v nach expr-second "definitiv zugewiesen" und der Zustand von v nach expr-first "nach false-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "definitiv zugewiesen".
* Wenn der Zustand von v nach expr-second "definitiv zugewiesen" oder "nach true-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "nach true-Ausdruck definitiv zugewiesen".
* Wenn der Zustand von v nach expr-first "nach false-Ausdruck definitiv zugewiesen" und der Zustand von v nach expr-second "nach false-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "nach false-Ausdruck definitiv zugewiesen".
* Andernfalls lautet der Status von v nach expr "nicht definitiv zugewiesen".

In dem Beispiel

class A  
{  
 static void F(int x, int y) {  
 int i;  
 if (x >= 0 && (i = y) >= 0) {  
 // i definitely assigned  
 }  
 else {  
 // i not definitely assigned  
 }  
 // i not definitely assigned  
 }  
}

wird die Variable i in genau einer der eingebetteten Anweisungen einer if-Anweisung als definitiv zugewiesen betrachtet. Für die jeweils anderen Anweisungen gilt dies nicht. In der if-Anweisung der Methode F ist die Variable i in der ersten eingebetteten Anweisung definitiv zugewiesen, da der Ausdruck (i = y) stets vor der eingebetteten Anweisung ausgeführt wird. Im Gegensatz dazu ist die Variable i in der zweiten eingebetteten Anweisung nicht definitiv zugewiesen, da der Test von x >= 0 den Wert false ergeben haben könnte, wodurch die Variable i nicht zugewiesen würde.

#### ||-Ausdrücke

Für einen Ausdruck expr der Form expr-first || expr-second gilt Folgendes:

* Der definitive Zuweisungsstatus von v vor dem Ausdruck expr-first ist mit dem definitiven Zuweisungsstatus von v vor expr identisch.
* Der definitive Zuweisungszustand von v vor expr-second ist definitiv zugewiesen, wenn der Zustand von v nach expr-first entweder definitiv zugewiesen ist oder "definitiv zugewiesen nach false-Ausdruck" lautet. Andernfalls ist die Variable nicht definitiv zugewiesen.
* Der definitive Zuweisungsstatus von v nach dem Ausdruck expr wird durch folgende Bedingungen bestimmt:
* Wenn expr-first ein konstanter Ausdruck mit dem Wert true ist, ist der definitive Zuweisungsstatus von v nach expr derselbe Zuweisungsstatus wie v nach expr-first.
* Wenn der Status von v nach expr-first jedoch definitiv zugewiesen ist, ist auch Status von v nach expr definitiv zugewiesen.
* Wenn der Zustand von v nach dem Ausdruck expr-second "definitiv zugewiesen" und der Zustand von v nach expr-first "nach true-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "definitiv zugewiesen".
* Andernfalls gilt, dass der Zustand von v nach dem Ausdruck expr-second "definitiv zugewiesen nach false-Ausdruck" lautet, wenn der Zustand von v nach expr definitiv zugewiesen ist oder "definitiv zugewiesen nach false-Ausdruck" lautet.
* Wenn der Zustand von v nach expr-first "nach true-Ausdruck definitiv zugewiesen" und der Zustand von v nach expr-second "nach true-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "nach true-Ausdruck definitiv zugewiesen".
* Andernfalls lautet der Status von v nach expr "nicht definitiv zugewiesen".

In dem Beispiel

class A  
{  
 static void G(int x, int y) {  
 int i;  
 if (x >= 0 || (i = y) >= 0) {  
 // i not definitely assigned  
 }  
 else {  
 // i definitely assigned  
 }  
 // i not definitely assigned  
 }  
}

wird die Variable i in genau einer der eingebetteten Anweisungen einer if-Anweisung als definitiv zugewiesen betrachtet. Für die jeweils anderen Anweisungen gilt dies nicht. In der if-Anweisung der Methode G ist die Variable i in der zweiten eingebetteten Anweisung definitiv zugewiesen, da der Ausdruck (i = y) stets vor der eingebetteten Anweisung ausgeführt wird. Im Gegensatz dazu ist die Variable i in der ersten eingebetteten Anweisung nicht definitiv zugewiesen, da der Test von x >= 0 den Wert true ergeben haben könnte, wodurch die Variable i nicht zugewiesen würde.

#### ! Ausdrücke

Für einen Ausdruck expr der Form ! expr-operand gilt Folgendes:

* Der definitive Zuweisungszustand von v vor dem Ausdruck expr-operand stimmt mit dem definitiven Zuweisungszustand von v vor expr überein.
* Der definitive Zuweisungsstatus von v nach dem Ausdruck expr wird durch folgende Bedingungen bestimmt:
* Wenn der Zustand von v nach expr-operand "definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr"definitiv zugewiesen".
* Wenn der Zustand von v nach dem Ausdruck expr-operand nicht definitiv zugewiesen ist, dann ist auch der Zustand von v nach dem Ausdruck expr nicht definitiv zugewiesen.
* Wenn der Zustand von v nach expr-operand "nach false-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr"nach true-Ausdruck definitiv zugewiesen".
* Wenn der Zustand von v nach expr-operand "nach true-Ausdruck definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr"nach false-Ausdruck definitiv zugewiesen".

#### ??-Ausdrücke

Für einen Ausdruck expr der Form expr-first ?? expr-second gilt Folgendes:

* Der definitive Zuweisungsstatus von v vor dem Ausdruck expr-first ist mit dem definitiven Zuweisungsstatus von v vor expr identisch.
* Der definitive Zuweisungszustand von v vor expr-second entspricht dem definitiven Zuweisungszustand von v nach expr-first.
* Der definitive Zuweisungsstatus von v nach dem Ausdruck expr wird durch folgende Bedingungen bestimmt:
* Wenn expr-first ein konstanter Ausdruck (§7.19) mit dem Wert NULL ist, dann entspricht der Zustand von v nach expr dem Zustand von v nach expr-second.
* Andernfalls ist der Zustand von v nach expr mit dem definitiven Zuweisungszustand von v nach expr-first identisch.

#### ?:-Ausdrücke

Für einen Ausdruck expr der Form expr-cond ? expr-true : expr-false gilt:

* Der definitive Zuweisungszustand von v vor dem Ausdruck expr-cond stimmt mit dem Zustand von v vor expr überein.
* Der definitive Zuweisungsstatus von v vor expr-true lautet "definitiv zugewiesen", wenn mindestens eine der folgenden Bedingungen ausdrücklich zutrifft:
* expr-cond ist ein konstanter Ausdruck mit dem Wert false
* Der Zustand von v nach expr-cond lautet "definitiv zugewiesen" oder "nach true-Ausdruck definitiv zugewiesen".
* Der definitive Zuweisungsstatus von v vor expr-false lautet "definitiv zugewiesen", wenn mindestens eine der folgenden Bedingungen ausdrücklich zutrifft:
* expr-cond ist ein konstanter Ausdruck mit dem Wert true
* Der Zustand von v nach expr-cond lautet "definitiv zugewiesen" oder "nach false-Ausdruck definitiv zugewiesen".
* Der definitive Zuweisungsstatus von v nach dem Ausdruck expr wird durch folgende Bedingungen bestimmt:
* Wenn expr-cond ein konstanter Ausdruck (§7.19) mit dem Wert true ist, dann entspricht der Zustand von v nach expr dem Zustand von v nach expr-true.
* Wenn expr-cond ein konstanter Ausdruck (§7.19) mit dem Wert false ist, dann entspricht der Zustand von v nach expr dem Zustand von v nach expr-false.
* Wenn der Zustand von v nach expr-true und der Zustand von v nach expr-false "definitiv zugewiesen" lautet, dann lautet der Zustand von v nach expr "definitiv zugewiesen".
* Andernfalls lautet der Status von v nach expr "nicht definitiv zugewiesen".

#### Anonyme Funktionen

Für eine lambda-expression oder eine anonymous-method-expression expr mit dem Text (entweder block oder expression) body gilt Folgendes:

* Der definitive Zuweisungszustand einer äußeren Variablen v vor body ist mit dem Zustand von v vor expr identisch. Das bedeutet, dass der definitive Zuweisungszustand von äußeren Variablen an den Kontext der anonymen Funktion vererbt wird.
* Der definitive Zuweisungszustand einer äußeren Variablen v nach expr ist mit dem Zustand von v vor expr identisch.

In dem Beispiel

delegate bool Filter(int i);

void F() {  
 int max;

// Error, max is not definitely assigned  
 Filter f = (int n) => n < max;

max = 5;  
 DoWork(f);  
}

generiert einen Kompilierungsfehler, da max bei Deklaration der anonymen Funktionen nicht definitiv zugewiesen ist. In dem Beispiel

delegate void D();

void F() {  
 int n;  
 D d = () => { n = 1; };

d();

// Error, n is not definitely assigned  
 Console.WriteLine(n);  
}

generiert ebenfalls einen Kompilierungsfehler, da die Zuweisung zu n in der anonymen Funktion keine Auswirkungen auf den definitiven Zuweisungszustand von n außerhalb der anonymen Funktion hat.

## Variablenverweise

Eine variable-reference ist eine expression, die als Variable klassifiziert ist. Eine variable-reference bezeichnet eine Speicherposition, unter der der aktuelle Wert abgerufen und ein neuer Wert gespeichert werden kann.

variable-reference:  
expression

In C und C++ wird ein variable-reference als lvalue bezeichnet.

## Unteilbarkeit von Variablenverweisen

Lese- und Schreibzugriffe der folgenden Datentypen sind unteilbar: bool, char, byte, sbyte, short, ushort, uint, int, float sowie Verweistypen. Darüber hinaus sind Lese- und Schreibzugriffe von Enumerationstypen mit einem zugrunde liegenden Typ aus der vorangehenden Liste unteilbar. Lese- und Schreibzugriffe anderer Typen, einschließlich long, ulong, double und decimal, sowie benutzerdefinierter Typen müssen nicht unbedingt unteilbar sein. Abgesehen von den für diesen Zweck entwickelten Bibliotheksfunktionen kann wie bei der Inkrementierung oder Dekrementierung nicht davon ausgegangen werden, dass Lese-, Änderungs- und Schreibzugriffe unteilbar sind.

# Konvertierungen

Mithilfe einer Konvertierung kann ein Ausdruck wie ein bestimmter Typ behandelt werden. Mit einer Konvertierung kann ein Ausdruck eines bestimmten Typs wie ein Ausdruck eines anderen Typs behandelt oder einem Ausdruck ohne Typ ein Typ zugeordnet werden. Konvertierungen können implizit oder explizit erfolgen. Die verwendete Konvertierungsart bestimmt, ob eine explizite Typumwandlung erforderlich ist. Eine Konvertierung aus dem Typ int in den Typ long ist beispielsweise implizit, sodass Ausdrücke mit dem Typ int implizit als Typ long behandelt werden können. Einer Konvertierung in die andere Richtung (aus dem Typ long in den Typ int) gilt als explizit. Somit ist eine explizite Typumwandlung erforderlich.

int a = 123;  
long b = a; // implicit conversion from int to long  
int c = (int) b; // explicit conversion from long to int

Einige Konvertierungen werden durch die Sprache definiert. Programme können ebenfalls eigene Konvertierungen definieren (§6.4).

## Implizite Konvertierungen

Die folgenden Konvertierungen sind als implizite Konvertierungen klassifiziert:

* Identitätskonvertierungen
* Implizite numerische Konvertierungen
* Implizite Enumerationskonvertierungen
* Implizite Konvertierungen, die NULL-Werte zulassen
* NULL-Literalkonvertierungen
* Implizite Verweiskonvertierungen
* Boxingkonvertierungen
* Implizite dynamische Konvertierungen
* Implizite Konvertierungen konstanter Ausdrücke
* Benutzerdefinierte implizite Konvertierungen
* Konvertierungen anonymer Funktionen
* Konvertierungen von Methodengruppen

Implizite Konvertierungen können in einer Vielzahl von Situationen auftreten, z. B. in Funktionsmemberaufrufen (§7.5.4), Umwandlungsausdrücken (§7.7.6) und Zuweisungen (§7.17).

Die vordefinierten, impliziten Konvertierungen sind immer erfolgreich und lösen nie Ausnahmen aus. Korrekt entwickelte, benutzerdefinierte, implizite Konvertierungen sollten sich gleichermaßen verhalten.

Für die Zwecke von Konvertierungen werden die Typen object und dynamic als äquivalent angesehen.

Dynamische Konvertierungen (§6.1.8 und §6.2.6) gelten jedoch nur für Ausdrücke vom Typ dynamic (§4.7)

### Identitätskonvertierungen

Bei einer Identitätskonvertierung wird ein beliebiger Typ in denselben Typ konvertiert. Der Sinn dieser Konvertierung besteht darin, dass eine Entität, die bereits den erforderlichen Typ besitzt, als in diesen Typ konvertierbar bezeichnet werden kann.

Da object und dynamic als äquivalent betrachtet werden, besteht eine Identitätskonvertierung zwischen object und dynamic sowie zwischen konstruierten Typen, die identisch sind, wenn alle Vorkommnisse von dynamic durch object ersetzt werden.

### Implizite numerische Konvertierungen

Folgende Konvertierungen gelten als implizit numerisch:

* Von sbyte in short, int, long, float, double oder decimal
* Von byte in short, ushort, int, uint, long, ulong, float, double oder decimal
* Von short in int, long, float, double oder decimal
* Von ushort in int, uint, long, ulong, float, double oder decimal.
* Von int in long, float, double oder decimal.
* Von uint in long, ulong, float, double oder decimal
* Von long in float, double oder decimal
* Von ulong in float, double oder decimal
* Von char in ushort, int, uint, long, ulong, float, double oder decimal.
* Von float in double

Konvertierungen von int, uint, long oder ulong in float und von long oder ulong in double können zwar zu einem Verlust an Genauigkeit führen, haben aber unter keinen Umständen einen Größenverlust zur Folge. Alle anderen impliziten, numerischen Konvertierungen führen nie zu einem Verlust an Daten.

Es gibt keine impliziten Konvertierungen in den Typ char. Daher werden Werte anderer ganzzahliger Typen nicht automatisch in den Typ char konvertiert.

### Implizite Enumerationskonvertierungen

Mithilfe einer impliziten Enumerationskonvertierung kann das decimal-integer-literal 0 in einen beliebigen enum-type konvertiert werden sowie in einen beliebigen nullable-type, dessen zugrunde liegender Typ ein enum-type ist. Im zweiten Fall wird die Konvertierung ausgewertet, indem der zugrunde liegende enum-type konvertiert und das Ergebnis gewrappt wird (§4.1.10).

### Implizite Konvertierungen, die NULL-Werte zulassen

Vordefinierte implizite Konvertierungen für Werttypen, die keine NULL-Werte zulassen, können auch für die Formen dieser Typen verwendet werden, die auch NULL-Werte zulassen. Für alle vordefinierten impliziten Identitäts- und numerischen Konvertierungen von einem Werttyp S, der keine NULL-Werte zulässt, in einen Werttyp T, der keine NULL-Werte zulässt, sind die folgenden impliziten Konvertierungen, die NULL-Werte zulassen, verfügbar:

* Implizite Konvertierung von S? in T?.
* Implizite Konvertierung von S in T?.

Die Auswertung einer impliziten Konvertierung, die NULL-Werte zulässt und auf einer zugrunde liegenden Konvertierung von S in T basiert, läuft folgendermaßen ab:

* Wenn die Konvertierung, die NULL-Werte zulässt, von S? in T? erfolgt:
* Wenn der Ausgangswert NULL ist (die HasValue-Eigenschaft ist false), ist das Ergebnis der NULL-Wert vom Typ T?.
* Andernfalls wird die Konvertierung als Entwrappen von S? in S ausgewertet, anschließend wird die zugrunde liegende Konvertierung von S in Tdurchgeführt, gefolgt von einem Wrapping (§4.1.10) von T in T?.
* Wenn die Konvertierung, die NULL-Werte zulässt, von S in T? erfolgt, wird die Konvertierung als die zugrunde liegende Konvertierung von S in T gefolgt von einem Wrapping von T in T? ausgewertet.

### NULL-Literalkonvertierungen

Eine implizite Konvertierung des null-Literals in einen beliebigen Typ, der NULL-Werte zulässt, ist möglich. Bei dieser Konvertierung wird der NULL-Wert (§4.1.10) des angegebenen Typs, der NULL-Werte zulässt, erstellt.

### Implizite Verweiskonvertierungen

Folgende Konvertierungen gelten als implizite Verweiskonvertierungen:

* Von einem beliebigen reference-type in object und dynamic.
* Von einem beliebigen class-type S in einen beliebigen class-type T, vorausgesetzt, dass S von T abgeleitet ist.
* Von einem beliebigen class-type S in einen beliebigen interface-type T, vorausgesetzt S implementiert T.
* Von einem beliebigen interface-type S in einen beliebigen interface-type T, vorausgesetzt, dass S von T abgeleitet ist.
* Von einem array-type S mit einem Elementtyp SE in einen array-type T mit einem Elementtyp TE, vorausgesetzt, dass alle folgenden Anforderungen erfüllt sind:
* S und T unterscheiden sich nur hinsichtlich des Elementtyps. Das bedeutet, dass S und T dieselbe Anzahl von Dimensionen haben.
* Hier handelt es sich sowohl bei SE als auch bei TE um reference-types.
* Es ist eine implizite Verweiskonvertierung von SE in TE vorhanden.
* Von einem beliebigen array-type in System.Array sowie die implementierten Schnittstellen.
* Von einem eindimensionalen Arraytyp S[] in System.Collections.Generic.IList<T> und seine Basisschnittstellen, sofern eine implizite Identitäts- oder Verweiskonvertierung von S in T vorhanden ist.
* Von einem beliebigen delegate-type in System.Delegate sowie die implementierten Schnittstellen.
* Vom NULL-Literal in einen beliebigen reference-type.
* Von einem beliebigen reference-type in einen reference-type T, wenn er eine implizite Identitäts- oder Verweiskonvertierung in einen reference-type T0 aufweist und T0 eine Identitätskonvertierung in T aufweist.
* Von einem beliebigen reference-type in einen Schnittstellen- oder Delegattyp T, wenn er eine implizite Identitäts- oder Verweiskonvertierung in einen Schnittstellen- oder Delegattyp T0 aufweist und T0 varianzkonvertierbar (§13.1.3.2) in T ist.
* Implizite Konvertierungen mit Typparametern, von denen bekannt ist, dass sie Verweistypen sind. Weitere Informationen über implizite Konvertierungen mit Typparametern finden Sie in §6.1.10.

Die impliziten Verweiskonvertierungen von reference-types verlaufen stets nachweisbar erfolgreich und erfordern daher keine Überprüfung zur Laufzeit.

Verweiskonvertierungen (implizit oder explizit) ändern niemals die referenzielle Identität des konvertierten Objekts. Anders ausgedrückt kann eine Verweiskonvertierung zwar den Verweistyp, niemals aber den Typ oder Wert des referenzierten Objekts ändern.

### Boxingkonvertierungen

Eine Boxingkonvertierung ermöglicht die implizite Konvertierung eines value-type in einen Verweistyp. Eine Boxingkonvertierung ermöglicht die Konvertierung eines beliebigen non-nullable-value-type in object und dynamic bzw. System.ValueType sowie einen beliebigen interface-type, der vom non-nullable-value-type implementiert wird. Darüber hinaus kann ein enum-type in den Typ System.Enum konvertiert werden.

Eine Boxingkonvertierung ermöglicht genau dann eine Konvertierung eines nullable-type in einen Verweistyp, wenn eine Boxingkonvertierung aus dem zugrunde liegenden non-nullable-value-type in den Verweistyp vorhanden ist.

Ein Werttyp hat eine Boxingkonvertierung in einen Schnittstellentyp I, wenn er eine Boxingkonvertierung in Schnittstellentyp I0 aufweist und I0 eine Identitätskonvertierung in I aufweist.

Ein Werttyp weist eine Boxingkonvertierung in einen Schnittstellentyp I auf, wenn er eine Boxingkonvertierung in einen Schnittstellen- oder Delegattyp I0 aufweist und I0 varianzkonvertierbar (§13.1.3.2) in I ist.

Das Boxing des Werts eines non-nullable-value-type besteht aus der Reservierung einer Objektinstanz und dem Kopieren des value-type-Werts in diese Instanz. Das Boxing einer Struktur in den Typ System.ValueType ist möglich, da dies die Basisklasse aller Strukturen (§11.3.2) ist.

Das Boxing eines nullable-type läuft folgendermaßen ab:

* Wenn der Ausgangswert NULL ist (die HasValue-Eigenschaft ist false), ist das Ergebnis ein NULL-Verweis auf den Zieltyp.
* Andernfalls ist das Ergebnis ein Verweis auf einen mithilfe von Boxing konvertierten Wert T, der durch Entwrappen und Konvertieren des Ausgangswerts erstellt wurde.

Boxingkonvertierungen werden ausführlich in §4.3.1 beschrieben.

### Implizite dynamische Konvertierungen

Es ist eine implizite dynamische Konvertierung von einem Ausdruck vom Typ dynamic in einen beliebigen Typ T vorhanden. Die Konvertierung ist dynamisch gebunden (§7.2.2), d. h. dass zur Laufzeit eine implizite Konvertierung vom Laufzeittyp des Ausdrucks in T gesucht wird. Wenn keine Konvertierung gefunden wird, wird eine Laufzeitausnahme ausgelöst.

Beachten Sie, dass diese implizite Konvertierung scheinbar dem Ratschlag vom Anfang von §6.1 widerspricht, dass eine implizite Konvertierung niemals eine Ausnahme auslösen sollte. Es ist jedoch nicht die Konvertierung selbst, die die Ausnahme auslöst, sondern das Finden der Konvertierung. Bei der Verwendung von dynamischer Bindung ist das Risiko von Laufzeitausnahmen inhärent. Wenn eine dynamische Bindung der Konvertierung nicht erwünscht ist, kann der Ausdruck zuerst in object und anschließend in den gewünschten Typ konvertiert werden.

Das folgende Beispiel veranschaulicht implizite dynamische Konvertierungen:

object o = “object”  
dynamic d = “dynamic”;

string s1 = o; // Fails at compile-time – no conversion exists  
string s2 = d; // Compiles and succeeds at run-time  
int i = d; // Compiles but fails at run-time – no conversion exists

Die Zuweisungen an s2 und i verwenden beide implizite dynamische Konvertierungen, wobei die Bindung der Vorgänge bis zur Laufzeit unterbrochen wird. Zur Laufzeit werden implizite Konvertierungen vom Laufzeittyp d – string – in den Zieltyp gesucht. Es wird eine Konvertierung in string, aber nicht in int gefunden.

### Implizite Konvertierungen konstanter Ausdrücke

Die implizite Konvertierung eines konstanten Ausdrucks bietet folgende Möglichkeiten:

* Eine constant-expression (§7.19) des Typs int kann in den Typ sbyte, byte, short, ushort, uint oder ulong konvertiert werden, vorausgesetzt, der Wert der constant-expression liegt innerhalb des Bereichs des Zieltyps.
* Eine constant-expression vom Typ long kann in den Typ ulong konvertiert werden, vorausgesetzt, der Wert der constant-expression ist nicht negativ.

### Implizite Konvertierungen mit Typparametern

Für einen bestimmten Typparameter T sind die folgenden impliziten Konvertierungen vorhanden:

* T in dessen effektive Basisklasse C, T in eine beliebige Basisklasse von C und T in eine beliebige von C implementierte Schnittstelle. Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Boxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als implizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* T in einen Schnittstellentyp I aus der effektiven Schnittstellengruppe von T und T in eine beliebige Basisschnittstelle von I. Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Boxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als implizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* Von T in einen Typparameter U, vorausgesetzt, T ist von U abhängig (§10.1.5). Wenn U ein Werttyp ist, weisen T und U zur Laufzeit notwendigerweise denselben Typ auf, und es wird keine Konvertierung durchgeführt. Wenn T andernfalls ein Werttyp ist, wird die Konvertierung zur Laufzeit als Boxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als implizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* Das NULL-Literal in T, sofern bekannt ist, dass T ein Verweistyp ist.
* Von T in einen Verweistyp I, wenn er eine implizite Konvertierung in einen Verweistyp S0 aufweist, und S0 eine Identitätskonvertierung in S aufweist. Zur Laufzeit wird die Konvertierung auf dieselbe Art und Weise ausgeführt wie die Konvertierung in S0.
* Von T in einen Schnittstellentyp I, wenn er eine implizite Konvertierung in einen Schnittstellen- oder Delegattyp I0 aufweist, und I0 varianzkonvertierbar in I ist (§13.1.3.2). Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Boxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als implizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.

Wenn von T bekannt ist, dass es ein Verweistyp ist (§10.1.5), gelten alle genannten Konvertierungen als implizite Verweiskonvertierung (§6.1.6). Wenn T nicht als Verweistyp bekannt ist, gelten die genannten Konvertierungen als Boxingkonvertierungen (§6.1.7).

### Benutzerdefinierte implizite Konvertierungen

Eine benutzerdefinierte, implizite Konvertierung besteht aus einer optionalen, impliziten Standardkonvertierung, gefolgt von der Ausführung eines benutzerdefinierten, impliziten Konvertierungsoperators, gefolgt von einer weiteren optionalen, impliziten Standardkonvertierung. Die genauen Regeln zur Auswertung benutzerdefinierter impliziter Konvertierungen werden in §6.4.4 beschrieben.

### Konvertierungen anonymer Funktionen und Methodengruppen

Anonyme Funktionen und Methodengruppen enthalten weder Typen noch verfügen sie über Typen von sich selbst, sie können aber implizit in Delegattypen oder Ausdrucksbaumstrukturtypen konvertiert werden. Weitere Informationen über Konvertierungen anonymer Funktionen und von Methodengruppen finden Sie in §6.5 bzw. §6.6.

## Explizite Konvertierungen

Die folgenden Konvertierungen sind als explizite Konvertierungen klassifiziert:

* Alle impliziten Konvertierungen.
* Explizite numerische Konvertierungen.
* Explizite Enumerationskonvertierungen.
* Explizite Konvertierungen, die NULL-Werte zulassen.
* Explizite Verweiskonvertierungen.
* Explizite Schnittstellenkonvertierungen.
* Unboxingkonvertierungen.
* Explizite dynamische Konvertierungen
* Benutzerdefinierte explizite Konvertierungen.

Explizite Konvertierungen können in Umwandlungsausdrücken (§7.7.6) auftreten.

Die Gruppe der expliziten Konvertierungen umfasst alle impliziten Konvertierungen. Das bedeutet, dass redundante Umwandlungsausdrücke zulässig sind.

Bei expliziten Konvertierungen, die nicht implizit sind, kann nicht davon ausgegangen werden, dass sie immer erfolgreich sind. Sie können zu Datenverlusten führen und darüber hinaus über unterschiedliche Typdomänen hinweg explizite Notationen erfordern.

### Explizite numerische Konvertierungen

Explizite numerische Konvertierungen sind Konvertierungen von einem numeric-type in einen anderen numeric-type, für den noch keine implizite numerische Konvertierung (§6.1.2) vorhanden ist:

* Von sbyte in byte, ushort, uint, ulong oder char
* Von byte in sbyte und char.
* Von short in sbyte, byte, ushort, uint, ulong oder char
* Von ushort in sbyte, byte, short oder char.
* Von int in sbyte, byte, short, ushort, uint, ulong oder char
* Von uint in sbyte, byte, short, ushort, int oder char
* Von long in sbyte, byte, short, ushort, int, uint, ulong oder char.
* Von ulong in sbyte, byte, short, ushort, int, uint, long oder char.
* Von char in sbyte, byte oder short
* Von float in sbyte, byte, short, ushort, int, uint, long, ulong, char oder decimal.
* Von double in sbyte, byte, short, ushort, int, uint, long, ulong, char, float oder decimal.
* Von decimal in sbyte, byte, short, ushort, int, uint, long, ulong, char, float oder double.

Da explizite Konvertierungen alle impliziten und expliziten numerischen Konvertierungen einschließen, ist es jederzeit möglich, einen beliebigen numeric-type mithilfe des Umwandlungsausdrucks (§7.7.6) in einen anderen numeric-type zu konvertieren.

Explizite Konvertierungen können zu Informationsverlusten führen und Ausnahmen auslösen. Eine explizite, numerische Konvertierung wird folgendermaßen verarbeitet:

* Bei der Konvertierung eines Ganzzahltyps in einen anderen Ganzzahltyp ist die Verarbeitung vom Kontext der Überlaufprüfung (§7.6.12) abhängig, in dem die Konvertierung stattfindet:
* In einem checked-Kontext ist die Konvertierung erfolgreich, wenn der Wert des Quelloperanden sich im Bereich des Zieltyps bewegt; sie löst jedoch eine System.OverflowException aus, wenn der Wert des Quelloperanden außerhalb des Bereichs für den Zieltyp liegt.
* In einem unchecked-Kontext ist die Konvertierung immer erfolgreich und wird folgendermaßen fortgesetzt:
* Wenn der Quelltyp größer als der Zieltyp ist, wird der Quellwert gekürzt, indem die überzähligen, höchstwertigen Bits verworfen werden. Das Ergebnis wird dann als Wert des Zieltyps betrachtet.
* Wenn der Quelltyp kleiner als der Zieltyp ist, wird der Quellwert entweder durch ein Vorzeichen oder durch eine 0 erweitert, sodass er dieselbe Größe wie der Zieltyp besitzt. Die Vorzeichenerweiterung wird verwendet, wenn der Quelltyp vorzeichenbehaftet ist. Die 0-Erweiterung kommt bei vorzeichenlosen Quelltypen zum Einsatz. Das Ergebnis wird dann als Wert des Zieltyps betrachtet.
* Wenn der Quelltyp die gleiche Größe wie der Zieltyp hat, wird der Quellwert als Wert des Zieltyps betrachtet.
* Bei der Konvertierung vom Typ decimal in einen ganzzahligen Typ wird der Quellwert gegen 0 auf den nächsten Ganzzahlwert gerundet. Dieser Ganzzahlwert ist dann das Ergebnis der Konvertierung. Befindet sich der resultierende ganzzahlige Wert außerhalb des Zieltypbereichs, wird eine System.OverflowException ausgelöst.
* Bei der Konvertierung von float oder double in einen Ganzzahltyp ist die Verarbeitung vom Kontext der Überlaufprüfung (§7.6.12) abhängig, in dem die Konvertierung stattfindet:
* In einem checked-Kontext wird die Konvertierung wie folgt durchgeführt:
* Wenn der Wert des Operanden NaN (Not a Number) oder unendlich ist, wird eine System.OverflowException ausgelöst.
* Andernfalls wird der Quelloperand gegen 0 auf den nächsten Ganzzahlwert gerundet. Wenn dieser Ganzzahlwert innerhalb des Zieltypbereichs liegt, wird dieser Wert als Ergebnis der Konvertierung ausgegeben.
* Andernfalls wird eine System.OverflowException ausgelöst.
* In einem unchecked-Kontext ist die Konvertierung immer erfolgreich und wird folgendermaßen fortgesetzt:
* Wenn der Wert des Operanden NaN oder unendlich ist, ist das Ergebnis der Konvertierung ein nicht spezifizierter Wert des Zieltyps.
* Andernfalls wird der Quelloperand gegen 0 auf den nächsten Ganzzahlwert gerundet. Wenn dieser Ganzzahlwert innerhalb des Zieltypbereichs liegt, wird dieser Wert als Ergebnis der Konvertierung ausgegeben.
* Andernfalls ist das Ergebnis der Konvertierung ein nicht spezifizierter Wert des Zieltyps.
* Bei einer Konvertierung von double in float wird der double-Wert auf den nächsten float-Wert gerundet. Wenn der double-Wert zu klein für die Darstellung als float-Wert ist, ist das Ergebnis positiv oder negativ 0. Wenn der double-Wert zu groß für die Darstellung als float-Wert ist, ist das Ergebnis positive oder negative Unendlichkeit. Wenn der double-Wert NaN ist, ist das Ergebnis ebenfalls NaN.
* Bei einer Konvertierung von float oder double in decimal wird der Ausgangswert in die decimal-Darstellung konvertiert und ggf. auf die nächste Zahl hinter der 28. Dezimalstelle gerundet (§4.1.7). Wenn der Quellwert für die decimal-Darstellung zu klein ist, ist das Ergebnis gleich 0. Wenn der Quellwert NaN unendlich bzw. für die decimal-Darstellung zu groß ist, wird eine System.OverflowException ausgelöst.
* Bei einer Konvertierung von decimal in float oder double wird der decimal-Wert auf den nächsten double- oder float-Wert gerundet. Die Konvertierung kann zwar zu einem Verlust an Genauigkeit führen, es wird jedoch nie eine Ausnahme ausgelöst.

### Explizite Enumerationskonvertierungen

Folgende Enumerationskonvertierungen gelten als explizit:

* Von sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double oder decimal in einen beliebigen enum-type.
* Von einem beliebigen enum-type in sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double oder decimal.
* Von einem beliebigen enum-type in einen beliebigen anderen enum-type.

Bei einer expliziten Enumerationskonvertierung zwischen zwei Typen wird jeder beteiligte enum-type wie der zugrunde liegende Typ dieses enum-type behandelt. Danach wird eine implizite oder explizite numerische Konvertierung zwischen den resultierenden Typen ausgeführt. Beispiel: Gegeben ist ein enum-type E mit einem zugrunde liegenden Typ int. Eine Konvertierung von E in byte wird als explizite numerische Konvertierung (§6.2.1) von int in byte und eine Konvertierung von byte in E als implizite numerische Konvertierung (§6.1.2) von byte in int durchgeführt.

### Explizite Konvertierungen, die NULL-Werte zulassen

Mit expliziten Konvertierungen, die NULL-Werte zulassen, können vordefinierte Konvertierungen, die Werttypen verarbeiten, die keine NULL-Werte zulassen, auch für Formen dieser Typen, die NULL-Werte zulassen, verwendet werden. Für alle vordefinierten expliziten Konvertierungen von einem Werttyp S, der keine NULL-Werte zulässt, in einen Werttyp T, der keine NULL-Werte zulässt, (§6.1.1, §6.1.2, §6.1.3, §6.2.1 und §6.2.2) sind die folgenden Konvertierungen, die NULL-Werte zulassen, verfügbar:

* Eine explizite Konvertierung von S? in T?.
* Eine explizite Konvertierung von S in T?.
* Eine explizite Konvertierung von S? in T.

Die Auswertung einer Konvertierung, die NULL-Werte zulässt und auf einer zugrunde liegenden Konvertierung von S in T basiert, läuft folgendermaßen ab:

* Wenn die Konvertierung, die NULL-Werte zulässt, von S? in T? erfolgt:
* Wenn der Ausgangswert NULL ist (die HasValue-Eigenschaft ist false), ist das Ergebnis der NULL-Wert vom Typ T?.
* Andernfalls wird die Konvertierung als Entwrappen von S? in S ausgewertet, anschließend wird die zugrunde liegende Konvertierung von S in T durchgeführt, gefolgt von einem Wrapping von T in T?.
* Wenn die Konvertierung, die NULL-Werte zulässt, von S in T? erfolgt, wird die Konvertierung als die zugrunde liegende Konvertierung von S in T gefolgt von einem Wrapping von T in T? ausgewertet.
* Wenn die Konvertierung, die NULL-Werte zulässt, von S? in T erfolgt, wird die Konvertierung als Entwrappen von S? in S gefolgt von der zugrunde liegenden Konvertierung von S in T ausgewertet.

Beachten Sie, dass beim Entwrappen eines Werts, der NULL-Werte zulässt, eine Ausnahme ausgelöst wird, wenn der Wert null ist.

### Explizite Verweiskonvertierungen

Als explizite Verweiskonvertierungen gelten folgende:

* Von object und dynamic in einen beliebigen anderen reference-type.
* Von einem beliebigen class-type S in einen beliebigen class-type T, vorausgesetzt, dass S die Basisklasse von T ist.
* Von einem beliebigen class-type S in einen beliebigen interface-type T, vorausgesetzt, dass S nicht versiegelt ist und S nicht von T implementiert wird.
* Von einem beliebigen interface-type S in einen beliebigen class-type T, vorausgesetzt, dass T nicht versiegelt ist oder S von T implementiert wird.
* Von einem beliebigen interface-type S in einen beliebigen interface-type T, vorausgesetzt, dass S nicht von T abgeleitet ist.
* Von einem array-type S mit einem Elementtyp SE in einen array-type T mit einem Elementtyp TE, vorausgesetzt, dass alle folgenden Anforderungen erfüllt sind:
* S und T unterscheiden sich nur hinsichtlich des Elementtyps. Das bedeutet, dass S und T dieselbe Anzahl von Dimensionen haben.
* Hier handelt es sich sowohl bei SE als auch bei TE um reference-types.
* Es ist eine explizite Verweiskonvertierung von SE in TE vorhanden.
* Von System.Array und den implementierten Schnittstellen in einen beliebigen array-type.
* Von einem eindimensionalen Arraytyp S[] in System.Collections.Generic.IList<T> und seine Basisschnittstellen, sofern eine explizite Verweiskonvertierung von S in T vorhanden ist.
* Von System.Collections.Generic.IList<S> und seinen Basisschnittstellen in einen eindimensionalen Arraytyp T[], sofern eine explizite Identitäts- oder Verweiskonvertierung von S in T vorhanden ist.
* Von System.Delegate und den implementierten Schnittstellen in einen beliebigen delegate-type.
* Von einem Verweistyp in einen Verweistyp T, wenn er eine explizite Verweiskonvertierung in einen Verweistyp T0 aufweist und T0 eine Identitätskonvertierung in T aufweist.
* Von einem Verweistyp in einen Schnittstellen- oder Delegattyp T, wenn er eine explizite Verweiskonvertierung in einen Schnittstellen- oder Delegattyp T0 aufweist und entweder T0 varianzkonvertierbar in T ist oder T varianzkonvertierbar in T0 (§13.1.3.2) ist.
* Von D<S1…Sn> ino D<T1…Tn> D<T1…Tn>, wobei D<X1…Xn> ein generischer Delegattyp ist, D<S1…Sn> nicht mit D<T1…Tn> kompatibel oder identisch ist und für jeden Typparameter Xi von D Folgendes gilt:
* Wenn Xi invariant ist, dann ist Si mit Ti identisch.
* Wenn Xi kovariant ist, dann ist eine implizite oder explizite Identitäts- oder Verweiskonvertierung von Si in Ti vorhanden.
* Ist Xi kontravariant, dann sind Si und Ti entweder identisch oder beides Verweistypen.
* Explizite Konvertierungen mit Typparametern, von denen bekannt ist, dass sie Verweistypen sind. Weitere Informationen über explizite Konvertierungen mit Typparametern finden Sie in §6.2.7.

Explizite Verweiskonvertierungen sind jene Konvertierungen zwischen reference-types, für die Laufzeitüberprüfungen erforderlich sind.

Damit eine explizite Verweiskonvertierung zur Laufzeit erfolgreich ist, muss entweder der Wert des Quelloperanden null sein, oder es muss sich bei dem eigentlichen Typ des vom Quelloperanden referenzierten Objekts um einen Typ handeln, der mithilfe einer impliziten Verweiskonvertierung (§6.1.6) oder Boxingkonvertierung (§6.1.7) in den Zieltyp konvertiert werden kann. Wenn eine explizite Verweiskonvertierung fehlschlägt, wird eine System.InvalidCastException ausgelöst.

Verweiskonvertierungen (implizit oder explizit) ändern niemals die referenzielle Identität des konvertierten Objekts. Anders ausgedrückt kann eine Verweiskonvertierung zwar den Verweistyp, niemals aber den Typ oder Wert des referenzierten Objekts ändern.

### Unboxingkonvertierungen

Eine Unboxingkonvertierung ermöglicht die explizite Konvertierung eines Verweistyps in einen value-type. Eine Unboxingkonvertierung ist von den Typen object, dynamic und System.ValueType in einen beliebigen non-nullable-value-type vorhanden sowie von einem beliebigen interface-type in einen beliebigen non-nullable-value-type, der den interface-type implementiert. Darüber hinaus kann eine Unboxingkonvertierung des Typs System.Enum in einen beliebigen enum-type durchgeführt werden.

Eine Unboxingkonvertierung ermöglicht die Konvertierung eines Verweistyps in einen nullable-type, wenn eine Unboxingkonvertierung aus dem Verweistyp in den zugrunde liegenden non-nullable-value-type des nullable-type vorhanden ist.

Ein Werttyp S weist eine Unboxingkonvertierung von einem Schnittstellentyp I auf, wenn er eine Unboxingkonvertierung von einem Schnittstellentyp I0 aufweist und I0 eine Identitätskonvertierung in I aufweist.

Ein Werttyp S weist eine Unboxingkonvertierung von einem Schnittstellentyp I auf, wenn er eine Unboxingkonvertierung von einem Schnittstellen- oder Delegattyp I0 aufweist und entweder I0 varianzkonvertierbar in I ist oder I varianzkonvertierbar in I0 ist (§13.1.3.2).

Bei einer Unboxingoperation wird zuerst überprüft, ob die Objektinstanz ein mittels Boxing gepackter Wert des angegebenen value-type ist, und anschließend wird der Wert aus der Instanz kopiert. Durch Unboxing eines NULL-Verweises auf einen nullable-type wird der NULL-Wert des nullable-type generiert. Das Unboxing einer Struktur aus dem Typ System.ValueType ist möglich, da dies die Basisklasse aller Strukturen (§11.3.2) ist.

Unboxingkonvertierungen werden ausführlich in §4.3.2 beschrieben.

### Explizite dynamische Konvertierungen

Es ist eine explizite dynamische Konvertierung von einem Ausdruck vom Typ dynamic in einen beliebigen Typ T vorhanden. Die Konvertierung wird dynamisch gebunden (§7.2.2), d. h. dass zur Laufzeit eine explizite Konvertierung vom Laufzeittyp des Ausdrucks in T gesucht wird. Wenn keine Konvertierung gefunden wird, wird eine Laufzeitausnahme ausgelöst.

Wenn eine dynamische Bindung der Konvertierung nicht erwünscht ist, kann der Ausdruck zuerst in object und anschließend in den gewünschten Typ konvertiert werden.

Angenommen, die folgende Klasse ist definiert:

class C  
{  
 int i;

public C(int i) { this.i = i; }

public static explicit operator C(string s)   
 {  
 return new C(int.Parse(s));  
 }  
}

Das folgende Beispiel veranschaulicht explizite dynamische Konvertierungen:

object o = "1";  
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails  
var c2 = (C)d; // Compiles and user defined conversion succeeds

Die beste während der Kompilierung gefundene Konvertierung von o in C ist eine explizite Verweiskonvertierung. Hierbei tritt zur Laufzeit ein Fehler auf, da es sich bei “1” nicht um ein C handelt. Da es sich um eine explizite dynamische Konvertierung handelt, wird die Konvertierung von d in C jedoch bis zur Laufzeit unterbrochen, wobei eine benutzerdefinierte Konvertierung des Laufzeittyps von d – string – in C gefunden wird. Die Ausführung ist erfolgreich.

### Explizite Konvertierungen mit Typparametern

Für einen bestimmten Typparameter T sind die folgenden expliziten Konvertierungen vorhanden:

* Die effektive Basisklasse C von T in T und eine beliebige Basisklasse von C in T. Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Unboxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als explizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* Von einem beliebigen Schnittstellentyp in T. Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Unboxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als explizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* T in einen beliebigen interface-type I, vorausgesetzt, es ist nicht bereits eine implizite Konvertierung von T in I vorhanden. Wenn T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Boxingkonvertierung, gefolgt von einer expliziten Verweiskonvertierung, ausgeführt. Andernfalls wird die Konvertierung als explizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.
* Von einem Typparameter U in T, vorausgesetzt, dass T von U abhängig ist (§10.1.5). Wenn U ein Werttyp ist, weisen T und U zur Laufzeit notwendigerweise denselben Typ auf, und es wird keine Konvertierung durchgeführt. Wenn andernfalls T ein Werttyp ist, wird die Konvertierung zur Laufzeit als Unboxingkonvertierung ausgeführt. Andernfalls wird die Konvertierung als explizite Verweiskonvertierung oder Identitätskonvertierung ausgeführt.

Wenn von T bekannt ist, dass es ein Verweistyp ist, gelten alle genannten Konvertierungen als explizite Verweiskonvertierung (§6.2.4). Wenn T nicht als Verweistyp bekannt ist, gelten die genannten Konvertierungen als Unboxingkonvertierungen (§6.2.5).

Die genannten Regeln erlauben keine direkte explizite Konvertierung eines nicht eingeschränkten Typparameters in einen Nicht-Schnittstellen-Typ, was möglicherweise nicht den Erwartungen entspricht. Der Grund hierfür ist, dass Verwirrung vermieden werden soll und die Semantik solcher Konvertierungen deutlich werden soll. Betrachten Sie z. B. die folgende Deklaration:

class X<T>  
{  
 public static long F(T t) {  
 return (long)t; // Error   
 }  
}

Wenn eine direkte explizite Konvertierung von t in int zulässig wäre, könnte leicht erwartet werden, dass X<int>.F(7) den Wert 7L zurückgibt. Dies wäre jedoch nicht der Fall, weil die numerischen Standardkonvertierungen nur berücksichtigt werden, wenn von den Typen zur Bindungszeit bekannt ist, dass sie numerisch sind. Um die Semantik deutlich zu machen, muss das oben stehende Beispiel hingegen wie folgt geschrieben werden:

class X<T>  
{  
 public static long F(T t) {  
 return (long)(object)t; // Ok, but will only work when T is long  
 }  
}

Dieser Code kann zwar kompiliert werden, aber durch Ausführen von X<int>.F(7) wird nun zur Laufzeit eine Ausnahme ausgelöst, da ein int nach Boxing nicht direkt in einen long konvertiert werden kann.

### Benutzerdefinierte explizite Konvertierungen

Eine benutzerdefinierte, explizite Konvertierung besteht aus einer optionalen, expliziten Standardkonvertierung, gefolgt von der Ausführung eines benutzerdefinierten, impliziten oder expliziten Konvertierungsoperators, gefolgt von einer weiteren optionalen, expliziten Standardkonvertierung. Die genauen Regeln zur Auswertung benutzerdefinierter expliziter Konvertierungen werden in §6.4.5 beschrieben.

## Standardkonvertierungen

Standardkonvertierungen sind jene vordefinierten Konvertierungen, die als Teil einer benutzerdefinierten Konvertierung auftreten können.

### Implizite Standardkonvertierungen

Die folgenden impliziten Konvertierungen sind als implizite Standardkonvertierungen klassifiziert:

* Identitätskonvertierungen (§6.1.1)
* Implizite numerische Konvertierungen (§6.1.2)
* Implizite Konvertierungen, die NULL-Werte zulassen (§6.1.4)
* Implizite Verweiskonvertierungen (§6.1.6)
* Boxingkonvertierungen (§6.1.7)
* Implizite Konvertierungen konstanter Ausdrücke (§6.1.8)
* Implizite Konvertierungen mit Typparametern (§6.1.10)

Die impliziten Standardkonvertierungen schließen vor allem benutzerdefinierte, implizite Konvertierungen aus.

### Explizite Standardkonvertierungen

Explizite Standardkonvertierungen sind alle impliziten Standardkonvertierungen sowie ein Teil der expliziten Konvertierungen, für die eine entgegengesetzte implizite Standardkonvertierung vorhanden ist. Das bedeutet, dass es bei Vorhandensein einer impliziten Standardkonvertierung vom Typ A in den Typ B sowohl eine explizite Standardkonvertierung vom Typ A in den Typ B als auch eine explizite Standardkonvertierung vom Typ B in den Typ A gibt.

## Benutzerdefinierte Konvertierungen

C# bietet die Möglichkeit, vordefinierte implizite und explizite Konvertierungen durch benutzerdefinierte Konvertierungen zu erweitern. Benutzerdefinierte Konvertierungen werden durch die Deklaration von Konvertierungsoperatoren (§10.10.3) in Klassen- und Strukturtypen eingeführt.

### Zugelassene benutzerdefinierte Konvertierungen

C# gestattet nur die Deklaration bestimmter benutzerdefinierter Konvertierungen. Insbesondere ist es nicht möglich, eine bereits vorhandene implizite oder explizite Konvertierung neu zu definieren.

Für einen angegebenen Quelltyp S und einen Zieltyp T, wobei S oder T Typen sind, die NULL-Werte zulassen, verweisen S0 und T0 auf ihre zugrunde liegenden Typen, andernfalls sind S0und T0 gleich S bzw. T. Eine Klasse oder Struktur darf eine Konvertierung von einem Quelltyp S in einen Zieltyp T nur deklarieren, wenn die folgenden Voraussetzungen erfüllt sind:

* S0 und T0 sind unterschiedliche Typen.
* Der Klassen- oder Strukturtyp, in dem die Operatordeklaration erfolgt, ist entweder S0 oder T0.
* Weder S0 noch T0 ist als interface-type angegeben.
* Mit Ausnahme von benutzerdefinierten Konvertierungen sind keine weiteren Konvertierungen von S in T oder von T in S vorhanden.

Die für benutzerdefinierte Konvertierungen geltenden Beschränkungen werden ausführlich in §10.10.3 beschrieben.

### Heraufgestufte Konvertierungsoperatoren

Für einen bestimmten benutzerdefinierten Konvertierungsoperator für eine Konvertierung von einem Werttyp S, der keine NULL-Werte zulässt, in einen Werttyp T, der keine NULL-Werte zulässt, gibt es einen heraufgestuften Konvertierungsoperator für die Konvertierung von S? in T?. Der heraufgestufte Konvertierungsoperator führt das Entwrappen von S? in S gefolgt von der benutzerdefinierten Konvertierung von S in T durch. Anschließend wird ein Wrapping von T in T? durchgeführt, wobei jedoch S? mit einem NULL-Wert direkt in T? mit einem NULL-Wert konvertiert wird.

Ein heraufgestufter Konvertierungsoperator verfügt über dieselbe implizite oder explizite Klassifizierung wie der zugrunde liegende benutzerdefinierte Konvertierungsoperator. Der Begriff „benutzerdefinierte Konvertierung“ bezieht sich auf die Verwendung von sowohl benutzerdefinierten als auch heraufgestuften Konvertierungsoperatoren.

### Auswertung benutzerdefinierter Konvertierungen

Bei einer benutzerdefinierten Konvertierung wird ein Wert aus seinem Typ, dem Quelltyp, in einen anderen Typ, den Zieltyp, konvertiert. Zentraler Punkt der Auswertung einer benutzerdefinierten Konvertierung ist das Erkennen des am genauesten zutreffenden benutzerdefinierten Konvertierungsoperators für den speziellen Quell- und Zieltyp. Dieser Vorgang vollzieht sich in mehreren Schritten:

* Auffinden der Gruppe von Klassen und Strukturen, für die benutzerdefinierte Konvertierungsoperatoren in Betracht kommen. Diese Gruppe besteht aus dem Quelltyp und seinen Basisklassen sowie dem Zieltyp und dessen Basisklassen (hierbei wird implizit vorausgesetzt, dass benutzerdefinierte Operatoren nur von Klassen und Strukturen deklariert werden können und dass Nicht-Klassentypen keine Basisklassen besitzen). Bei diesem Schritt wird, wenn entweder der Quell- oder der Zieltyp ein nullable-type ist, stattdessen der zugrunde liegende Typ verwendet.
* Ausgehend von dieser Gruppe von Typen: Auffinden jener benutzerdefinierten und heraufgestuften Konvertierungsoperatoren, die anwendbar sind. Ein Konvertierungsoperator ist nur anwendbar, wenn er eine Standardkonvertierung (§6.3) aus dem Quelltyp in den Operandentyp des Operators und aus dem Ergebnistyp des Operators in den Zieltyp ausführen kann.
* Ermitteln des unzweifelhaft am besten zutreffenden Operators aus der Gruppe der anwendbaren benutzerdefinierten Operatoren. Prinzipiell ist der Operator am spezifischsten, dessen Operandentyp dem Quelltyp am nächsten kommt und dessen Ergebnistyp dem Zieltyp am nächsten kommt. Benutzerdefinierte Konvertierungsoperatoren haben eine höhere Priorität als heraufgestufte Konvertierungsoperatoren. Die genauen Regeln zum Bestimmen des zutreffendsten benutzerdefinierten Konvertierungsoperators werden in den folgenden Abschnitten erläutert.

Nachdem der zutreffendste benutzerdefinierte Konvertierungsoperator ermittelt wurde, wird die aus bis zu drei Schritten bestehende benutzerdefinierte Konvertierung ausgeführt:

* Zuerst wird, sofern erforderlich, eine Standardkonvertierung aus dem Quelltyp in den Operandentyp des benutzerdefinierten oder heraufgestuften Konvertierungsoperators ausgeführt.
* Als Nächstes wird der benutzerdefinierte oder heraufgestufte Konvertierungsoperator zum Ausführen der Konvertierung aufgerufen.
* Abschließend wird ggf. eine Standardkonvertierung aus dem Ergebnistyp des benutzerdefinierten oder heraufgestuften Konvertierungsoperators in den Zieltyp ausgeführt.

Die Auswertung einer benutzerdefinierten Konvertierung umfasst immer nur einen benutzerdefinierten oder heraufgestuften Konvertierungsoperator. Das bedeutet, dass bei einer Konvertierung von einem Typ S in einen Typ T nicht erst eine benutzerdefinierte Konvertierung von S in X und anschließend eine benutzerdefinierte Konvertierung von X in T ausgeführt wird.

Genaue Definitionen für die Auswertung benutzerdefinierter impliziter oder expliziter Konvertierungen finden Sie in den folgenden Abschnitten. In den Definitionen werden die folgenden Begriffe verwendet:

* Wenn eine implizite Standardkonvertierung (§6.3.1) aus dem Typ A in den Typ B vorhanden ist und weder A noch B interface-types sind, dann wird A als von B umfasst und B gegenüber A als umfassend bezeichnet.
* Der am stärksten umfassende Typ aus einer Gruppe von Typen ist jener, der alle anderen Typen dieser Gruppe umfasst. Wenn kein Typ vorhanden ist, der alle anderen Typen umfasst, besitzt die Gruppe keinen am stärksten umfassenden Typ. Vereinfachend könnte man sagen, dass der umfassendste Typ der „größte“ Typ der Gruppe ist, d. h. jener Typ, in den alle anderen Typen implizit konvertiert werden können.
* Der am stärksten umfasste Typ aus einer Gruppe von Typen ist jener, der von allen anderen Typen dieser Gruppe umfasst wird. Wenn kein Typ vorhanden ist, der von allen anderen Typen umfasst wird, besitzt die Gruppe keinen umfasstesten Typ. Vereinfachend könnte man sagen, dass der umfassteste Typ der „kleinste“ Typ der Gruppe ist, d. h. jener Typ, der in alle anderen Typen implizit konvertiert werden kann.

### Benutzerdefinierte implizite Konvertierungens

Eine benutzerdefinierte implizite Konvertierung vom Typ S in den Typ T wird folgendermaßen verarbeitet:

* Bestimmen der Typen S0 und T0. Wenn S oder T Typen sind, die NULL-Werte zulassen, sind S0 und T0 ihre zugrunde liegenden Typen, andernfalls sind S0 und T0 gleich S bzw. T.
* Auffinden der Gruppe von Typen (D), für die benutzerdefinierte Konvertierungsoperatoren in Betracht kommen. Diese Gruppe besteht aus S0 (wenn S0 eine Klasse oder Struktur ist), den Basisklassen von S0 (wenn S0 eine Klasse ist) und T0 (wenn T0 eine Klasse oder Struktur ist).
* Auffinden der Gruppe der anwendbaren benutzerdefinierten und heraufgestuften Operatoren (U). Diese Gruppe besteht aus den von den Klassen oder Strukturen in D deklarierten benutzerdefinierten und heraufgestuften impliziten Konvertierungsoperatoren, die einen S umfassenden Typ in einen von T umfassten Typ konvertieren. Wenn U leer ist, gilt die Konvertierung als nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Auffinden des von den Operatoren in U am meisten zutreffenden Quelltypen (SX):
* Wenn einer der Operatoren in U aus S konvertiert wird, dann ist SX gleich S.
* Andernfalls ist SX der am stärksten umfasste Typ in der kombinierten Gruppe von Quelltypen der Operatoren in U. Wenn genau ein am stärksten umfasster Typ nicht gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Auffinden des für die Operatoren in TX am meisten zutreffenden Zieltypen U:
* Wenn einer der Operatoren in U nach T konvertiert wird, dann ist TX gleich T.
* Andernfalls ist TX der am stärksten umfassende Typ in der kombinierten Gruppe von Zieltypen der Operatoren in U. Wenn genau ein am stärksten umfassender Typ nicht gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Suchen des am meisten zutreffenden Konvertierungsoperators:
* Wenn U genau einen benutzerdefinierten Konvertierungsoperator enthält, der von SX in TX konvertiert, dann ist dieser der spezifischste Konvertierungsoperator.
* Wenn andernfalls U genau einen heraufgestuften Konvertierungsoperator enthält, der von SX in TX konvertiert, dann ist dieser der spezifischste Konvertierungsoperator.
* Andernfalls ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Abschließend, Anwenden der Konvertierung:
* Wenn S nicht SX ist, wird eine implizite Standardkonvertierung von S in SX ausgeführt.
* Zur Konvertierung von SX in TX wird der spezifischste Konvertierungsoperator aufgerufen.
* Wenn TX nicht T ist, wird eine implizite Standardkonvertierung von TX in T ausgeführt.

### Benutzerdefinierte explizite Konvertierungens

Eine benutzerdefinierte explizite Konvertierung vom Typ S in den Typ T wird folgendermaßen verarbeitet:

* Bestimmen der Typen S0 und T0. Wenn S oder T Typen sind, die NULL-Werte zulassen, sind S0 und T0 ihre zugrunde liegenden Typen, andernfalls sind S0 und T0 gleich S bzw. T.
* Auffinden der Gruppe von Typen (D), für die benutzerdefinierte Konvertierungsoperatoren in Betracht kommen. Diese Gruppe besteht aus S0 (wenn S0 eine Klasse oder Struktur ist), den Basisklassen von S0 (wenn S0 eine Klasse ist), T0 (wenn T0 eine Klasse oder Struktur ist) und den Basisklassen von T0 (wenn T0 eine Klasse ist).
* Auffinden der Gruppe der anwendbaren benutzerdefinierten und heraufgestuften Operatoren (U). Diese Gruppe besteht aus den von den Klassen oder Strukturen in D deklarierten benutzerdefinierten und heraufgestuften impliziten oder expliziten Konvertierungsoperatoren, die einen S umfassenden oder von S umfassten Typ in einen T umfassenden oder von T umfassten Typ konvertieren. Wenn U leer ist, gilt die Konvertierung als nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Auffinden des von den Operatoren in U am meisten zutreffenden Quelltypen (SX):
* Wenn einer der Operatoren in U aus S konvertiert wird, dann ist SX gleich S.
* Wenn andererseits einer der Operatoren in U aus S umfassenden Typen konvertiert, dann ist SX der am stärksten umfasste Typ in der kombinierten Gruppe von Quelltypen dieser Operatoren. Wenn kein am stärksten umfasster Typ gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls ist SX der am stärksten umfassende Typ in der kombinierten Gruppe von Quelltypen der Operatoren in U. Wenn genau ein am stärksten umfassender Typ nicht gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Auffinden des für die Operatoren in TX am meisten zutreffenden Zieltypen U:
* Wenn einer der Operatoren in U nach T konvertiert wird, dann ist TX gleich T.
* Wenn andererseits einer der Operatoren in U in von T umfasste Typen konvertiert, dann ist TX der am stärksten umfassende Typ in der kombinierten Gruppe von Zieltypen dieser Operatoren. Wenn genau ein am stärksten umfassender Typ nicht gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls ist TX der am stärksten umfasste Typ in der kombinierten Gruppe von Zieltypen der Operatoren in U. Wenn kein am stärksten umfasster Typ gefunden werden kann, ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Suchen des am meisten zutreffenden Konvertierungsoperators:
* Wenn U genau einen benutzerdefinierten Konvertierungsoperator enthält, der von SX in TX konvertiert, dann ist dieser der spezifischste Konvertierungsoperator.
* Wenn andernfalls U genau einen heraufgestuften Konvertierungsoperator enthält, der von SX in TX konvertiert, dann ist dieser der spezifischste Konvertierungsoperator.
* Andernfalls ist die Konvertierung mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Abschließend, Anwenden der Konvertierung:
* Wenn S nicht SX ist, wird eine explizite Standardkonvertierung von S in SX ausgeführt.
* Zur Konvertierung von SX in TX wird der spezifischste benutzerdefinierte Konvertierungsoperator aufgerufen.
* Wenn TX nicht T ist, wird eine explizite Standardkonvertierung von TX in T ausgeführt.

## Konvertierungen anonymer Funktionen

Eine anonymous-method-expression oder lambda-expression wird als anonyme Funktion (§7.15) klassifiziert. Der Ausdruck weist keinen Typ auf, er kann jedoch implizit in einen kompatiblen Delegattyp oder einen Ausdrucksbaumstrukturtyp konvertiert werden. Eine anonyme Funktion F ist insbesondere dann mit einem Delegattyp D kompatibel, wenn Folgendes gilt:

* Wenn F eine anonymous-function-signature enthält, verfügen D und F über dieselbe Anzahl von Parametern.
* Wenn F keine anonymous-function-signature enthält, kann D 0 (null) oder mehr Parameter von einem beliebigen Typ enthalten, solange kein Parameter von D den Parametermodifizierer out besitzt.
* Wenn F über eine explizit typisierte Parameterliste verfügt, besitzen sämtliche Parameter in D denselben Typ und dieselben Modifizierer wie der entsprechende Parameter in F.
* Wenn F über eine implizit typisierte Parameterliste verfügt, besitzt D keine ref- oder out-Parameter.
* Wenn der Text von F ein Ausdruck ist und entweder D einen void-Rückgabetyp aufweist oder F "async" ist und D den Rückgabetyp Task aufweist und wenn jeder Parameter von F den Typ des entsprechenden Parameters in D angibt, ist der Text von F ein gültiger Ausdruck (siehe §7), der als statement-expression (§8.6) zulässig ist.
* Wenn der Text von F ein Anweisungsblock ist und entweder D einen void-Rückgabetyp aufweist oder F "async" ist und D den Rückgabetyp Task aufweist und wenn jeder Parameter von F den Typ des entsprechenden Parameters in D angibt, ist der Text von F ein gültiger Anweisungsblock (siehe §8.2), in dem keine return-Anweisung einen Ausdruck angibt.
* Wenn der Text von F ein Ausdruck ist und *entweder* F nicht asynchron ist und D einen nicht leeren Rückgabetyp T hat, *oder* F asynchron ist und D einen Rückgabetyp Task<T> hat, dann ist, wenn jeder Parameter von F den Typ des entsprechenden Parameters in D angibt, der Text von F ein gültiger Ausdruck (siehe §7), der implizit in T konvertiert werden kann.
* Wenn der Text von F ein Anweisungsblock ist, und *entweder* F nicht asynchron ist und D einen nicht leeren Rückgabetyp T hat, *oder* F asynchron ist und D einen Task<T> hat, dann ist, wenn jeder Parameter von F den Typ des entsprechenden Parameters in D angibt, der Text von F ein gültiger Ausdrucksblock (swrt §8.2) mit einem nicht erreichbaren Endpunkt, in dem jede return-Anweisung einen Ausdruck angibt, der implizit in T konvertiert werden kann.

Aus Platzgründen wird in diesem Abschnitt für die Aufgabentypen Task und Task<T> (§10.14) die Kurzform verwendet.

Ein Lambda-Ausdruck F ist mit einem Ausdrucksbaumstrukturtyp Expression<D> kompatibel, wenn F mit dem Delegattyp D kompatibel ist. Dies gilt nicht für anonyme Methoden, sondern nur für Lambda-Ausdrücke.

Bestimmte Lambda-Ausdrücke können nicht in Ausdrucksbaumstrukturtypen konvertiert werden: Selbst wenn die Konvertierung vorhanden ist, tritt während der Kompilierung ein Fehler auf. Dies ist der Fall, wenn der Lambda-Ausdruck:

* Einen block-Text aufweist
* Enthält einfache oder Verbundzuweisungsoperatoren
* Einen dynamisch gebundenen Ausdruck enthält
* "async" ist

In den folgenden Beispielen wird der generische Delegattyp Func<A,R> verwendet, der eine Funktion darstellt, die ein Argument vom A akzeptiert und einen Wert vom Typ R zurückgibt:

delegate R Func<A,R>(A arg);

In den Zuweisungen

Func<int,int> f1 = x => x + 1; // Ok

Func<int,double> f2 = x => x + 1; // Ok

Func<double,int> f3 = x => x + 1; // Error

Func<int, Task<int>> f4 = async x => x + 1; // Ok

werden die Parameter- und Rückgabetypen der einzelnen anonymen Funktionen über den Typ der Variablen bestimmt, der die anonyme Funktion zugewiesen ist.

Die erste Zuweisung konvertiert die anonyme Funktion erfolgreich in den Delegattyp Func<int,int>, da x + 1 ein gültiger Ausdruck ist, der implizit in den Typ int konvertiert werden kann, wenn x der Typ int zugewiesen wird.

Entsprechen konvertiert die zweite Zuweisung die anonyme Funktion erfolgreich in den Delegattyp Func<int,double>, da das Ergebnis von x + 1 (vom Typ int) implizit in den Typ double konvertiert werden kann.

Die dritte Zuweisung führt hingegen zur Kompilierzeit zu einem Fehler, da das Ergebnis von x + 1 (vom Typ double) nicht implizit in den Typ int konvertiert werden kann, wenn x der Typ double zugeordnet wird.

Die vierte Zuweisung konvertiert die anonyme Funktion erfolgreich in den Delegattyp Func<int, Task<int>>, da das Ergebnis von x + 1 (vom Typ int) implizit in den Ergebnistyp int vom Aufgabentyp Task<int> konvertiert werden kann.

Anonyme Funktionen können Einfluss auf die Überladungsauflösung haben und bei Typrückschlüssen zum Einsatz kommen. In §7.5 finden Sie weitere Einzelheiten.

### Auswertung von Konvertierungen anonymer Funktionen in Delegattypen

Bei der Konvertierung einer anonymen Funktion in einen Delegattyp wird eine Delegatinstanz erstellt, die auf die anonyme Funktion und die (möglicherweise leere) Gruppe von erfassten äußeren Variablen verweist, die zum Zeitpunkt der Auswertung aktiv sind. Beim Aufruf des Delegaten wird der Text der anonymen Funktion ausgeführt. Der Code im Text wird mithilfe der Gruppe der erfassten äußeren Variablen ausgeführt, auf die vom Delegaten verwiesen wird.

Die von einer anonymen Funktion erstellte Aufrufliste eines Delegaten enthält einen einzigen Eintrag. Das genaue Zielobjekt und die genaue Zielmethode des Delegaten werden nicht angegeben. Insbesondere ist nicht festgelegt, ob das Zielobjekt des Delegaten null, der this-Wert des einschließenden Funktionsmembers oder ein anderes Objekt ist.

Konvertierungen semantisch identischer anonymer Funktionen mit derselben (möglicherweise leeren) Gruppe von erfassten äußeren Variableninstanzen in dieselben Delegattypen können (müssen jedoch nicht) dieselbe Delegatinstanz zurückgeben. „Semantisch identisch“ bedeutet hier, dass durch die Ausführung der anonymen Funktionen mit denselben Argumenten in jedem Fall dieselben Auswirkungen erzielt werden. Diese Regel ermöglicht das Optimieren beispielsweise folgenden Codes.

delegate double Function(double x);

class Test  
{  
 static double[] Apply(double[] a, Function f) {  
 double[] result = new double[a.Length];  
 for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);  
 return result;  
 }

static void F(double[] a, double[] b) {  
 a = Apply(a, (double x) => Math.Sin(x));  
 b = Apply(b, (double y) => Math.Sin(y));  
 ...  
 }  
}

Da die beiden anonymen Funktionsdelegaten über dieselbe (leere) Gruppe von erfassten äußeren Variablen verfügen und semantisch identisch sind, kann der Compiler festlegen, dass die Delegaten auf dieselbe Zielmethode verweisen. Tatsächlich kann der Compiler von beiden anonymen Funktionsausdrücken dieselbe Delegatinstanz zurückgeben lassen.

### Auswertung von Konvertierungen anonymer Funktionen in Ausdrucksbaumstrukturtypen

Die Konvertierung einer anonymer Funktion in einen Ausdrucksbaumstrukturtyp erzeugt eine Ausdrucksbaumstruktur (§4.6). Das bedeutet, dass die Auswertung der Konvertierung der anonymen Funktion das Erstellen einer Objektstruktur nach sich zieht, die die Struktur der anonymen Funktion selbst darstellt. Die genaue Struktur der Ausdrucksbaumstruktur und der genaue Ablauf der Erstellung sind durch die Implementierung definiert.

### Implementierungsbeispiel

In diesem Beispiel wird eine mögliche Implementierung von Konvertierungen anonymen Funktionen im Hinblick auf andere C#-Konstrukte beschrieben. Die hier erläuterte Implementierung beruht auf denselben Grundsätzen wie denen, die vom Microsoft C#-Compiler verwendet werden, sie stellt jedoch unter keinen Umständen eine zwingend erforderliche oder die einzig mögliche Implementierung dar. Dabei werden die Konvertierungen in Ausdrucksbaumstrukturen nur kurz beschrieben, da ihre Semantik nicht Inhalt dieser Spezifikation ist.

Im Rest dieses Abschnitts finden Sie eine Reihe von Codebeispielen, die anonyme Funktionen mit unterschiedlichen Eigenschaften enthalten. Für jedes Beispiel wird eine entsprechende Übersetzung in Code angegeben, in dem ausschließlich andere C#-Konstrukte verwendet werden. In den Beispielen wird der Bezeichner D durch die Darstellung des folgenden Delegattyps angenommen:

public delegate void D();

Eine anonyme Funktion in der einfachsten Form erfasst keine äußeren Variablen:

class Test  
{  
 static void F() {  
 D d = () => { Console.WriteLine("test"); };  
 }  
}

Dies kann in eine Delegatinstanziierung übersetzt werden, die auf eine vom Compiler generierte statische Methode verweist, in der der Code der anonymen Funktion platziert wird:

class Test  
{  
 static void F() {  
 D d = new D(\_\_Method1);  
 }

static void \_\_Method1() {  
 Console.WriteLine("test");  
 }  
}

Im folgenden Beispiel verweist die anonyme Funktion auf Instanzmember von this:

class Test  
{  
 int x;

void F() {  
 D d = () => { Console.WriteLine(x); };  
 }  
}

Dies kann in eine vom Compiler generierte Instanzmethode übersetzt werden, die den Code der anonymen Funktion enthält:

class Test  
{  
 int x;

void F() {  
 D d = new D(\_\_Method1);  
 }

void \_\_Method1() {  
 Console.WriteLine(x);  
 }  
}

In diesem Beispiel erfasst die anonyme Funktion eine lokale Variable:

class Test  
{  
 void F() {  
 int y = 123;  
 D d = () => { Console.WriteLine(y); };  
 }  
}

Die Lebensdauer der lokalen Variablen muss nun mindestens auf die Lebensdauer des anonymen Funktionsdelegaten verlängert werden. Dies kann durch „Herausheben“ der lokalen Variablen in ein Feld einer vom Compiler generierten Klasse erreicht werden. Das Instanziieren der lokalen Variablen (§7.15.5.2) entspricht dann dem Erstellen einer Instanz der vom Compiler generierten Klasse, und das Zugreifen auf die lokale Variable entspricht dem Zugreifen auf ein Feld in der Instanz der vom Compiler generierten Klasse. Zudem wird die anonyme Funktion zu einer Instanzmethode der vom Compiler generierten Klasse:

class Test  
{  
 void F() {  
 \_\_Locals1 \_\_locals1 = new \_\_Locals1();  
 \_\_locals1.y = 123;  
 D d = new D(\_\_locals1.\_\_Method1);  
 }

class \_\_Locals1  
 {  
 public int y;

public void \_\_Method1() {  
 Console.WriteLine(y);  
 }  
 }  
}

Schließlich erfasst die folgende anonyme Funktion this sowie zwei lokale Variablen mit unterschiedlicher Lebensdauer:

class Test  
{  
 int x;

void F() {  
 int y = 123;  
 for (int i = 0; i < 10; i++) {  
 int z = i \* 2;  
 D d = () => { Console.WriteLine(x + y + z); };  
 }  
 }  
}

Hier wird für jeden Anweisungsblock eine vom Compiler generierte Klasse erstellt, in dem lokale Variablen so erfasst werden, dass die lokalen Variablen in den verschiedenen Blöcken eine unabhängige Lebensdauer aufweisen können. Eine Instanz von \_\_Locals2, der vom Compiler generierten Klasse für den inneren Anweisungsblock, enthält die lokale Variable z und ein Feld, das auf eine Instanz von \_\_Locals1 verweist. Eine Instanz von \_\_Locals1, der vom Compiler generierten Klasse für den äußeren Anweisungsblock, enthält die lokale Variable y sowie ein Feld, das auf this des einschließenden Funktionsmembers verweist. Mithilfe dieser Datenstrukturen kann über eine Instanz von \_\_Local2 auf alle erfassten äußeren Variablen zugegriffen werden, und der Code der anonymen Funktion kann somit als Instanzmethode dieser Klasse implementiert werden.

class Test  
{  
 void F() {  
 \_\_Locals1 \_\_locals1 = new \_\_Locals1();  
 \_\_locals1.\_\_this = this;  
 \_\_locals1.y = 123;  
 for (int i = 0; i < 10; i++) {  
 \_\_Locals2 \_\_locals2 = new \_\_Locals2();  
 \_\_locals2.\_\_locals1 = \_\_locals1;  
 \_\_locals2.z = i \* 2;  
 D d = new D(\_\_locals2.\_\_Method1);  
 }  
 }

class \_\_Locals1  
 {  
 public Test \_\_this;  
 public int y;  
 }

class \_\_Locals2  
 {  
 public \_\_Locals1 \_\_locals1;  
 public int z;

public void \_\_Method1() {  
 Console.WriteLine(\_\_locals1.\_\_this.x + \_\_locals1.y + z);  
 }  
 }  
}

Die hier angewendete Technik zum Erfassen der lokalen Variablen kann auch beim Konvertieren anonymer Funktionen in Ausdrucksbaumstrukturen verwendet werden: Verweise auf vom Compiler generierte Objekte können in der Ausdrucksbaumstruktur gespeichert werden, und der Zugriff auf die lokalen Variablen kann als Feld dargestellt werden, das auf diese Objekte zugreift. Der Vorteil dieses Ansatzes liegt darin, dass er das gemeinsame Verwenden „heraufgestufter“ lokaler Variablen durch Delegaten und Ausdrucksbaumstrukturen ermöglicht.

## Konvertierungen von Methodengruppen

Eine implizite Konvertierung (§6.1) aus einer Methodengruppe (§7.1) in einen kompatiblen Delegattyp ist verfügbar. Für einen Delegattyp D und einen Ausdruck E, der als Methodengruppe klassifiziert ist, erfolgt die implizite Konvertierung von E in D, wenn E mindestens eine Methode enthält, die in ihrer normalen Form (§7.5.3.1) auf eine Argumentliste angewendet werden kann, die mithilfe der Parametertypen und Modifizierer von D erstellt wurde, wie im Folgenden beschrieben.

Die Kompilierungsanwendung einer Konvertierung von einer Methodengruppe E in einen Delegattyp D wird im Folgenden beschrieben. Beachten Sie, dass durch das Vorhandensein einer impliziten Konvertierung von E in D nicht sichergestellt ist, dass die Kompilierungsanwendung der Konvertierung ohne Fehler ausgeführt werden kann.

* Eine einzelne Methode M wird gemäß einem Methodenaufruf (§7.6.5.1) der Form E(A) mit den folgenden Modifikationen ausgewählt:
* Die Argumentliste A ist eine Liste von Ausdrücken, die jeweils als Variable klassifiziert sind und über den Typ und den Modifizierer (ref oder out) des entsprechenden Parameters in der formal-parameter-list von D verfügen.
* Als Methoden kommen lediglich die Methoden infrage, die in ihrer normalen Form (§7.5.3.1) angewendet werden können, und nicht diejenigen, die zuvor erweitert werden müssen.
* Wenn der Algorithmus von §7.6.5.1 einen Fehler verursacht, tritt ein Kompilierungsfehler auf. Andernfalls erzeugt der Algorithmus eine einzelne optimale Methode M, die über dieselbe Anzahl von Parametern wie D verfügt, und die Konvertierung gilt als vorhanden.
* Die ausgewählte Methode M muss mit dem Delegattyp D kompatibel (§15.2) sein. Andernfalls tritt ein Kompilierungsfehler auf.
* Wenn es sich bei der ausgewählten Methode M um eine Instanzmethode handelt, bestimmt der E zugeordnete Instanzausdruck das Zielobjekt des Delegaten.
* Wenn es sich bei der ausgewählten Methode M um eine Erweiterungsmethode handelt, die durch einen Memberzugriff auf einen Instanzausdruck gekennzeichnet ist, bestimmt der Instanzausdruck das Zielobjekt des Delegaten.
* Das Ergebnis der Konvertierung ist ein Wert vom Typ D, d. h. ein neu erstellter Delegat, der auf die ausgewählte Methode und das Zielobjekt verweist.

Beachten Sie, dass durch diesen Vorgang ein Delegat für eine Erweiterungsmethode erstellt werden kann, wenn der Algorithmus von §7.6.5.1 keine Instanzmethode finden, aber den Aufruf von E(A) als einen Aufruf einer Erweiterungsmethode (§7.6.5.2) verarbeiten kann. Der aus diesem Grund erstellte Delegat erfasst die Erweiterungsmethode und deren erstes Argument.

Im folgenden Beispiel werden Konvertierungen von Methodengruppen veranschaulicht:

delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test  
{  
 static string F(object o) {...}

static void G() {  
 D1 d1 = F; // Ok  
 D2 d2 = F; // Ok  
 D3 d3 = F; // Error – not applicable  
 D4 d4 = F; // Error – not applicable in normal form  
 D5 d5 = F; // Error – applicable but not compatible

}  
}

Durch die Zuweisung zu d1 wird die Methodengruppe F implizit in einen Wert vom Typ D1 konvertiert.

Die Zuweisung zu d2 beweist, dass das Erstellen eines Delegaten für eine Methode möglich ist, die weniger abgeleitete (kontravariante) Parametertypen und einen mehr abgeleiteten (kovarianten) Rückgabetyp besitzt.

Die Zuweisung zu d3 beweist, dass keine Konvertierung vorhanden ist, wenn die Methode nicht anwendbar ist.

Die Zuweisung zu d4 zeigt, wie die Methode in ihrer normalen Form anwendbar sein sollte.

Die Zuweisung zu d5 veranschaulicht, wie Parameter und Rückgabetypen des Delegaten und der Methoden lediglich für Verweistypen voneinander abweichen können.

Wie bei allen anderen impliziten und expliziten Konvertierungen kann mithilfe des Umwandlungsoperators explizit eine Konvertierung von Methodengruppen ausgeführt werden. Das Beispiel

object obj = new EventHandler(myDialog.OkClick);

kann daher auch wie folgt geschrieben werden:

object obj = (EventHandler)myDialog.OkClick;

Methodengruppen können Einfluss auf die Überladungsauflösung haben und bei Typrückschlüssen zum Einsatz kommen. In §7.5 finden Sie weitere Einzelheiten.

Die Auswertung der Konvertierung einer Methodengruppe zur Laufzeit erfolgt folgendermaßen:

* Wenn es sich bei der zur Kompilierzeit ausgewählten Methode um eine Instanzmethode handelt oder um eine Erweiterungsmethode, auf die als Instanzmethode zugegriffen wird, wird das Zielobjekt des Delegaten über den E zugeordneten Instanzausdruck bestimmt:
* Der Instanzausdruck wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Wenn der Instanzausdruck einem reference-type entspricht, wird der durch den Instanzausdruck berechnete Wert zum Zielobjekt. Wenn es sich bei der ausgewählten Methode um eine Instanzmethode handelt und das Zielobjekt null ist, wird eine System.NullReferenceException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Wenn der Instanzausdruck ein value-type ist, wird ein Boxingvorgang (§4.3.1) ausgeführt, um den Wert in ein Objekt zu konvertieren. Dieses Objekt wird zum Zielobjekt.
* Andernfalls ist die ausgewählte Methode Teil eines statischen Methodenaufrufs, und das Zielobjekt des Delegaten ist gleich null.
* Eine neue Instanz des Delegattyps D wird zugewiesen. Wenn nicht genügend Speicher für die neue Instanz belegt werden kann, wird eine System.OutOfMemoryException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Die neue Delegatinstanz wird mit einem Verweis auf die Methode initialisiert, die während der Kompilierung bestimmt wurde, sowie mit einem Verweis auf das oben berechnete Zielobjekt.

# Ausdrücke

Ein Ausdruck ist eine Folge von Operatoren und Operanden. Dieses Kapitel definiert die Syntax und Bedeutung von Ausdrücken sowie die Rangfolge von Operanden und Operatoren.

## Ausdrucksklassifizierungen

Ein Ausdruck kann als eine der folgenden Kategorien klassifiziert werden:

* Wert. Jedem Wert wird ein Typ zugewiesen.
* Variable. Jeder Variablen wird der deklarierte Variablentyp zugewiesen.
* Namespace. Ein Ausdruck mit dieser Klassifizierung kann nur links in einem member-access (§7.6.4) stehen. In jedem anderen Kontext verursacht ein als Namespace klassifizierter Ausdruck einen Kompilierungsfehler.
* Typ. Ein Ausdruck mit dieser Klassifizierung kann nur links in einem member-access (§7.6.4) oder als ein Operand für den as-Operator (§7.10.11), den is-Operator (§7.10.10) oder den typeof-Operator (§7.6.11) angezeigt werden. In jedem anderen Kontext verursacht ein als Typ klassifizierter Ausdruck einen Kompilierungsfehler.
* Methodengruppe. Hierbei handelt es sich um eine Reihe überladener Methoden, die aus einer Membersuche resultieren (§7.4). Eine Methodengruppe kann über einen zugeordneten Instanzausdruck und eine zugeordnete Typargumentliste verfügen. Beim Aufrufen einer Instanzmethode wird die Instanz als Ergebnis der Auswertung des Instanzausdrucks durch this (§7.6.7) dargestellt. Methodengruppen sind in einer invocation-expression (§7.6.5), einer delegate-creation-expression (§7.6.10.5) und auf der linken Seite eines is-Operators zulässig und können implizit in einen kompatiblen Delegattyp (§6.6) konvertiert werden. In jedem anderen Kontext verursacht ein als Methodengruppe klassifizierter Ausdruck einen Kompilierungsfehler.
* Ein NULL-Literal. Ein Ausdruck mit dieser Klassifizierung kann implizit in einen Verweistyp oder einen Typ, der NULL-Werte zulässt, konvertiert werden.
* Eine anonyme Funktion. Ein Ausdruck mit dieser Klassifizierung kann implizit in einen kompatiblen Delegattyp oder einen Ausdrucksbaumstrukturtyp konvertiert werden.
* Eigenschaftenzugriff. Jedem Eigenschaftenzugriff wird der Eigenschaftentyp zugewiesen. Weiterhin wird einem Eigenschaftenzugriff ggf. ein Instanzausdruck zugeordnet. Wenn ein Accessor (der get-Block oder der set-Block) eines Instanzeigenschaftenzugriffs aufgerufen wird, wird die Instanz als Ergebnis der Auswertung des Instanzausdrucks durch this (§7.6.7) dargestellt.
* Ereigniszugriff. Jedem Ereigniszugriff wird der Ereignistyp zugewiesen. Weiterhin wird einem Ereigniszugriff ggf. ein Instanzausdruck zugeordnet. Ein Ereigniszugriff kann als linker Operand des +=-Operators bzw. des -=-Operators (§7.17.3) auftreten. In jedem anderen Kontext verursacht ein als Ereigniszugriff klassifizierter Ausdruck einen Kompilierungsfehler.
* Indexerzugriff. Jedem Indexerzugriff wird der Elementtyp des Indexers zugewiesen. Außerdem verfügt ein Indexerzugriff über einen zugeordneten Instanzausdruck und eine zugeordnete Argumentliste. Wenn ein Accessor (der get-Block oder der set-Block) eines Indexzugriffs aufgerufen wird, wird die Instanz als Ergebnis der Bewertung des Instanzausdrucks durch this (§7.6.7) dargestellt, und das Ergebnis der Auswertung der Argumentenliste wird zur Parameterliste für den Aufruf.
* Nothing. Ergebnis, wenn der Ausdruck als Aufruf einer Methode mit dem Rückgabetyp void dient. Ein als Nothing klassifizierter Ausdruck ist nur im Kontext einer statement-expression (§8.6) gültig.

Das Endergebnis eines Ausdrucks kann niemals ein Namespace, ein Typ, eine Methodengruppe oder ein Ereigniszugriff sein. Wie oben beschrieben, sind diese Ausdruckskategorien Zwischenergebnisse, die nur in bestimmten Kontexten zulässig sind.

Ein Eigenschaften- oder Indexerzugriff wird als Wert stets durch Aufrufen des get-accessor oder des set-accessor neu klassifiziert. Der jeweilige Accessor wird durch den Kontext des Eigenschaften- oder Indexerzugriffs festgelegt: Wenn der Zugriff das Zuweisungsziel ist, wird der set-accessor aufgerufen und ein neuer Wert (§7.17.1) zugewiesen. Andernfalls wird der get-accessor aufgerufen und der aktuelle Wert (§7.1.1) abgerufen.

### Werte von Ausdrücken

Die meisten Konstrukte mit einem Ausdruck erfordern, dass der Ausdruck einen Wert ergibt. Wenn in solchen Fällen der tatsächliche Ausdruck einen Namespace, einen Typ, eine Methodengruppe oder keinen Wert ausgibt, tritt ein Kompilierungsfehler auf. Wenn der Ausdruck jedoch einen Eigenschaftenzugriff, einen Indexerzugriff oder eine Variable ausgibt, wird der Wert der Eigenschaft, des Indexers oder der Variablen implizit ersetzt:

* Der Wert der Variablen entspricht dem Wert, der in dem durch die Variable festgelegten Speicherplatz aktuell gespeichert ist. Eine Variable muss definitiv zugewiesen sein (§5.3), bevor deren Wert abgerufen werden kann. Andernfalls tritt während der Kompilierung ein Fehler auf.
* Der Wert des Ausdrucks eines Eigenschaftenzugriffs wird durch Aufrufen des get-accessor der Eigenschaft abgerufen. Wenn die Eigenschaft keinen get-accessor aufweist, tritt ein Kompilierungsfehler auf. Andernfalls wird ein Funktionsmember (§7.5.4) aufgerufen und als Ergebnis des Aufrufs der Wert des Eigenschaftenzugriffsausdrucks zurückgegeben.
* Der Wert eines Indexerzugriffsausdrucks wird durch Aufrufen des get-accessor des Indexers abgerufen. Wenn der Indexer keinen get-accessor aufweist, tritt ein Kompilierungsfehler auf. Andernfalls wird ein Funktionsmember (§7.5.4) aufgerufen, wobei die Argumentenliste mit dem Indexerzugriffsausdruck verknüpft ist und als Ergebnis des Aufrufs der Wert des Indexerzugriffsausdrucks zurückgegeben wird.

## Statische und dynamische Bindung

Die Vorgehensweise zum Bestimmen der Bedeutung einer Operation basierend auf dem Typ oder Wert enthaltener Ausdrücke (Argumente, Operanden, Empfänger) wird häufig als Binden bezeichnet. Die Bedeutung eines Methodenaufrufs wird beispielsweise basierend auf dem Typ des Empfängers und der Argumente bestimmt. Die Bedeutung eines Operators wird basierend auf dem Typ seiner Operanden bestimmt.

In C# wird die Bedeutung einer Operation in der Regel während der Kompilierung basierend auf dem Kompilierungstyp der enthaltenen Ausdrücke bestimmt. Entsprechend wird, wenn ein Ausdruck einen Fehler enthält, dieser vom Compiler ermittelt und gemeldet. Dieser Ansatz wird als statische Bindung bezeichnet.

Wenn es sich bei einem Ausdruck jedoch um einen dynamischen Ausdruck handelt (dieser also den Typ dynamic aufweist), bedeutet dies, dass eine Bindung, an der er beteiligt ist, auf seinem Laufzeittyp (d. h. dem tatsächlichen Typ des Objekts, das der Ausdruck zur Laufzeit kennzeichnet) basieren sollte und nicht auf dem Typ bei der Kompilierung. Die Bindung einer solchen Operation wird daher bis zu dem Zeitpunkt verschoben, zu dem die Operation während der Ausführung des Programms ausgeführt werden soll. Dies wird als dynamische Bindung bezeichnet.

Wenn eine Operation dynamisch gebunden wird, wird vom Compiler nur eine schnelle bzw. keine Prüfung durchgeführt. Stattdessen werden Fehler bei der Laufzeitbindung als Ausnahmen zur Laufzeit gemeldet.

Die folgenden Operationen in C# unterliegen der Bindung:

* Memberzugriff: e.M
* Aufrufmethode: e.M(e1,…,en)
* Delegataufruf: e(e1,…,en)
* Elementzugriff: e[e1,…,en]
* Erstellen von Objekten: new C(e1,…,en)
* Überladene unäre Operatoren: +, -, !, ~, ++, --, true, false
* Überladene binäre Operatoren: +, -, \*, /, %, &, &&, |, ||, ??, ^, <<, >>, ==,!=, >, <, >=, <=
* Zuweisungsoperatoren: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
* Implizite und explizite Konvertierungen

Wenn keine dynamischen Ausdrücke vorhanden sind, wird in C# standardmäßig die statische Bindung verwendet. Dies bedeutet, dass die Kompilierungstypen von enthaltenen Ausdrücken im Auswahlprozess verwendet werden. Wenn jedoch einer der enthaltenen Ausdrücke in den oben aufgeführten Operationen ein dynamischer Ausdruck ist, wird die Operation stattdessen dynamisch gebunden.

### Bindungszeit

Die statische Bindung findet während der Kompilierung statt, die dynamische Bindung hingegen zur Laufzeit. In den folgenden Abschnitten bezieht sich der Begriff Bindungszeit entweder auf die Kompilierung oder die Laufzeit, je nachdem, wann die Bindung stattfindet.

Im folgenden Beispiel werden die statische und die dynamische Bindung sowie die Bindungszeit erläutert.

object o = 5;  
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)  
Console.WriteLine(o); // static binding to Console.WriteLine(object)  
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)

Die ersten beiden Aufrufe werden statisch gebunden: Die Überladung von Console.WriteLine wird basierend auf dem Kompilierungstyp ihrer Argumente ermittelt. Die Bindungszeit bezieht sich also auf die Kompilierung.

Der dritte Aufruf wird dynamisch gebunden: Die Überladung von Console.WriteLine wird basierend auf dem Laufzeittyp seiner Argumente ermittelt. Dies liegt daran, dass das Argument ein dynamischer Ausdruck ist – sein Kompilierungstyp ist dynamic. Die Bindungszeit für den dritten Aufruf bezieht sich also auf die Laufzeit.

### Dynamische Bindung

Sinn und Zweck der dynamischen Bindung besteht darin, C#-Programmen die Interaktion mit dynamischen Objekten, d. h. Objekten, die nicht den normalen Regeln des C#-Typsystems folgen, zu ermöglichen. Dynamische Objekte können Objekte aus anderen Programmiersprachen mit unterschiedlichen Typsystemen sein oder Objekte, die programmgesteuert so eingerichtet sind, dass sie ihre eigene Bindungssemantik für unterschiedlichen Operationen implementieren.

Der Mechanismus, anhand dem ein dynamisches Objekt seine eigene Semantik implementiert, wird durch die Implementierung definiert. Eine bestimmte Schnittstelle – die ebenfalls durch die Implementierung definiert wird – wird von dynamischen Objekten implementiert, um der C#-Laufzeit zu signalisieren, dass sie eine spezielle Semantik aufweist. Immer dann, wenn Operationen in einem dynamischen Objekt dynamisch gebunden werden, wird daher ihre eigene Bindungssemantik und nicht die Semantik in C# verwendet, wie in diesem Dokument beschrieben.

Während der Sinn und Zweck der dynamischen Bindung darin besteht, die Interaktion mit dynamischen Objekten zu ermöglichen, ermöglicht C# die dynamische Bindung aller Objekte, unabhängig davon, ob diese dynamisch sind oder nicht. Auf diese Weise wird eine nahtlosere Integration dynamischer Objekte ermöglicht, da es sich bei den Ergebnissen von Operationen für diese Objekte möglicherweise nicht um dynamische Objekte handelt, diese jedoch einen dem Programmierer während der Kompilierung unbekannten Typ aufweisen. Durch die dynamische Bindung kann auch fehleranfälliger reflektionsbasierter Code beseitigt werden, selbst wenn es sich bei keinem der Objekte um dynamische Objekte handelt.

In den folgenden Abschnitten wird für jedes Konstrukt in der Sprache genau beschrieben, wann die dynamische Bindung angewendet wird, welche Kompilierzeitprüfung ggf. angewendet wird und worum es sich bei dem Kompilierzeitergebnis und der Ausdrucksklassifizierung handelt.

### Typen von enthaltenden Ausdrücken

Wenn ein Vorgang statisch gebunden ist, wird der Typ eines enthaltenden Ausdrucks (z. B. ein Empfänger, ein Argument, ein Index oder ein Operand) immer als der Kompilierungstyp dieses Ausdrucks betrachtet.

Ist ein Vorgang dynamisch gebunden, wird der Typ eines enthaltenden Ausdrucks auf unterschiedliche Art und Weise bestimmt, abhängig vom Kompilierungstyp des enthaltenden Ausdrucks:

* Der Typ eines enthaltenden Ausdrucks vom Kompilierungstyp dynamic wird als der Typ des tatsächlichen Werts betrachtet, den der Ausdruck zur Laufzeit auswertet.
* Bei einem enthaltenden Ausdruck, dessen Kompilierungstyp ein Typparameter ist, wird davon ausgegangen, dass er den Typ aufweist, an den der Typparameter zur Laufzeit gebunden ist.
* Andernfalls wird davon ausgegangen, dass der enthaltende Ausdruck seinen Kompilierungstyp aufweist.

## Operatoren

Ausdrücke werden aus Operanden und Operatoren gebildet. Die Operatoren eines Ausdrucks geben an, welche Operationen auf die Operanden angewendet werden. Beispiele für Operatoren sind +, -, \*, / und new. Beispiele für Operanden sind Literale, Felder, lokale Variablen und Ausdrücke.

Es gibt drei Arten von Operatoren:

* Unäre Operatoren Unäre Operatoren verwenden einen Operanden und entweder Präfix-Notation (z. B. –x) oder Postfix-Notation (z. B. x++).
* Binäre Operatoren Binäre Operatoren verwenden stets zwei Operanden und Infix-Notation (z. B. x + y).
* Ternäre Operatoren. Nur ein ternärer Operator, nämlich ?:, ist verfügbar. Dieser Operator verwendet drei Operanden und Infix-Notation (c? x: y).

Die Reihenfolge der Auswertung der Operatoren in einem Ausdruck wird durch Vorrang und Orientierung der Operatoren (§7.3.1) festgelegt.

Operanden werden in einem Ausdruck von links nach rechts ausgewertet. In dem Ausdruck F(i) + G(i++) \* H(i) wird beispielsweise die Methode F mit dem alten Wert von i, dann die Methode G mit dem alten Wert von i und zum Schluss die Methode H mit dem neuen Wert von i aufgerufen. Dies erfolgt unabhängig vom Vorrang der Operatoren.

Bestimmte Operatoren können überladen werden. Durch das Überladen von Operatoren können benutzerdefinierte Operatoren für Operationen implementiert werden, bei denen mindestens einer der beiden Operanden einem benutzerdefinierten Klassen- oder Strukturtyp (§7.3.2) entspricht.

### Operatorvorrang und -orientierung

Wenn ein Ausdruck mehrere Operatoren enthält, wird anhand des Vorrangs der Operatoren festgelegt, in welcher Reihenfolge die einzelnen Operatoren ausgewertet werden. Der Ausdruck x + y \* z wird beispielsweise als x + (y \* z) ausgewertet, da der \*-Operator vor dem binären +-Operator Vorrang hat. Der Vorrang eines Operators wird durch die Definition in der zugehörigen Syntaxbildung festgelegt. Eine additive-expression besteht z. B. aus einer Reihenfolge von multiplicative-expressions, die durch den +-Operator oder den --Operator voneinander getrennt sind, wobei der +-Operator und der --Operator einen niedrigeren Stellenwert erhalten als der \*-Operator, der /-Operator und der %-Operator.

Die folgende Tabelle fasst die Operatoren in der Reihenfolge von der höchsten zur niedrigsten Rangfolge zusammen:

|  |  |  |
| --- | --- | --- |
| **Abschnitt** | **Kategorie** | **Operatoren** |
| 7.6 | Primär | x.y f(x) a[x] x++ x-- new  typeof default checked unchecked delegate |
| 7.7 | Unär | + - ! ~ ++x --x (T)x |
| 7.8 | Multiplikativ | \* / % |
| 7.8 | Additiv | + - |
| 7.9 | Verschiebung | << >> |
| 7.10 | Relational und Typtest | < > <= >= is as |
| 7.10 | Gleichheit | == != |
| 7.11 | Logisches AND | & |
| 7.11 | Logisches XOR | ^ |
| 7.11 | Logisches OR | | |
| 7.12 | Bedingtes AND | && |
| 7.12 | Bedingtes OR | || |
| 7.13 | Null Coalesce | ?? |
| 7.14 | Bedingung | ?: |
| 7.17, 7.15 | Zuweisungs- und Lambda-Ausdruck | = \*= /= %= += -= <<= >>= &= ^= |=  => |

Wenn sich ein Operand zwischen zwei Operatoren gleichen Ranges befindet, entscheidet die Orientierung des Operators, in welcher Reihenfolge die Operationen ausgeführt werden:

* Mit Ausnahme der Zuweisungsoperatoren und des NULL-Sammeloperators sind alle binären Operatoren linksassoziativ, d. h., die Operationen werden von links nach rechts ausgeführt. x + y + z wird beispielsweise als (x + y) + z ausgewertet.
* Die Zuweisungsoperatoren, der NULL-Sammeloperator und der bedingte Operator (?:) sind rechtsassoziativ, d. h., die Operationen werden von rechts nach links ausgeführt. x = y = z wird beispielsweise als x = (y = z) ausgewertet.

Vorrang und Orientierung können durch die Verwendung von Klammern gesteuert werden. Beispielsweise wird beim Ausdruck x + y \* z zuerst y mit z multipliziert und anschließend x zum Ergebnis addiert. Bei (x + y) \* z wird zuerst x und y addiert und dann das Ergebnis mit z multipliziert.

### Überladen von Operatoren

Alle unären und binären Operatoren verfügen über vordefinierte Implementierungen, die in allen Ausdrücken automatisch verfügbar sind. Zusätzlich zu den vordefinierten Implementierungen können durch Einfügen von operator-Deklarationen in Klassen und Strukturen (§10.10) benutzerdefinierte Implementierungen eingeführt werden. Benutzerdefinierte Operatorimplementierungen haben stets Vorrang gegenüber vordefinierten Operatorimplementierungen. Nur wenn keine benutzerdefinierte Operatorimplementierung vorhanden ist, wird die vordefinierte Operatorimplementierung verwendet, wie in §7.3.3 und §7.3.4 beschrieben.

Folgende unäre Operatoren können überladen werden:

+ - ! ~ ++ -- true false

Obwohl true und false in Ausdrücken nicht explizit verwendet werden (und daher in der Tabelle zum Vorrang in §7.3.1 nicht aufgeführt ist), gelten sie als Operatoren, weil sie im Kontext verschiedener Ausdrücke verwendet werden: boolesche Ausdrücke (§7.20) und Ausdrücke mit Bedingungsoperatoren (§7.14) und logischen Bedingungsoperatoren (§7.12).

Folgende binäre Operatoren können überladen werden:

+ - \* / % & | ^ << >> == != > < >= <=

Prinzipiell können nur die oben aufgeführten Operatoren überladen werden. Memberzugriffe, Methodenaufrufe oder die Operatoren =, &&, ||, ??, ?:, =>, checked, unchecked, new, typeof, default, as und is können nicht überladen werden.

Beim Überladen eines binären Operators wird implizit auch der zugehörige Zuweisungsoperator, sofern vorhanden, überladen. Eine Überladung des Operators \* ist gleichbedeutend mit einer Überladung des Operators \*=. Dies wird in §7.17.2 näher erläutert. Beachten Sie, dass der Zuweisungsoperator selbst (=) nicht überladen werden kann. Bei einer Zuweisung wird stets ein Wert bitweise in eine Variable kopiert.

Konvertierungsoperationen wie (T)x werden durch benutzerdefinierte Konvertierungen überladen (§6.4).

Elementzugriff wie a[x] gilt nicht als ein überladbarer Operator. Benutzerdefinierte Indizierung wird stattdessen durch Indexer (§10.9) unterstützt.

In Ausdrücken werden Operatoren anhand der Operatornotation, in Deklarationen hingegen anhand funktionaler Notation dargestellt. Die folgende Tabelle verdeutlicht die Beziehung zwischen Operatornotation und funktionalen Notationen für unäre und binäre Operatoren. Im ersten Eintrag steht op für einen beliebigen überladbaren, unären Präfixoperator. Im zweiten Eintrag bezeichnet op die unären Postfixoperatoren ++ und --. Im dritten Eintrag steht op für einen beliebigen überladbaren, binären Operator.

|  |  |
| --- | --- |
| **Operatornotation** | **Funktionale Notation** |
| op x | operator op(x) |
| x op | operator op(x) |
| x op y | operator op(x, y) |

Bei benutzerdefinierten Operatordeklarationen muss stets mindestens einer der Parameter den Klassen- oder Strukturtyp aufweisen, der die Operatordeklaration enthält. Daher kann für einen benutzerdefinierten Operator nicht dieselbe Signatur wie für einen vordefinierten Operator festgelegt sein.

Mit benutzerdefinierte Operatordeklarationen können weder die Syntax noch der Vorrang oder die Orientierung eines Operators geändert werden. Beispielsweise ist der /-Operator stets ein binärer Operator, hat immer den in §7.3.1 definierten Vorrang und ist immer linksassoziativ.

Während ein benutzerdefinierter Operator jede gewünschte Berechnung ausführen kann, wird von Implementierungen, die einen vom intuitiv erwarteten Ergebnis abweichenden Wert erbringen, strikt abgeraten. Beispielsweise soll die Implementierung von operator operator == die beiden Operanden auf deren Gleichheit überprüfen und das entsprechende Ergebnis vom Typ bool zurückgeben.

Die in §7.6 bis §7.12 enthaltenen Beschreibungen einzelner Operatoren definieren die vordefinierten Implementierungen von Operatoren sowie alle zusätzlichen Regeln, die auf die einzelnen Operatoren zutreffen. In den Beschreibungen werden die Begriffe Überladungsaufösung für unäre Operatoren, Überladungsauflösung für binäre Operatoren und numerische Erweiterung verwendet. Die Definitionen dieser Begriffe finden Sie in den folgenden Abschnitten.

### Überladungsauflösung unärer Operatoren

Eine Operation der Form op x oder x op, wobei op einem überladbaren unären Operator entspricht und x ein Ausdruck des Typs X ist, wird wie folgt verarbeitet:

* Die durch X vorgegebene Gruppe potenzieller benutzerdefinierter Operatoren für die Operation operator op(x) wird anhand der Regeln in §7.3.5 festgelegt.
* Wenn die Gruppe potenzieller benutzerdefinierter Operatoren nicht leer ist, wird sie zur Gruppe der potenziellen Operatoren für die Operation. Andernfalls werden die vordefinierten unären Implementierungen, einschließlich ihrer heraufgestuften Formen, von operator op zur Gruppe der potenziellen Operatoren für die Operation. Die vordefinierten Implementierungen eines bestimmten Operators werden in der Beschreibung des Operators (§7.6 und §7.7) festgelegt.
* Die Regeln zur Überladungsauflösung in §7.5.3 werden auf die Gruppe der potenziellen Operatoren angewendet, um den hinsichtlich der Argumentenliste (x) am besten geeigneten Operator auszuwählen. Dieser Operator wird als Ergebnis des Überladungsauflösungsprozesses verwendet. Wenn die Überladungsauflösung keinen am besten geeigneten Operator auswählt, wird ein Bindungsfehler angezeigt.

### Überladungsauflösung binärer Operatoren

Eine Operation der Form x op y, wobei op einem überladbaren binären Operator entspricht, x ein Ausdruck des Typs X und y ein Ausdruck des Typs Y ist, wird wie folgt verarbeitet:

* Die durch X und Y vorgegebene Gruppe potenzieller benutzerdefinierter Operatoren für die Operation operator op(x, y) wird festgelegt. Die Gruppe setzt sich aus den durch X vorgegebenen potenziellen Operatoren und den durch Y vorgegebenen potenziellen Operatoren zusammen, wobei beide Gruppen anhand der Regeln in §7.3.5 festgelegt sind. Wenn X und Y vom selben Typ sind, oder wenn X und Y aus einem gemeinsamen Basistyp abgeleitet wurden, kommen gemeinsam genutzte potenzielle Operatoren nur in der kombinierten Gruppe vor.
* Wenn die Gruppe potenzieller benutzerdefinierter Operatoren nicht leer ist, wird sie zur Gruppe der potenziellen Operatoren für die Operation. Andernfalls werden die vordefinierten binären Implementierungen, einschließlich ihrer heraufgestuften Formen, von operator op zur Gruppe der potenziellen Operatoren für die Operation. Die vordefinierten Implementierungen eines bestimmten Operators werden in der Beschreibung des Operators (§7.8 bis §7.12) festgelegt. Für vordefinierte Enumerations- und Delegatoperatoren sind die einzigen berücksichtigten Operatoren die von einem Enumerations- oder Delegattyp definierten, bei dem es sich um den Bindungstyp von einem der Operanden handelt.
* Die Regeln zur Überladungsauflösung in §7.5.3 werden auf die Gruppe der potenziellen Operatoren angewendet, um den hinsichtlich der Argumentenliste (x, y) am besten geeigneten Operator auszuwählen. Dieser Operator wird als Ergebnis des Überladungsauflösungsprozesses verwendet. Wenn die Überladungsauflösung keinen am besten geeigneten Operator auswählt, wird ein Bindungsfehler angezeigt.

### Benutzerdefinierte Kandidatenoperatoren

Wenn ein Typ T und eine operator op(A) gegeben ist, wobei op ein überladbarer Operator und A eine Argumentenliste ist, wird die durch T vorgegebene Gruppe potenzieller benutzerdefinierter Operatoren für operator op(A) wie folgt definiert:

* Bestimmen des Typs T0. Wenn T ein Typ ist, der NULL-Werte zulässt, ist T0 der zugrunde liegende Typ, andernfalls ist T0 gleich T.
* Für alle operator op-Deklarationen in T0 und alle heraufgestuften Formen dieser Operatoren gilt: Wenn mindestens ein Operator (§7.5.3.1) hinsichtlich der Argumentenliste A zutreffend ist, dann enthält die Gruppe der potenziellen Operatoren alle zutreffenden Operatoren in T0.
* Andernfalls gilt: Wenn T0 dem object entspricht, dann ist die Gruppe der potenziellen Operatoren leer.
* Andernfalls entspricht die durch T0 vorgegebene Gruppe der potenziellen Operatoren der durch die direkte Basisklasse von T0 oder die effektive Basisklasse von T0, wenn T0 ein Typparameter ist, vorgegebenen Gruppe der potenziellen Operatoren.

### Numerische Erweiterungen

Eine numerische Erweiterung besteht aus der automatischen Ausführung bestimmter impliziter Umwandlungen von Operanden zu den vordefinierten unären und binären numerischen Operatoren. Dabei ist die numerische Erweiterung kein spezieller Mechanismus, sondern ergibt sich aus der Anwendung von Überladungsauflösungen auf vordefinierte Operatoren. Numerische Erweiterungen wirken sich nicht auf die Auswertung von benutzerdefinierten Operatoren aus, obwohl sich durch Implementierung benutzerdefinierter Operatoren vergleichbare Auswirkungen ergeben.

Ein Beispiel für eine numerische Erweiterung ist die vordefinierte Implementierung des binären \*-Operators:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);  
float operator \*(float x, float y);  
double operator \*(double x, double y);  
decimal operator \*(decimal x, decimal y);

Wenn die Regeln zur Überladungsauflösung (§7.5.3) auf diese Operatoren angewendet werden, wird der erste Operator ausgewählt, für den implizite Umwandlungen des Operandentyps bestehen. Für die Operation b \* s, bei der b für byte und s für short steht, wählt die Überladungsauflösung als besten Operator operator \*(int, int) aus. Dadurch werden b und s zu int konvertiert, und der Ergebnistyp ist int. Analog dazu wird für die Operation i \* d, bei der i für int und d für double steht, als bester Operator operator \*(double, double) ausgewählt.

#### Unäre numerische Erweiterungen

Unäre numerische Erweiterungen werden für die Operanden des vordefinierten unären +-Operators, –-Operators und ~-Operators durchgeführt. Eine unäre numerische Erweiterung ist lediglich eine Umwandlung von Operanden des Typs sbyte, byte, short, ushort oder char in den Typ int. Beim unären –-Operator wandelt die unäre numerische Erweiterung zusätzlich Operanden vom Typ uint in den Typ long um.

#### Binäre numerische Erweiterungen

Binäre numerische Erweiterungen werden für Operanden zu den vordefinierten binären Operatoren +, –, \*, /, %, &, |, ^, ==, !=, >, <, >= und <= durchgeführt. Eine binäre numerische Erweiterung wandelt beide Operanden implizit in einen gemeinsamen Typ um, der im Fall nicht relationaler Operatoren gleichzeitig Ergebnistyp der Operation ist. Bei einer binären numerischen Erweiterung werden die folgenden Regeln in der Reihenfolge ihrer Auflistung angewendet:

* Wenn einer der Operanden vom Typ decimal ist, wird der andere Operand in den Typ decimal umgewandelt. Wenn der andere Operand vom Typ float oder double ist, wird ein Bindungsfehler angezeigt.
* Wenn einer der Operanden vom Typ double ist, wird auch der andere Operand in den Typ double umgewandelt.
* Wenn einer der Operanden vom Typ float ist, wird auch der andere Operand in den Typ float umgewandelt.
* Wenn einer der Operanden vom Typ ulong ist, wird auch der andere Operand in den Typ ulong umgewandelt. Wenn der andere Operand vom Typ sbyte, short, int oder long ist, wird ein Bindungsfehler angezeigt.
* Wenn einer der Operanden vom Typ long ist, wird auch der andere Operand in den Typ long umgewandelt.
* Wenn einer der Operanden vom Typ uint ist und der andere Operand vom Typ sbyte, short oder int, werden beide Operanden in den Typ long umgewandelt.
* Wenn einer der Operanden vom Typ uint ist, wird auch der andere Operand in den Typ uint umgewandelt.
* Andernfalls werden beide Operanden in den Typ int umgewandelt.

Beachten Sie, dass mit der ersten Regel alle Operationen unterdrückt werden, die den decimal-Typ mit den Typen double und float kombinieren. Die Regel ergibt sich aus der Tatsache, dass zwischen dem decimal-Typ und den Typen double und float keine impliziten Umwandlungen zulässig sind.

Darüber hinaus sind keine Operanden vom Typ ulong zulässig, wenn es sich bei dem anderen Operanden um einen vorzeichenbehafteten Ganzzahltyp handelt. Der Grund dafür besteht darin, dass ein ganzzahliger Typ weder den vollständigen Bereich von ulong noch den von vorzeichenbehafteten Ganzzahltypen darstellen kann.

In beiden der oben dargestellten Fälle kann ein Operand mithilfe eines Typumwandlungsausdrucks explizit in einen Typ umgewandelt werden, der mit dem Typ des anderen Operanden kompatibel ist.

In dem Beispiel

decimal AddPercent(decimal x, double percent) {  
 return x \* (1.0 + percent / 100.0);  
}

tritt ein Fehler während der Bindung auf, da der Typ decimal nicht mit dem Typ double multipliziert werden kann. Der Fehler wird behoben, indem der zweite Operand explizit in den Typ decimal umgewandelt wird:

decimal AddPercent(decimal x, double percent) {  
 return x \* (decimal)(1.0 + percent / 100.0);  
}

### Heraufgestufte Operatoren

Mit heraufgestuften Operatoren können vordefinierte und benutzerdefinierte Operatoren, die Werttypen verarbeiten, die keine NULL-Werte zulassen, auch für Formen dieser Typen, die NULL-Werte zulassen, verwendet werden. Heraufgestufte Operatoren werden aus vordefinierten und benutzerdefinierten Operatoren konstruiert, die bestimmten Anforderungen entsprechen, die im Folgenden beschrieben werden:

* Bei den unären Operatoren

+ ++ - -- ! ~

ist eine heraufgestufte Form eines Operators vorhanden, wenn der Typ von Operand und Ergebnis jeweils keine Werttypen sind, die NULL-Werte zulassen. Die heraufgestufte Form wird durch Hinzufügen eines einzelnen ?-Modifizierers an den Operand- und den Ergebnistyp konstruiert. Der heraufgestufte Operator erstellt einen NULL-Wert, wenn der Operand NULL ist. Andernfalls entwrappt der Operator den Operanden, wendet den zugrund liegenden Operator an und wrappt dann das Ergebnis.

* Bei den binären Operatoren

+ - \* / % & | ^ << >>

ist eine heraufgestufte Form eine Operators vorhanden, wenn die Typen von Operanden und Ergebnis jeweils keine Werttypen sind, die NULL-Werte zulassen. Die heraufgestufte Form wird durch Hinzufügen eines einzelnen ?-Modifizierers an jeden Operandentyp und den Ergebnistyp konstruiert. Der heraufgestufte Operator erstellt einen NULL-Wert, wenn mindestens einer der Operanden NULL ist (eine Ausnahme bilden die Operatoren & und | für den bool?-Typ, wie in §7.11.3) beschrieben). Andernfalls entwrappt der Operator die Operanden, wendet den zugrund liegenden Operator an und wrappt dann das Ergebnis.

* Bei den Gleichheitsoperatoren

== !=

ist eine heraufgestufte Form des Operators vorhanden, wenn die Operandentypen jeweils keine Werttypen sind, die NULL-Werte zulassen, und der Ergebnistyp bool ist. Die heraufgestufte Form wird durch Hinzufügen eines einzelnen ?-Modifizierers an jeden Operandentyp konstruiert. Der heraufgestufte Operator nimmt zwei NULL-Werte als gleich und einen NULL-Wert verglichen mit einem Nicht-NULL-Wert als ungleich an. Wenn beide Operanden nicht NULL sind, entwrappt der heraufgestufte Operator die Operanden und wendet den zugrunde liegenden Operator an, um das Ergebnis vom Typ bool zu erstellen.

* Bei den relationalen Operatoren

< > <= >=

ist eine heraufgestufte Form des Operators vorhanden, wenn die Operandentypen jeweils keine Werttypen sind, die NULL-Werte zulassen, und der Ergebnistyp bool ist. Die heraufgestufte Form wird durch Hinzufügen eines einzelnen ?-Modifizierers an jeden Operandentyp konstruiert. Der heraufgestufte Operator generiert den Wert false, wenn mindestens ein Operand NULL ist. Andernfalls entwrappt der heraufgestufte Operator die Operanden und wendet den zugrunde liegenden Operator an, um das Ergebnis vom Typ bool zu erzeugen.

## Membersuche

Die Membersuche ist der Vorgang, bei dem die Bedeutung eines Namens im Kontext eines Typs bestimmt wird. Membersuchvorgänge können als Teil der Auswertung eines simple-name (§7.6.2) oder member-access (§7.6.4) in einem Ausdruck auftreten. Wenn der simple-name oder member-access als primary-expression einer invocation-expression (§7.6.5.1) auftritt, wird der Member als aufgerufen bezeichnet.

Wenn ein Member eine Methode oder ein Ereignis ist bzw. eine Konstante, ein Feld oder eine Eigenschaft entweder eines Delegattyps (§15) oder des Typs dynamic (§4.7), wird der Member als aufrufbar bezeichnet.

Bei Membersuchvorgängen wird nicht nur der Name eines Members, sondern auch die Anzahl von Typparametern berücksichtigt, die ein Member besitzt, und ob der Member zugreifbar ist. Im Rahmen der Membersuche besitzen generische Methoden und geschachtelte generische Typen die Anzahl von Typparametern, die in deren entsprechenden Deklarationen angeben ist, und alle anderen Member besitzen 0 (null) Typparameter.

Die Membersuche für einen Namen N mit K Typparametern in einem Typ T wird wie folgt verarbeitet:

* Zuerst wird die Menge der zugreifbaren Member mit dem Namen N bestimmt:
* Wenn T ein Typparameter ist, dann ist die Menge die Gesamtmenge der zugreifbaren Member mit dem Namen N in jedem als primäre oder sekundäre Einschränkung (§10.1.5) für Tangegebenen Typ, zuzüglich der Menge der zugreifbaren Member mit dem Namen N in object.
* Andernfalls besteht die Menge aus allen zugreifbaren (§3.5) Membern mit dem Namen N in T, einschließlich vererbter Member und der zugreifbaren Member mit dem Namen N in object. Wenn T ein konstruierter Typ ist, wird die Menge der Member durch Ersetzen der Typargumente gemäß der Beschreibung in §10.3.2 abgerufen. Member mit override-Modifizierer werden aus der Menge ausgeschlossen.
* Wenn als Nächstes K gleich 0 (null) ist, werden alle geschachtelten Typen entfernt, deren Deklarationen Typparameter enthalten. Wenn K nicht 0 (null) ist, werden alle Member mit abweichender Anzahl von Typparametern entfernt. Beachten Sie, dass keine Methoden mit Typparametern entfernt werden, wenn K 0 (null) ist, da der Typrückschluss (§7.5.2) möglicherweise auf die Typargumente schließen kann.
* Als Nächstes werden, wenn der Member aufgerufen wird, alle nicht aufrufbaren Member aus der Menge entfernt.
* Anschließend werden Member, die durch andere Member verdeckt werden, aus der Gruppe entfernt. Für alle S.M-Member in der Menge (wobei S der Typ ist, in dem der Member M deklariert ist) gelten die folgenden Regeln:
* Wenn M eine Konstante, ein Feld, eine Eigenschaft, ein Ereignis oder ein Enumerationsmember ist, werden alle in einem Basistyp von S deklarierten Member aus der Menge entfernt.
* Wenn M eine Typdeklaration ist, werden alle Member, die keine Typen sind und in einer Basisklasse von S deklariert sind, aus der Menge entfernt, und alle Typdeklarationen mit der gleichen Anzahl von Typparametern wie M, die in einem Basistyp von S deklariert sind, werden aus der Menge entfernt.
* Wenn M eine Methode ist, werden alle Member, die keine Methoden sind und in einem Basistyp von S deklariert sind, aus der Menge entfernt.
* Als Nächstes werden Schnittstellenmember, die durch Klassenmember verdeckt werden, aus der Menge entfernt. Dieser Schritt hat nur dann eine Auswirkung, wenn T ein Typparameter ist und T sowohl eine effektive Basisklasse, die nicht object ist, als auch eine nicht leere effektive Schnittstellengruppe (§10.1.5) besitzt. Für alle S.M-Member in der Menge (wobei S der Typ ist, in dem der Member M deklariert ist) gelten die folgenden Regeln, wenn S eine Klassendeklaration ist, die nicht object ist:
* Wenn M eine Konstante, ein Feld, eine Eigenschaft, ein Ereignis, ein Enumerationsmember oder eine Typdeklaration ist, werden alle in einer Schnittstellendeklaration deklarierten Member aus der Menge entfernt.
* Wenn M eine Methode ist, werden alle Member, die keine Methoden sind und in einer Schnittstellendeklaration deklariert sind, aus der Menge entfernt, und alle Methoden mit der gleichen Signatur wie M, die in einer Schnittstellendeklaration deklariert sind, werden aus der Menge entfernt.
* Nachdem alle verdeckten Member entfernt wurden, steht das Ergebnis der Suche fest:
* Wenn die Menge aus einem einzelnen Member besteht, der keine Methode ist, ist dieser Member das Ergebnis der Suche.
* Wenn die Gruppe nur aus Methoden besteht, ist diese Methodengruppe das Ergebnis der Suche.
* Andernfalls ist die Suche mehrdeutig, und es tritt ein Bindungsfehler auf.

Bei Membersuchvorgängen in Typen, die keine Typparameter und Schnittstellen sind, bzw. bei Membersuchvorgängen in Schnittstellen, die ausschließlich einfach vererbt sind (jede Schnittstelle hat in der Vererbungskette genau eine oder keine direkte Basisschnittstelle), wirken sich die Suchregeln so aus, dass abgeleitete Member Basismember mit demselben Namen oder derselben Signatur verdecken. Suchvorgänge mit einfacher Vererbung sind stets eindeutig. Mehrdeutigkeiten, die sich aus Membersuchen in Schnittstellen mit Mehrfachvererbung ergeben, werden in §13.2.5 beschrieben.

### Basistypen

Bei der Membersuche wird ein Typ T mit den folgenden Basistypen angenommen:

* Wenn T ein object ist, weist T keine Basistypen auf.
* Wenn T ein enum-type ist, sind die Basistypen von T die Klassentypen System.Enum, System.ValueType und object.
* Wenn T ein struct-type ist, sind die Basistypen von T die Klassentypen System.ValueType und object.
* Wenn T ein class-type ist, sind die Basistypen von T die Basisklassen von T, einschließlich des Klassentyps object.
* Wenn T ein interface-type ist, sind die Basistypen von T die Basisschnittstellen von T und der Klassentyp object.
* Wenn T ein array-type ist, sind die Basistypen von T die Klassentypen System.Array und object.
* Wenn T ein delegate-type ist, sind die Basistypen von T die Klassentypen System.Delegate und object.

## Funktionsmember

Bei Funktionsmembern handelt es sich um Member, die ausführbare Anweisungen enthalten. Funktionsmember sind immer Member von Typen und nicht von Namespaces. C# definiert die folgenden Kategorien von Funktionsmembern:

* Methoden
* Eigenschaften
* Ereignisse
* Indexer
* Benutzerdefinierte Operatoren
* Instanzkonstruktoren
* Statische Konstruktoren
* Destruktoren

Mit Ausnahme von Destruktoren und statischen Konstruktoren (die nicht explizit aufgerufen werden können) werden die in Funktionsmembern enthaltenen Anweisungen über Funktionsmemberaufrufe ausgeführt. Die eigentliche Syntax für einen Funktionsmemberaufruf ist je nach der Funktionsmemberkategorie unterschiedlich.

Die Argumentenliste (§7.5.1) eines Funktionsmemberaufrufs enthält die tatsächlichen Wert- oder Variablenverweise für die Parameter des Funktionsmembers.

Bei Aufrufen von generischen Methoden kann ein Typrückschluss verwendet werden, um die Gruppe von Typargumenten zu bestimmen, die an die Methode übergeben werden sollen. Dieser Vorgang wird in §7.5.2 beschrieben.

Bei Aufrufen von Methoden, Indexern, Operatoren und Instanzkonstruktoren wird mithilfe von Überladungsauflösung bestimmt, welche Gruppe potenzieller Funktionsmember aufgerufen wird. Dieser Vorgang wird in §7.5.3 beschrieben.

Nachdem ein bestimmter Funktionsmember während der Bindung gefunden wurde (möglicherweise durch Überladungsauflösung), beginnt der eigentliche Laufzeitprozess des Funktionsmemberaufrufs (siehe §7.5.4).

In der folgenden Tabelle sind die Prozesse zusammengefasst, die in Konstrukten mit allen sechs Kategorien von explizit aufrufbaren Funktionsmembern ausgeführt werden. In der Tabelle stehen e, x, y und value für als Variablen oder Werte klassifizierte Ausdrücke, T bezeichnet einen als Typ klassifizierten Ausdruck, F ist der einfache Name einer Methode und P der einfache Name einer Eigenschaft.

| **Konstrukt** | **Beispiel** | **Beschreibung** |
| --- | --- | --- |
| Methodenaufruf | F(x, y) | Durch Überladungsauflösung wird die am besten geeignete Methode F der zugehörigen Klasse oder Struktur ausgewählt. Die Methode wird mit der Argumentenliste (x, y) aufgerufen. Wenn die Methode nicht static ist, dann ist der Instanzausdruck this. |
| T.F(x, y) | Durch Überladungsauflösung wird die am besten geeignete Methode F der Klasse oder Struktur T ausgewählt. Wenn die Methode nicht static ist, tritt ein Bindungsfehler auf. Die Methode wird mit der Argumentenliste (x, y) aufgerufen. |
| e.F(x, y) | Durch Überladungsauflösung wird die am besten geeignete Methode F der Klasse, Struktur oder Schnittstelle des Typs e ausgewählt. Wenn die Methode static ist, tritt ein Bindungsfehler auf. Die Methode wird mit dem Instanzausdruck e und der Argumentliste (x, y) aufgerufen. |
| Eigenschaftenzugriff | P | Es wird der get-Accessor der Eigenschaft P der zugehörigen Klasse oder Struktur aufgerufen. Wenn P lesegeschützt, tritt ein Kompilierungsfehler auf. Wenn P nicht static ist, dann ist der Instanzausdruck this. |
| P = value | Der set-Accessor der Eigenschaft P der zugehörigen Klasse oder Struktur T wird mit der Argumentliste (value) aufgerufen. Wenn P schreibgeschützt, tritt ein Kompilierungsfehler auf. Wenn P nicht static ist, dann ist der Instanzausdruck this. |
| T.P | Es wird der get-Accessor der Eigenschaft P in der Klasse oder Struktur T aufgerufen. Wenn P nicht static oder wenn P lesegeschützt ist, tritt ein Kompilierungsfehler auf. |
| T.P = value | Der set-Accessor der Eigenschaft P in der Klasse oder Struktur T wird mit der Argumentliste (value) aufgerufen. Wenn P nicht static oder wenn P schreibgeschützt ist, tritt ein Kompilierungsfehler auf. |
| e.P | Der get-Accessor der Eigenschaft P der durch den Typ e vorgegebenen Klasse, Struktur oder Schnittstelle wird mit dem Instanzausdruck e aufgerufen. Wenn P nicht static oder wenn P lesegeschützt ist, tritt ein Bindungsfehler auf. |
| e.P = value | Der set-Accessor der Eigenschaft P der durch den Typ e vorgegebenen Klasse, Struktur oder Schnittstelle wird mit dem Instanzausdruck e und der Argumentliste (value) aufgerufen. Wenn P nicht static oder wenn P schreibgeschützt ist, tritt ein Bindungsfehler auf. |
| Ereigniszugriff | E += value | Der add-Accessor des Ereignisses E der zugehörigen Klasse oder Struktur wird aufgerufen. Wenn E nicht static ist, dann ist der Instanzausdruck this. |
| E -= value | Der remove-Accessor des Ereignisses E der zugehörigen Klasse oder Struktur wird aufgerufen. Wenn E nicht static ist, dann ist der Instanzausdruck this. |
| T.E += value | Es wird der add-Accessor des Ereignisses E in der Klasse oder Struktur T aufgerufen. Wenn E nicht static ist, tritt ein Bindungsfehler auf. |
| T.E -= value | Es wird der remove-Accessor des Ereignisses E in der Klasse oder Struktur T aufgerufen. Wenn E nicht static ist, tritt ein Bindungsfehler auf. |
| e.E += value | Der add-Accessor des Ereignisses E der durch den Typ e vorgegebenen Klasse, Struktur oder Schnittstelle wird mit dem Instanzausdruck e aufgerufen. Wenn E static ist, tritt ein Bindungsfehler auf. |
| e.E -= value | Der remove-Accessor des Ereignisses E der durch den Typ e vorgegebenen Klasse, Struktur oder Schnittstelle wird mit dem Instanzausdruck e aufgerufen. Wenn E static ist, tritt ein Bindungsfehler auf. |
| Indexerzugriff | e[x, y] | Durch Überladungsauflösung wird der am besten geeignete Indexer in der durch den Typ e vorgegebenen Klasse, Struktur oder Schnittstelle ausgewählt. Der get-Accessor des Indexers wird mit dem Instanzausdruck e und der Argumentliste (x, y) aufgerufen. Wenn der Indexer lesegeschützt ist, tritt ein Bindungsfehler auf. |
| e[x, y] = value | Durch Überladungsauflösung wird der am besten geeignete Indexer in der Klasse, Struktur oder Schnittstelle des Typs e ausgewählt. Der set-Accessor des Indexers wird mit dem Instanzausdruck e und der Argumentliste (x, y, value) aufgerufen. Wenn der Indexer schreibgeschützt ist, tritt ein Bindungsfehler auf. |
| Operatoraufruf | -x | Durch Überladungsauflösung wird der am besten geeignete unäre Operator der durch den Typ x vorgegebenen Klasse oder Struktur ausgewählt. Der ausgewählte Operator wird mit der Argumentliste (x) aufgerufen. |
| x + y | Durch Überladungsauflösung wird der am besten geeignete binäre Operator in den durch die Typen x und y vorgegebenen Klassen oder Strukturen ausgewählt. Der ausgewählte Operator wird mit der Argumentliste (x, y) aufgerufen. |
| Instanzkonstruktoraufruf | new T(x, y) | Durch Überladungsauflösung wird der am besten geeignete Instanzenkonstruktor in der Klasse oder Struktur T ausgewählt. Der Instanzkonstruktor wird mit der Argumentliste (x, y) aufgerufen. |

### Argumentlisten

Alle Aufrufe von Funktionsmembern und Delegaten enthalten eine Argumentliste mit den tatsächlichen Wert- oder Variablenverweisen für die Parameter des Funktionsmembers. Die eigentliche Syntax zur Festlegung einer Argumentliste zu einem Funktionsmemberaufruf ist je nach der Funktionsmemberkategorie unterschiedlich:

* Für Instanzkonstruktoren, Methoden, Indexer und Delegaten werden die Argumente, wie im Folgenden beschrieben, als argument-list festgelegt. Bei Indexern enthält die Argumentliste beim Aufruf des set-Accessors zusätzlich den Ausdruck, der als rechter Operand des Zuweisungsoperators festgelegt wurde.
* Für Eigenschaften ist die Argumentliste leer, wenn der get-Accessor aufgerufen wird. Sie besteht aus dem Ausdruck, der als rechter Operand des Zuweisungsoperators beim Aufruf des set-Accessors festgelegt wurde.
* Bei Ereignissen besteht die Argumentliste aus dem Ausdruck, der als rechter Operand des +=- oder -=-Operators festgelegt wurde.
* Für benutzerdefinierte Operatoren besteht die Argumentliste aus einem einzigen Operanden des unären Operators oder aus beiden Operanden des binären Operators.

Die Argumente von Eigenschaften (§10.7), Ereignissen (§10.8) und benutzerdefinierten Operatoren (§10.10) werden immer als Werteparameter (§10.6.1.1) übergeben. Die Argumente von Indexern (§10.9) werden immer als Wertparameter (§10.6.1.1) oder Parameterarrays (§10.6.1.4) übergeben. Verweis- und Ausgabeparameter werden für diese Kategorien von Funktionsmembern nicht unterstützt.

Die Argumente eines Instanzkonstruktors, einer Methode, eines Indexers oder eines Delegataufrufs werden als argument-list festgelegt:

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

Eine argument-list besteht aus einem oder mehreren arguments, die durch Trennzeichen getrennt sind. Jedes Argument besteht aus einem optionalen argument-name, auf den ein argument-value folgt. Ein argument mit einem argument-name wird als benanntes Argument bezeichnet, wohingegen es sich bei einem argument ohne argument-name um ein Positionsargument handelt. Ein Positionsargument darf nicht nach einem benannten Argument in einer argument-list stehen.

Der argument-value kann in folgender Form vorliegen:

* Als expression, der angibt, dass das Argument als Werteparameter (§10.6.1.1) übergeben wird.
* Als Schlüsselwort ref gefolgt von einer variable-reference (§5.4), die angibt, dass das Argument als Verweisparameter (§10.6.1.2) übergeben wird. Eine Variable muss endgültig zugewiesen werden (§5.3), bevor sie als Verweisparameter weitergegeben werden kann. Als Schlüsselwort out gefolgt von einer variable-reference (§5.4), die angibt, dass das Argument als Ausgabeparameter (§10.6.1.3) übergeben wird. Nach einem Funktionsmemberaufruf, in dem eine Variable als Ausgabeparameter übergeben wird, gilt die Variable als definitiv zugewiesen (§5.3).

#### Zusammengehörige Parameter

Für jedes Argument in einer Argumentliste muss es einen entsprechenden Parameter im Funktionsmember oder Delegat geben, der aufgerufen wird.

Die nachfolgend verwendete Parameterliste wird folgendermaßen bestimmt:

* Bei virtuellen Methoden und Indexern, die in Klassen definiert sind, wird die Parameterliste aus der spezifischsten Deklaration oder Überschreibung des Funktionsmembers ausgewählt, angefangen mit dem statischen Typ des Empfängers. Anschließend werden die Basisklassen des Funktionsmembers durchsucht.
* Bei Schnittstellenmethoden und Indexern wird die Parameterliste aus der spezifischsten Definition des Members ausgewählt, angefangen mit dem Schnittstellentyp. Anschließend werden die Basisschnittstellen durchsucht. Wenn keine eindeutige Parameterliste gefunden wird, wird eine Parameterliste mit nicht zugänglichen Namen und ohne optionale Parameter erstellt, sodass Aufrufe keine benannten Parameter verwenden oder optionale Argumente auslassen können.
* Bei partiellen Methoden wird die Parameterliste der definierenden partiellen Methodendeklaration verwendet.
* Bei allen anderen Funktionsmembern und Delegaten gibt es nur eine einzige Parameterliste, die verwendet wird.

Die Position eines Arguments oder Parameters ist definiert als die Anzahl von Argumenten oder Parametern, die in der Argumentliste oder Parameterliste davor stehen.

Die entsprechenden Parameter für Funktionsmemberargumente werden wie folgt eingerichtet:

* Argumente in der argument-list von Instanzkonstruktoren, Methoden, Indexern und Delegaten:
* Ein Positionsargument, in dem ein fester Parameter, der an derselben Position in der Parameterliste steht, diesem Parameter entspricht.
* Ein Positionsargument eines Funktionsmembers mit einem Parameterarray, das in seiner normalen Form aufgerufen wird, entspricht dem Parameterarray, das an derselben Position in der Parameterliste stehen muss.
* Ein Positionsargument eines Funktionsmembers mit einem Parameterarray, das in seiner erweiterten Form aufgerufen wird, wobei kein fester Parameter, der an derselben Position in der Parameterliste steht, einem Element in dem Parameterarray entspricht.
* Ein benanntes Argument entspricht dem Parameter desselben Namens in der Parameterliste.
* Bei Indexern entspricht der als rechter Operand des Zuweisungsoperators festgelegte Ausdruck beim Aufruf des set-Accessors dem impliziten value-Parameter der set-Accessordeklaration.
* Für Eigenschaften gibt es beim Aufruf des get-Accessors keine Argumente. Beim Aufrufen des set-Accessors entspricht der als rechter Operand des Zuweisungsoperators festgelegte Ausdruck dem impliziten value-Parameter der set-Accessordeklaration.
* Für benutzerdefinierte unäre Operatoren (einschließlich Konvertierungen) entspricht der einzige Operand dem einzigen Parameter der Operatordeklaration.
* Für benutzerdefinierte binäre Operatoren entspricht der linke Operand dem ersten Parameter, und der rechte Operand entspricht dem zweiten Parameter der Operatordeklaration.

#### Laufzeitauswertung von Argumentlisten

Während der Laufzeitverarbeitung eines Funktionsmemberaufrufs (§7.5.4) werden die Ausdrücke oder Variablenverweise einer Argumentliste von links nach rechts wie folgt ausgewertet:

* Für einen Werteparameter wird der Argumentausdruck ausgewertet und implizit in den entsprechenden Parametertyp umgewandelt (§6.1). Der daraus resultierende Wert wird als erster Wert des Werteparameters im Funktionsmemberaufruf verwendet.
* Für einen Verweis- oder Ausgabeparameter wird der Variablenverweis ausgewertet und als Speicherplatz der durch die Parameter im Funktionsmemberaufruf festgelegte Speicherplatz ermittelt. Wenn der als Verweis- oder Ausgabeparameter angegebene Variablenverweis ein Arrayelement eines reference-type ist, wird eine Laufzeitprüfung durchgeführt, um sicherzustellen, dass der Elementtyp des Arrays mit dem Typ des Parameters identisch ist. Wenn diese Prüfung negativ ausfällt, wird System.ArrayTypeMismatchException ausgelöst.

Methoden, Indexer und Instanzkonstruktoren können ggf. den am weitesten rechts stehenden Parameter als Parameterarray (§10.6.1.4) deklarieren. Solche Funktionsmember werden entweder in ihrer normalen oder in ihrer erweiterten Form aufgerufen, je nachdem, welche geeignet ist (§7.5.3.1):

* Wenn ein Funktionsmember mit einem Parameterarray in seiner normalen Form aufgerufen wird, muss das für das Parameterarray vorgegebene Argument ein einziger Ausdruck sein, der implizit in den Typ des Parameterarrays konvertierbar ist (§6.1). In diesem Fall hat das Parameterarray dieselbe Funktion wie ein Werteparameter.
* Wenn ein Funktionsmember mit einem Parameterarray in seiner erweiterten Form aufgerufen wird, muss der Aufruf keine oder mehrere Positionsargumente für das Parameterarray angeben, wobei jedes Argument ein Ausdruck ist, der implizit in den Elementtyp des Parameterarrays konvertierbar ist (§6.1). In diesem Fall erstellt der Aufruf eine Instanz des Parameterarraytyps mit einer Länge, die der Anzahl der Argumente entspricht, initialisiert die Elemente der Arrayinstanz mit den angegebenen Argumentwerten und verwendet die neu erstellte Arrayinstanz als das tatsächliche Argument.

Die Ausdrücke einer Argumentliste werden stets in der Reihenfolge ausgewertet, in der sie geschrieben werden. Das Beispiel

class Test  
{  
 static void F(int x, int y = -1, int z = -2) {  
 System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);  
 }

static void Main() {  
 int i = 0;  
 F(i++, i++, i++);  
 F(z: i++, x: i++);  
 }  
}

erzeugt die Ausgabe

x = 0, y = 1, z = 2  
x = 4, y = -1, z = 3

Durch die Arraykovarianzregeln (§12.5) wird ermöglicht, dass ein Wert eines Arraytyps A[] ein Verweis auf eine Instanz eines Arraytyps B[] sein kann, sofern eine implizite Verweiskonvertierung von B in A vorhanden ist. Wenn ein Arrayelement eines reference-type als Verweis- oder Ausgabeparameter übergeben wird, ist aufgrund dieser Regeln eine Laufzeitprüfung erforderlich, um sicherzustellen, dass der tatsächliche Elementtyp des Arrays mit dem Typ des Parameters identisch ist. In dem Beispiel

class Test  
{  
 static void F(ref object x) {...}

static void Main() {  
 object[] a = new object[10];  
 object[] b = new string[10];  
 F(ref a[0]); // Ok  
 F(ref b[1]); // ArrayTypeMismatchException  
 }  
}

verursacht der zweite Aufruf von F das Auslösen einer System.ArrayTypeMismatchException, da der eigentliche Elementtyp von b nicht vom Typ object, sondern vom Typ string ist.

Wenn ein Funktionsmember mit einem Parameterarray in seiner erweiterten Form aufgerufen wird, wird der Aufruf genauso verarbeitet, als ob ein Arrayerstellungsausdruck mit einem Arrayinitialisierer (§7.6.10.4) um die erweiterten Parameter eingefügt wurde. Bei den Deklarationen

void F(int x, int y, params object[] args);

stimmen die Aufrufe der erweiterten Form der Methode

F(10, 20);  
F(10, 20, 30, 40);  
F(10, 20, 1, "hello", 3.0);

genau überein mit

F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});

Wenn für das Parameterarray keine Argumente angegeben sind, wird ein leeres Array erstellt.

Wenn Argumente in einem Funktionsmember mit entsprechenden optionalen Parameter ausgelassen werden, werden die Standardargumente der Funktionsmemberdeklaration implizit übergeben. Da diese immer konstant sind, hat ihre Auswertung keine Auswirkung auf die Auswertungsreihenfolge der verbleibenden Argumente.

### Typrückschluss

Wenn eine generische Methode ohne Angabe von Typargumenten aufgerufen wird, versucht der Prozess des Typrückschlusses, die Typargumente für den Aufruf abzuleiten. Die Typherleitung ermöglicht es, für Aufrufe generischer Methoden eine einfachere Syntax zu verwenden, und erspart dem Programmierer die Angabe redundanter Typinformationen. Beispielsweise kann bei folgender Methodendeklaration:

class Chooser  
{  
 static Random rand = new Random();

public static T Choose<T>(T first, T second) {  
 return (rand.Next(2) == 0)? first: second;  
 }  
}

die Choose-Methode ohne explizite Angabe eines Typarguments aufgerufen werden:

int i = Chooser.Choose(5, 213); // Calls Choose<int>

string s = Chooser.Choose("foo", "bar"); // Calls Choose<string>

Über den Typrückschluss werden die Typargumente int und string anhand der Argumente für die Methode bestimmt.

Der Typrückschluss erfolgt als Teil der Verarbeitung eines Methodenaufrufs zur Bindungszeit (§7.6.5.1) und vor der Überladungsauflösung im Rahmen des Aufrufs. Wenn in einem Methodenaufruf eine bestimmte Methodengruppe angegeben wird und der Methodenaufruf keine Typargumente enthält, wird die Typherleitung auf alle generischen Methoden in der Methodengruppe angewendet. Wenn die Typherleitung erfolgreich ausgeführt wird, werden die Typen der Argumente für die spätere Überladungsauflösung anhand der abgeleiteten Typargumente bestimmt. Wenn bei der Überladungsauflösung eine generische Methode als die aufzurufende Methode identifiziert wird, werden die abgeleiteten Typargumente als die eigentlichen Typargumente für den Aufruf verwendet. Wenn bei der Typherleitung für eine bestimmte Methode ein Fehler auftritt, wird diese Methode bei der Überladungsauflösung nicht berücksichtigt. Ein Fehler beim Typrückschluss führt noch nicht zu einem Bindungsfehler. Hingegen führt er oft zu einem Bindungsfehler, wenn daraufhin die Überladungsauflösung keine geeigneten Methoden finden kann.

Wenn sich die angegebene Anzahl von Argumenten von der Anzahl von Parametern in der Methode unterscheidet, gibt die Typherleitung unmittelbar einen Fehler aus. Angenommen, die Methode weist stattdessen eine Signatur ähnlich der Folgenden auf:

Tr M<X1…Xn>(T1 x1 … Tm xm)

Bei einem Methodenaufruf der M(E1 …Em) hat der Typrückschluss die Aufgabe, eindeutige Typargumente S1…Sn für die einzelnen Typparameter X1…Xn zu finden, damit der Aufruf M<S1…Sn>(E1…Em)gültig wird.

Während des Rückschlusses werden die einzelnen Typparameter Xi entweder auf einen bestimmten Typ Si *fixiert*, oder sie sind *nicht fixiert* mit einem zugehörigen Satz von *Grenzen.* ede dieser Grenzen entspricht einem Typ T. Anfänglich sind alle Typvariablen Xi nicht fixiert mit einem leeren Satz von Grenzen.

Der Typrückschluss erfolgt in mehreren Phasen. Bei jeder Phase wird versucht, anhand der Ergebnis der vorherigen Phase auf Typargumente für weitere Typvariablen zu schließen. Während der ersten Phase werden einige anfängliche Rückschlüsse von Grenzen durchgeführt, und in der zweiten Phase werden dann Typvariablen auf bestimmte Typen fixiert und auf weitere Grenzen geschlossen. Die zweite Phase muss möglicherweise mehrmals wiederholt werden.

*Hinweis:* Typrückschlüsse treten nicht nur beim Aufruf einer generischen Methode auf. Typrückschlüsse für die Konvertierung von Methodengruppen werden in §7.5.2.13 beschrieben. Informationen zum Ermitteln des optimalen Typs für einen Satz von Ausdrücken finden Sie in §7.5.2.14.

#### Die erste Phase

Für jedes Methodenargument Ei gilt:

* Ist Ei eine anonyme Funktion, wird ein expliziter Parametertyprückschluss (§7.5.2.7) von Ei in Ti durchgeführt.
* Wenn andernfalls Ei den Typ U aufweist und xi ein Werteparameter ist, wird ein *Rückschluss auf die Untergrenze* von U in Ti durchgeführt.
* Wenn andernfalls Ei einen Typ U aufweist und xi ein ref- oder out-Parameter ist, wird ein genauer Rückschluss von U in Ti durchgeführt.
* Andernfalls werden keine Rückschlüsse für dieses Argument durchgeführt.

#### Die zweite Phase

Die zweite Phase läuft folgendermaßen ab:

* Sämtliche nicht fixierten Typvariablen Xi, die von keinem Xj abhängig sind (§7.5.2.5), werden fixiert (§7.5.2.10)).
* Sind keine solchen Typvariablen vorhanden, werden alle nicht fixierten Typvariablen Xi fixiert, für die alle der folgenden Bedingungen gelten:
  + Es ist mindestens eine Typvariable Xj vorhanden, die von Xi abhängig ist.
  + Der Satz von Grenzen für Xi ist nicht leer.
* Wenn keine solchen Typvariablen vorhanden sind, aber noch Typvariablen nicht fixiert wurden, schlägt der Typrückschluss fehl.
* Wenn andernfalls keine weiteren nicht fixierten Typvariablen vorhanden sind, wird der Typrückschluss erfolgreich abgeschlossen.
* Trifft dies nicht zu, wird für alle Argumente Ei mit einem zugehörigen Parametertyp Ti ein Ausgabetyprückschluss (§7.5.2.4) von Ei in Ti durchgeführt, wenn die Ausgabetypen (§7.5.2.3) nicht fixierte Typvariablen Xj umfassen, die Eingabetypen (§7.5.2.6) hingegen nicht. Anschließend wird die zweite Phase wiederholt.

#### Eingabetypen

Wenn E eine Methodengruppe oder eine implizit typisierte anonyme Funktion und T ein Delegattyp oder ein Ausdrucksbaumstrukturtyp ist, sind alle Parametertypen von T *Eingabetypen* *von* E *mit dem Typ* T.

#### Ausgabetypen

Wenn E eine Methodengruppe oder eine anonyme Funktion und T ein Delegattyp oder ein Ausdrucksbaumstrukturtyp ist, ist der Rückgabetyp von T ein *Ausgabetyp* *von* E *mit dem Typ* T.

#### Abhängigkeit

Eine nicht fixierte Typvariable Xi ist *direkt abhängig* von einer nicht fixierten Typvariablen Xj, wenn für ein Argument Ek mit dem Typ Tk Xj in einem Eingabetyp von Ek mit dem Typ Tk und Xi in einem Ausgabetyp von Ek mit dem Typ Tk auftritt.

Xj *ist abhängig von* Xi, wenn Xj direkt abhängig von Xi ist oder wenn Xi direkt abhängig von Xk ist und Xk abhängig von Xj ist. Insofern ist "ist abhängig von" der transitive, aber nicht reflexive Abschluss von "ist direkt abhängig von".

#### Ausgabetyprückschluss

Ein Ausgabetyprückschluss von einem Ausdruck E in einen Typ T wird folgendermaßen durchgeführt:

* Wenn E eine anonyme Funktion mit dem abgeleiteten Rückgabetyp U (§7.5.2.12) und T ein Delegattyp oder ein Ausdrucksbaumstrukturtyp mit Rückgabetyp Tb ist, wird ein Rückschluss auf die Untergrenze (§7.5.2.9) von von U in Tb durchgeführt.
* Wenn andernfalls E eine Methodengruppe und T ein Delegattyp oder ein Ausdrucksbaumstrukturtyp mit den Parametertypen T1…Tk und dem Rückgabetyp Tb ist und des Weiteren die Auflösung der Überladung von E mit den Typen T1…Tk eine einzelne Methode mit dem Rückgabetyp U ergibt, wird ein Rückschluss auf die Untergrenze von U in Tb durchgeführt.
* Wenn andernfalls E ein Ausdruck mit dem Typ U ist, wird ein Rückschluss auf die Untergrenze von U in T durchgeführt.
* Treffen diese Aussagen nicht zu, werden keine Rückschlüsse durchgeführt.

#### Explizite Parametertyprückschlüsse

Ein expliziter Parametertyprückschluss von einem Ausdruck E in den Typ T wird folgendermaßen durchgeführt:

* Wenn E eine explizit typisierte anonyme Funktion mit den Parametertypen U1…Uk und T ein Delegattyp oder ein Ausdrucksbaumstrukturtyp mit den Parametertypen V1…Vk ist, wird für jedes Ui ein genauer Rückschluss (§7.5.2.8) von Ui in den zugehörigen Vi durchgeführt.

#### Genaue Rückschlüsse

Ein genauer Rückschluss von einem Typ U in einen Typ V wird folgendermaßen durchgeführt:

* Wenn V ein nicht fixiertes Xi ist, wird U dem Satz von genauen Grenzen für Xi hinzugefügt.
* Andernfalls werden die Sätze V1…Vk und U1…Uk ermittelt, indem geprüft wird, ob einer der folgenden Fälle zutrifft:
* V ist ein Arraytyp V1[…] , und U ist ein Arraytyp U1[…] mit demselben Rang.
* V ist der Typ V1?, und U ist der Typ U1?
* V ist ein konstruierter Typ C<V1…Vk> and U ist ein konstruierter Typ C<U1…Uk>

Wenn einer dieser Fälle zutrifft, wird ein genauer Rückschluss von jedem Ui in den zugehörigen Typ Vi durchgeführt.

* Andernfalls werden keine Rückschlüsse durchgeführt.

#### Rückschlüsse auf Untergrenzen

Ein Rückschluss auf die Untergrenze von einem Typ U in einen Typ V wird folgendermaßen durchgeführt:

* Wenn V ein nicht fixiertes Xi ist, wird U dem Satz von unteren Grenzen für Xi hinzugefügt.
* Wenn andernfalls V über den Typ V1? und U über den Typ U1? verfügt, wird ein Rückschluss auf die Untergrenze von U1 in V1 durchgeführt.
* Andernfalls werden die Sätze U1…Uk und V1…Vk ermittelt, indem geprüft wird, ob einer der folgenden Fälle zutrifft:
* V ist ein Arraytyp V1[…] und U ist ein Arraytyp U1[…] (oder ein Typparameter, dessen effektiver Basistyp U1[…] ist) mit demselben Rang.
* V ist IEnumerable<V1>, ICollection<V1> oder IList<V1> und U ist ein eindimensionaler Arraytyp U1[](oder ein Typparameter, dessen effektiver Basistyp U1[] ist).
* V ist eine konstruierte Klasse, eine Struktur, eine Schnittstelle oder ein Delegattyp C<V1…Vk>, und es gibt einen eindeutigen Typ C<U1…Uk> , sodass U (oder seine effektive Basisklasse bzw. ein Member der effektiven Schnittstellengruppe, falls U ein Typparameter ist) identisch ist mit C<U1…Uk>., davon (direkt oder indirekt) erbt und diesen Typ (direkt oder indirekt) implementiert.

(Die Einschränkung der "Eindeutigkeit" bedeutet, dass im Falle von interface C<T>{} class U: C<X>, C<Y>{} bei einem Rückschluss von U in C<T> kein Rückschluss erfolgt, da U1 X oder Y sein könnte).

Wenn einer dieser Fälle zutrifft, wird ein Rückschluss von jedem Ui in den zugehörigen Typ Vi folgendermaßen durchgeführt:

* Wenn Ui nicht als Verweistyp bekannt ist, wird ein *genauer Rückschluss* durchgeführt.
* Andernfalls wird ein *Rückschluss auf die Untergrenze* durchgeführt, wenn U ein Arraytyp ist.
* Andernfalls ist der Rückschluss von dem i-ten Typparameter von C abhängig, wenn V gleich C<V1…Vk> ist.
* Wenn V kovariant ist, wird ein *Rückschluss auf die Untergrenze* durchgeführt.
* Wenn V kontravariant ist, wird ein *Rückschluss auf die Obergrenze* durchgeführt.
* Wenn V invariant ist, wird ein *genauer Rückschluss* durchgeführt.
* Treffen diese Aussagen nicht zu, werden keine Rückschlüsse durchgeführt.

#### Rückschlüsse auf Obergrenzen

Ein Rückschluss auf die Obergrenze von einem Typ U in einen Typ V wird folgendermaßen durchgeführt:

* Wenn V ein nicht fixiertes Xi ist, wird U dem Satz von oberen Grenzen für Xi hinzugefügt.
* Andernfalls werden die Sätze V1…Vk und U1…Uk ermittelt, indem geprüft wird, ob einer der folgenden Fälle zutrifft:
* U ist ein Arraytyp U1[…], und V ist ein Arraytyp V1[…]mit demselben Rang.
* U ist IEnumerable<Ue>, ICollection<Ue> oder IList<Ue>, und V ist ein eindimensionaler Arraytyp Ve[]
* U ist der Typ U1?, und V ist der Typ V1?
* U ist eine konstruierte Klasse, eine Struktur, eine Schnittstelle oder ein Delegattyp C<U1…Uk>, und V st eine Klasse, eine Struktur, eine Schnittstelle oder ein Delegattyp, die bzw. der identisch ist mit dem eindeutigen Typ C<V1…Vk>, davon (direkt oder indirekt) erbt und diesen (direkt oder indirekt) implementiert.

(Die Einschränkung der "Eindeutigkeit" bedeutet, dass im Falle von interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{} bei einem Rückschluss von C<U1> in V<Q> kein Rückschluss erfolgt. Von U1 werden keine Rückschlüsse in X<Q> oder Y<Q> durchgeführt.)

Wenn einer dieser Fälle zutrifft, wird ein Rückschluss von jedem Ui in den zugehörigen Typ Vi folgendermaßen durchgeführt:

* Wenn Ui nicht als Verweistyp bekannt ist, wird ein *genauer Rückschluss* durchgeführt.
* Andernfalls wird ein *Rückschluss auf die Obergrenze* durchgeführt, wenn V ein Arraytyp ist.
* Andernfalls ist der Rückschluss von dem i-ten Typparameter von C abhängig, wenn U gleich C<U1…Uk> ist.
* Wenn U kovariant ist, wird ein *Rückschluss auf die Obergrenze* durchgeführt.
* Wenn U kontravariant ist, wird ein *Rückschluss auf die Untergrenze* durchgeführt.
* Wenn V invariant ist, wird ein *genauer Rückschluss* durchgeführt.
* Treffen diese Aussagen nicht zu, werden keine Rückschlüsse durchgeführt.

#### Fixieren

Eine nicht fixierte Typvariable Xi mit einem Satz von Grenzen wird folgendermaßen fixiert:

* Der Satz von *infrage kommenden Typen* Uj umfasst anfänglich alle Typen im Satz der Grenzen von Xi.
* Anschließend werden die Grenzen von Xi nacheinander untersucht: Für jede Grenze U von Xi werden alle Typen Uj, die nicht identisch mit U sind, aus dem Satz der möglichen Typen entfernt. Für jede untere Grenze U von Xi werden alle Typen Uj, für die *keine* implizite Konvertierung von U vorhanden ist, aus dem Satz der möglichen Typen entfernt. Für jede obere Grenze U von Xi werden alle Typen Uj, für die *keine* implizite Konvertierung in U vorhanden ist, aus dem Satz der möglichen Typen entfernt.
* Wenn die restlichen infrage kommenden Typen Uj einen eindeutigen Typen V enthalten, für den eine implizite Konvertierung in alle anderen infrage kommenden Typen vorhanden ist, wird Xi auf V fixiert.
* Andernfalls schlägt der Typrückschluss fehl.

#### Hergeleiteter Rückgabetyp

Der hergeleitete Rückgabetyp einer anonymen Funktion F wird beim Typrückschluss und bei der Auflösung von Überladungen verwendet. Der hergeleitete Rückgabetyp für eine anonyme Funktion kann nur dann bestimmt werden, wenn sämtliche Parametertypen bekannt sind. Dies trifft zu, wenn diese explizit angegeben werden, wenn sie über die Konvertierung einer anonymen Funktion angegeben werden oder wenn sie während des Typrückschlusses beim Aufruf auf eine einschließende generische Methode geschlossen werden.

Der hergeleitete Ergebnistyp wird folgendermaßen bestimmt:

* Wenn der Text von F eine expression mit einem Typ ist, ist der hergeleitete Rückgabetyp von F der Typ dieses Ausdrucks.
* Wenn der Text von F ein block ist und der Satz von Ausdrücken in den return-Anweisungen des Blocks über einen optimalen Typ T (§7.5.2.14) verfügt, ist T der hergeleitete Rückgabetyp von F.
* Andernfalls kann auf keinen Ergebnistyp für F geschlossen werden.

Der hergeleitete Rückgabetyp wird folgendermaßen bestimmt:

* Wenn F "async" ist und der Text von F entweder ein als Nothing klassifizierter Ausdruck (§7.1) oder ein Anweisungsblock ist, bei dem keine Rückgabeanweisungen über Ausdrücke verfügen, ist der hergeleitete Rückgabetyp System.Threading.Tasks.Task
* Wen F asynchron ist und über einen hergeleiteten Ergebnistyp T verfügt, ist System.Threading.Tasks.Task<T> der hergeleitete Rückgabetyp.
* Wenn F nicht "async" ist und einen hergeleiteten Ergebnistyp T aufweist, ist der hergeleitete Rückgabetyp T.
* Andernfalls kann auf keinen Rückgabetyp für F geschlossen werden.

Ein Beispiel für einen Typrückschluss mit anonymen Funktionen ist die Erweiterungsmethode Select, die in der System.Linq.Enumerable-Klasse deklariert ist:

namespace System.Linq  
{  
 public static class Enumerable  
 {  
 public static IEnumerable<TResult> Select<TSource,TResult>(  
 this IEnumerable<TSource> source,  
 Func<TSource,TResult> selector)  
 {  
 foreach (TSource element in source) yield return selector(element);  
 }  
 }  
}

Wenn der System.Linq-Namespace mithilfe einer using-Klausel importiert wird und eine Customer-Klasse mit einer Name-Eigenschaft vom Typ string vorhanden ist, können Sie mit der Select-Methode Namen aus einer Kundenliste auswählen:

List<Customer> customers = GetCustomerList();  
IEnumerable<string> names = customers.Select(c => c.Name);

Der Aufruf der Erweiterungsmethode (§7.6.5.2) von Select erfolgt durch Umschreiben des Aufrufs in den Aufruf einer statischen Methode:

IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);

Da nicht explizit Typargumente angegeben wurden, werden diese mithilfe eines Typrückschlusses bestimmt. Zunächst wird das customers-Argument dem source-Parameter zugeordnet, sodass auf T als Customer geschlossen wird. Anschließend wird mithilfe des oben beschriebenen Typrückschlusses für anonyme Funktionen c dem Typ Customer und der Ausdruck c.Name dem Rückgabetyp des selector-Parameters zugeordnet, sodass auf string als Typ von S geschlossen wird. Daher ist der Aufruf mit folgendem Beispiel identisch:

Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)

Das Ergebnis ist vom Typ IEnumerable<string>.

Im folgenden Beispiel wird veranschaulicht, wie der Typrückschluss für anonyme Funktionen beim Aufruf einer generischen Methode eine Übergabe von Typinformationen zwischen Argumenten ermöglicht. Für die Methode

static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {  
 return f2(f1(value));  
}

erfolgt der Typrückschluss beim Aufruf von

double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);

folgendermaßen: Zuerst wird das Argument "1:15:30" dem value-Parameter zugeordnet, sodass auf string als Typ von X geschlossen wird. Anschließend wird dem Parameter der ersten anonymen Funktion s der hergeleitete Typ string zugeordnet, und der Ausdruck TimeSpan.Parse(s) wird dem Rückgabetyp von f1 zugeordnet, sodass für Y auf System.TimeSpan geschlossen wird. Zuletzt wird dem Parameter der zweiten anonymen Funktion t der hergeleitete Typ System.TimeSpan zugeordnet, und der Ausdruck t.TotalSeconds wird dem Rückgabetyp von f2 zugeordnet, sodass für Z auf double geschlossen wird. Daher ist das Ergebnis des Aufrufs vom Typ double.

#### Typrückschluss für Konvertierungen von Methodengruppen

Wie beim Aufruf von generischen Methoden muss auch dann ein Typrückschluss angewendet werden, wenn eine Methodengruppe M mit einer generischen Methode in einen angegebenen Delegattyp D (§6.6) konvertiert wird. Bei einer Methode

Tr M<X1…Xn>(T1 x1 … Tm xm)

und der Methodengruppe M, die dem Delegattyp D zugeordnet ist, müssen beim Typrückschluss Typargumente S1…Sn ermittelt werden, damit der Ausdruck

M<S1…Sn>

mit D kompatibel ist (§15.1).

Anders als beim Algorithmus für den Typrückschluss beim Aufruf einer generischen Methode sind in diesem Fall nur Argumenttypen vorhanden, aber keine Argumentausdrücke. Insbesondere sind keine anonymen Funktionen vorhanden, sodass der Rückschluss nicht in mehreren Phasen ablaufen muss.

Stattdessen gelten sämtliche Xi als nicht fixiert, und für jeden Argumenttyp Uj von D wird ein Rückschluss auf die Untergrenze in den zugehörigen Parametertyp Tj von M durchgeführt. Wenn für ein Xi keine Grenzen ermittelt werden können, schlägt der Typrückschluss fehl. Andernfalls werden alle Xi auf den zugehörigen Typ Si fixiert, der als Ergebnis des Typrückschlusses ermittelt wird.

#### Ermitteln des optimalen Typs für einen Satz von Ausdrücken

In einigen Situationen muss für einen Satz von Ausdrücken auf einen gemeinsamen Typ geschlossen werden. Besonders häufig werden die Elementtypen implizit typisierter Arrays und die Rückgabetypen anonymer Funktionen mit block-Texten auf diese Weise ermittelt.

Erwartungsgemäß sollte dieser Rückschluss für einen gegebenen Satz von Ausdrücken E1…Em mit dem Aufruf einer Methode

Tr M<X>(X x1 … X xm)

identisch sein, deren Argumente die Ei sind.

Das bedeutet, dass der Rückschluss mit einer nicht fixierten Typvariablen X beginnt. Ausgabetyprückschlüsse werden dann von Ei in X durchgeführt. X ist schließlich fixiert, und der resultierende Typ S ist, nach erfolgreicher Durchführung, der resultierende optimale Typ für die Ausdrücke. Wenn kein solcher Typ S vorhanden ist, weisen die Ausdrücke keinen optimalen Typ auf.

### Überladungsauflösung

Überladungsauflösungen sind Bindungsmechanismen, mit denen die am besten geeigneten Funktionsmember ausgewählt und eine gegebene Argumentliste aufgerufen sowie eine Gruppe potenzieller Funktionsmember festgelegt werden. Mithilfe von Überladungsauflösungen werden die Funktionsmember ausgewählt, mit denen folgende unterschiedliche Kontexte innerhalb von C# aufgerufen werden:

* Aufruf einer in einer invocation-expression (§7.6.5.1) bezeichneten Methode.
* Aufruf eines in einer object-creation-expression (§7.6.10.1) bezeichneten Instanzkonstruktors.
* Aufruf eines in einem element-access (§7.6.6) bezeichneten Indexeraccessors.
* Aufruf eines vordefinierten oder benutzerdefinierten Operators, auf den in einem Ausdruck (§7.3.3 und §7.3.4) verwiesen wird.

Alle diese Kontexte definieren die potenziellen Funktionsmember und die Liste der Argumente auf eigene, eindeutige Weise, die in den oben aufgeführten Abschnitten detailliert beschrieben wird. Die Gruppe von potenziellen Methodenaufrufen enthält z. B. keine Methoden mit der Bezeichnung override (§7.4), und die Methoden in einer Basisklasse können nicht potenziell eingesetzt werden, wenn eine der Methoden in einer abgeleiteten Klasse anwendbar ist (§7.6.5.1).

Sobald die potenziellen Funktionsmember und die Argumentliste festgelegt wurden, ist die Auswahl der besten Funktionsmember in allen Fällen identisch:

* Innerhalb der Gruppe der zutreffenden potenziellen Funktionsmember wird der beste Funktionsmember lokalisiert. Wenn die Gruppe nur einen Funktionsmember enthält, ist dieser automatisch der beste Funktionsmember. Andernfalls ist der beste Funktionsmember derjenige, der hinsichtlich der gegebenen Argumentenliste besser als alle anderen Funktionsmember ist, vorausgesetzt, dass alle Funktionsmember anhand der Regeln in §7.5.3.2 miteinander verglichen werden. Wenn kein Funktionsmember besser als die anderen geeignet ist, dann ist der Aufruf von Funktionsmembern nicht eindeutig, und es tritt ein Bindungsfehler auf.

In den folgenden Abschnitten wird die genaue Bedeutung der Begriffe anwendbare Funktionsmember und bessere Funktionsmember definiert.

#### Anwendbare Funktionsmember

Ein Funktionsmember wird bezüglich einer Argumentenliste A als anwendbarer Funktionsmember bezeichnet, wenn alle folgenden Bedingungen zutreffen:

* Jedes Argument A entspricht einem Parameter in der Funktionsmemberdeklaration, wie in §7.5.1.1 beschrieben, und alle Parameter, für die es kein entsprechendes Argument gibt, sind optionale Parameter.
* Für alle Argumente in A ist der Parameterübergabemodus des jeweiligen Arguments (d. h. Wert, ref oder out) mit dem Parameterübergabemodus des zugehörigen Parameters identisch, und
* für einen Werteparameter oder ein Parameterarray ist eine implizite Konvertierung (§6.1) des Arguments in den Typ des zugehörigen Parameters vorhanden, oder
* für einen ref-Parameter oder out-Parameter ist der Argumenttyp mit dem Typ des zugehörigen Parameters identisch. Insgesamt gesehen ist ein ref-Parameter oder ein out-Parameter ein Alias für das übergebene Argument.

Ein Funktionsmember, der ein Parameterarray einschließt und anhand der obigen Regeln verwendbar ist, gilt als in seiner Normalform verwendbar. Wenn ein Funktionsmember, der ein Parameterarray enthält, nicht in seiner Normalform verwendbar ist, kann dieser stattdessen in seiner erweiterten Form verwendbar sein:

* Die erweiterte Form wird erstellt, indem der Parameterarray in der Funktionsmemberdeklaration durch keine oder mehrere Werteparameter des Elementtyps des Parameterarrays ersetzt wird, sodass die Anzahl der Argumente in der Argumentenliste A der Gesamtanzahl der Parameter entspricht. Wenn A weniger Argumente aufweist als die Anzahl der feststehenden Parameter in der Funktionsmemberdeklaration, kann die erweiterte Form des Funktionsmembers nicht erstellt werden und ist somit nicht verwendbar.
* Andernfalls gilt die erweiterte Form als verwendbar, wenn alle Argumente in A denselben Parameterübergabemodus haben wie die zugehörigen Parameter, und
* für einen feststehenden Werteparameter oder einen Werteparameter, der durch die Erweiterung erzeugt wurde, eine implizite Konvertierung (§6.1) des Argumenttyps in den Typ des zugehörigen Parameters vorhanden ist, oder
* für einen ref-Parameter oder out-Parameter ist der Argumenttyp mit dem Typ des zugehörigen Parameters identisch.

#### Besserer Funktionsmember

Zur Ermittlung des besseren Funktionsmembers wird eine gekürzte Argumentliste A mit nur den Argumentausdrücken in der Reihenfolge erstellt, in der sie in der ursprünglichen Argumentliste vorkommen.

Parameterlisten für die einzelnen infrage kommenden Funktionsmember werden folgendermaßen erstellt::

* Die erweiterte Form wird verwendet, wenn der Funktionsmember nur in der erweiterten Form verwendet werden konnte.
* Optionale Parameter ohne entsprechende Argumente werden aus der Parameterliste entfernt.
* Die Parameter werden neu angeordnet, sodass sie an derselben Position wie das entsprechende Argument in der Argumentliste stehen.

Bei einer Argumentliste A mit einer Reihe von Argumentausdrücken { E1, E2, ..., EN }, zwei verwendbaren Funktionsmembern MP und MQ mit den Parametertypen { P1, P2, ..., PN } und { Q1, Q2, ..., QN } gilt MP als besserer Funktionsmember als MQ, wenn

* für alle Argumente die implizite Konvertierung von EX in QX nicht besser ist als die implizite Konvertierung von EX in PX und
* für wenigstens ein Argument die Konvertierung von EX in PX besser ist als die Konvertierung von EX in QX.

Bei dieser Auswertung verweisen MP oder MQ auf einen Parameter in der erweiterten Form der Parameterliste, wenn PX oder QX in der erweiterten Form verwendbar sind.

In dem Fall, dass die Parametertypfolgen {P1, P2, …, PN} und {Q1, Q2, …, QN} äquivalent sind (d. h. jedes Pi weist eine Identitätskonvertierung für das entsprechende Qi auf), werden die folgenden Konfliktauflösungsregeln in der angegebenen Reihenfolge angewendet, um den besseren Funktionsmember zu bestimmen.

* Wenn MP eine nicht generische Methode und MQ eine generische Methode ist, ist MP besser als MQ.
* Wenn andernfalls MP in ihrer normalen Form anwendbar ist und MQ ein params-Array besitzt und nur in ihrer erweiterten Form anwendbar ist, dann ist MP besser als MQ.
* Wenn andernfalls für MP mehr Parameter als für MQ deklariert sind, dann ist MP besser als MQ. Dies kann der Fall sein, wenn beide Methoden params-Arrays besitzen und nur in ihrer erweiterten Form anwendbar sind.
* Wenn andernfalls alle Parameter von MP über ein entsprechendes Argument verfügen, wobei die Standardargumente allerdings durch mindestens einen optionalen Parameter in MQ ersetzt werden müssen, dann ist MP besser als MQ.
* Wenn andernfalls für MP spezifischere Parameter als für MQ deklariert sind, dann ist MP besser als MQ. {R1, R2, …, RN} und {S1, S2, …, SN} stellen die nicht instanziierten und nicht erweiterten Parametertypen von MP und MQ dar. Parametertypen von MP sind spezifischer als die von MQ, wenn für jeden Parameter <RX nicht weniger spezifisch als SX und für mindestens einen Parameter RX spezifischer als SX ist:
* Ein Typparameter ist weniger spezifisch als ein Parameter, der kein Typparameter ist.
* In der Rekursion ist ein konstruierter Typ spezifischer als ein anderer konstruierter Typ (mit der gleichen Anzahl von Typargumenten), wenn mindestens ein Typargument spezifischer und kein Typargument weniger spezifisch als das entsprechende Typargument im anderen Typ ist.
* Ein Arraytyp ist spezifischer als ein anderer Arraytyp (mit der gleichen Anzahl von Dimensionen), wenn der Elementtyp des ersten Arrays spezifischer als der Elementtyp des zweiten Array ist.
* Wenn andernfalls ein Member ein nicht heraufgestufter Operator und der andere ein heraufgestufter Operator ist, gilt der nicht heraufgestufte Operator als besser.
* Andernfalls ist keiner der Funktionsmember besser.

#### Bessere Konvertierung von Ausdrücken

Bei einer impliziten Konvertierung C1, die einen Ausdruck E in einen Typ T1 konvertiert, und einer impliziten Konvertierung C2, die einen Ausdruck E in einen Typ T2 konvertiert, ist C1 eine bessere Konvertierung als C2, wenn mindestens eine der folgenden Bedingungen zutrifft:

* E besitzt einen Typ S, und es ist eine Identitätskonvertierung von S in T1, jedoch nicht von S in T2 vorhanden.
* E ist keine anonyme Funktion, und T1 ist ein besseres Konvertierungsziel als T2 (§7.5.3.5).
* E ist eine anonyme Funktion, T1 ist ein Delegattyp D1 oder ein Ausdrucksbaumstrukturtyp Expression<D1>, T2 ist ein Delegattyp D2 oder ein Ausdrucksbaumstrukturtyp Expression<D2>, und eine der folgenden Bedingungen trifft zu:
* D1 ist ein besseres Konvertierungsziel als D2
* D1 und D2 haben identische Parameterlisten, und eine der folgenden Bedingungen trifft zu:
* D1 verfügt über einen Rückgabetyp Y1, und D2 verfügt über einen Rückgabetyp Y2, für E ist im Kontext dieser Parameterliste (§7.5.2.12) ein hergeleiteter Rückgabetyp X vorhanden, und die Konvertierung von X in Y1  ist besser als die Konvertierung von X in Y2.
* E is asynchron, D1 verfügt über einen Rückgabetyp YTask<Y1>, und D2 verfügt über einen Rückgabetyp Task<Y2>, für E ist im Kontext dieser Parameterliste (§7.5.2.12) ein hergeleiteter Rückgabetyp Task<X> vorhanden, und die Konvertierung von X in Y1 ist besser als die Konvertierung von X in Y2.
* D1 besitzt einen Rückgabetyp Y, und D2 ist leer.

#### Bessere Konvertierung von Typen

Bei einer Konvertierung C1, die einen Typ S in einen Typ T1 konvertiert, und einer Konvertierung C2, die einen Typ S in einen Typ T2 konvertiert, ist C1 eine bessere Konvertierung als C2, wenn mindestens eine der folgenden Bedingungen zutrifft:

* Von S in T1 erfolgt eine Identitätskonvertierung, jedoch nicht von S in T2.
* T1 ist ein besseres Konvertierungsziel als T2 (§7.5.3.5)

#### Besseres Konvertierungsziel

Bei zwei unterschiedlichen Typen T1 und T2, ist T1 ein besseres Konvertierungsziel als T2, wenn mindestens eine der folgenden Bedingungen zutrifft:

* Es ist eine implizite Konvertierung von T1 in T2 vorhanden, und es ist keine implizite Konvertierung von T2 in T1 vorhanden.
* T1 ist ein ganzzahliger Typ mit Vorzeichen, und T2 ist ein ganzzahliger Typ ohne Vorzeichen. Genauer gesagt:
* T1 ist sbyte, und T2 ist byte, ushort, uint oder ulong
* T1 ist short, und T2 ist ushort, uint oder ulong
* T1 ist int, und T2 ist uint oder ulong
* T1 ist long und T2 ist ulong

#### Überladen in generischen Klassen

Während die Signaturen bei der Deklaration eindeutig sein müssen, können Ersetzungen von Typargumenten zu identischen Signaturen führen. In solchen Fällen wird aufgrund der Regeln bei Gleichheit während der Auflösung der Überladung oben der spezifischste Member gewählt.

In den folgenden Beispielen werden Überladungen gezeigt, die entsprechend dieser Regel gültig bzw. ungültig sind:

interface I1<T> {...}

interface I2<T> {...}

class G1<U>  
{  
 int F1(U u); // Overload resulotion for G<int>.F1  
 int F1(int i); // will pick non-generic

void F2(I1<U> a); // Valid overload  
 void F2(I2<U> a);  
}

class G2<U,V>  
{  
 void F3(U u, V v); // Valid, but overload resolution for  
 void F3(V v, U u); // G2<int,int>.F3 will fail

void F4(U u, I1<V> v); // Valid, but overload resolution for   
 void F4(I1<V> v, U u); // G2<I1<int>,int>.F4 will fail

void F5(U u1, I1<V> v2); // Valid overload  
 void F5(V v1, U u2);

void F6(ref U u); // valid overload  
 void F6(out V v);  
}

### Prüfung der dynamischen Überladungsauflösung bei der Kompilierung

Bei den meisten dynamisch gebundenen Operationen ist der Satz von möglichen Kandidaten für die Auflösung bei der Kompilierung unbekannt. In bestimmten Fällen ist der infrage kommende Satz zur bei der Kompilierung jedoch bekannt:

* Statische Methodenaufrufe mit dynamischen Argumenten
* Instanzmethodenaufrufe, bei denen der Empfänger kein dynamischer Ausdruck ist
* Indexeraufrufe, bei denen der Empfänger kein dynamischer Ausdruck ist
* Konstruktoraufrufe mit dynamischen Argumenten

In diesen Fällen wird eine beschränkte Prüfung während der Kompilierung für alle Kandidaten durchgeführt, um zu ermitteln, ob einer davon zur Laufzeit angewendet werden könnte. Diese Prüfung umfasst die folgenden Schritte:

* Partieller Typrückschluss: Jedes Typargument, das nicht direkt oder indirekt von einem Argument vom Typ dynamic abhängig ist, wird mithilfe der Regeln von §7.5.2 abgeleitet. Die verbleibenden Typargumente sind unbekannt.
* Partielle Anwendbarkeitsprüfung: Die Anwendbarkeit wird gemäß §7.5.3.1 geprüft, dabei werden jedoch Parameter ignoriert, deren Typen unbekannt sind.

Wenn kein potenzieller Parameter diesen Test besteht, tritt während der Kompilierung ein Fehler auf.

### Funktionsmemberaufruf

Dieser Abschnitt beschreibt den Vorgang, der während der Laufzeit einen bestimmten Funktionsmember aufruft. Dabei wird vorausgesetzt, dass ein Bindungsvorgang den aufzurufenden Member bereits bestimmt hat, ggf. durch Anwendung der Überladungsauflösung auf eine Reihe von potenziellen Funktionsmembern.

Um den Aufrufvorgang beschreiben zu können, werden die Funktionsmember in zwei Kategorien geteilt:

* Statische Funktionsmember. Dazu gehören Instanzkonstruktoren, statische Methoden, statische Eigenschaftenaccessoren und benutzerdefinierte Operatoren. Statische Funktionsmember sind prinzipiell nicht virtuell.
* Instanzfunktionsmember. Dazu gehören Instanzmethoden, Instanzeigenschaftenaccessoren und Indexeraccessoren. Instanzfunktionsmember sind entweder virtuell oder nicht virtuell; sie werden stets in einer bestimmten Instanz aufgerufen. Die Instanz wird durch einen Instanzausdruck berechnet und ist innerhalb des Funktionsmembers als this (§7.6.7) verfügbar.

Der Laufzeitvorgang eines Funktionsmemberaufrufs besteht aus den folgenden Schritten, wobei M der Funktionsmember und E der Instanzausdruck ist, wenn M ein Instanzmember ist:

* M ist ein statischer Funktionsmember:
* Die Auswertung der Argumentenliste erfolgt wie in §7.5.1 beschrieben.
* M wird aufgerufen.
* M als in einem value-type deklarierter Instanzfunktionsmember:
* E wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Wenn E nicht als eine Variable klassifiziert wird, dann wird eine temporäre lokale Variable des Typs E’ erstellt und der Wert von E der Variablen zugewiesen. E wird dann als ein Verweis auf diese temporäre lokale Variable erneut klassifiziert. Die temporäre Variable ist als this innerhalb von M aufrufbar, der Vorgang ist in umgekehrter Richtung nicht möglich. Daher gilt, dass die von E an this vorgenommenen Änderungen nur dann vom Aufrufer erkannt werden, wenn M eine echte Variable ist.
* Die Auswertung der Argumentenliste erfolgt wie in §7.5.1 beschrieben.
* M wird aufgerufen. Die Variable, auf die E verweist, wird zur Variablen, auf die durch this verwiesen wird.
* Wenn M ein in einem reference-type deklarierter Instanzfunktionsmember ist:
* E wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Die Auswertung der Argumentenliste erfolgt wie in §7.5.1 beschrieben.
* Wenn der Typ von E ein value-type ist, wird eine Boxingkonvertierung (§4.3.1) durchgeführt, um E in den Typ object zu konvertieren. In den folgenden Schritten wird E als Typ object behandelt. In diesem Fall kann M nur ein Member von System.Object sein.
* Der Wert von E wird auf seine Gültigkeit überprüft. Wenn der Wert von E gleich null ist, wird System.NullReferenceException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Die aufzurufende Funktionsmemberimplementierung wird bestimmt:
* Wenn der Bindungstyp von E eine Schnittstelle ist, ist der aufzurufende Funktionsmember die Implementierung von M, die vom Laufzeittyp der Instanz bereitgestellt wird, auf die von E verwiesen wird. Dieser Funktionsmember wird durch Anwenden der Schnittstellenzuordnungsregeln (§13.4.4) bestimmt, um die Implementierung von M zu ermitteln, die vom Laufzeittyp der Instanz bereitgestellt wird, auf die von E verwiesen wird.
* Wenn andernfalls M ein virtueller Funktionsmember ist, ist der aufzurufende Funktionsmember die Implementierung von M, die vom Laufzeittyp der Instanz bereitgestellt wird, auf die von E verwiesen wird. Dieser Funktionsmember wird durch Anwenden der Regeln zum Bestimmen der am meisten abgeleiteten Implementierung (§10.6.3) von M bezüglich des Laufzeittyps der Instanz ermittelt, auf die von E verwiesen wird.
* Andernfalls ist M ein nicht virtueller Funktionsmember, und der aufzurufende Funktionsmember ist M selbst.
* Die im vorigen Schritt festgelegte Funktionsmemberimplementierung wird aufgerufen. Das Objekt, auf das E verweist, wird zum Objekt, auf das durch this verwiesen wird.

#### Aufrufe in geschachtelten Instanzen

Ein in einem value-type implementierter Funktionsmember kann durch eine geschachtelte Instanz dieses value-type in folgenden Situationen aufgerufen werden:

* Wenn der Funktionsmember ein override einer vom Typ object vererbten Methode ist und durch einen Instanzausdruck des Typs object aufgerufen wird.
* Wenn der Funktionsmember eine Implementierung eines Schnittstellenfunktionsmembers ist und durch einen Instanzausdruck eines interface-type aufgerufen wird.
* Wenn der Funktionsmember durch einen Delegaten aufgerufen wird.

In diesen Situationen enthält die geschachtelte Instanz eine Variable des value-type, die zu der Variablen wird, auf die innerhalb des Funktionsmemberaufrufs durch this verwiesen wird. Das bedeutet im Einzelnen, dass beim Aufruf eines Funktionsmembers in einer geschachtelten Instanz der Funktionsmember den in der geschachtelten Instanz enthaltenen Wert ändern kann.

## Primäre Ausdrücke

Primäre Ausdrücke umfassen die einfachsten Formen von Ausdrücken.

primary-expression:   
primary-no-array-creation-expression  
array-creation-expression

primary-no-array-creation-expression:  
literal  
simple-name  
parenthesized-expression  
member-access  
invocation-expression  
element-access  
this-access  
base-access  
post-increment-expression  
post-decrement-expression  
object-creation-expression  
delegate-creation-expression  
anonymous-object-creation-expression  
typeof-expression  
 checked-expression  
unchecked-expression   
default-value-expression  
anonymous-method-expression

Primäre Ausdrücke werden in array-creation-expressions und primary-no-array-creation-expressions unterteilt. Wenn die Arrayerstellungsausdrücke nicht zusammen mit anderen einfachen Ausdrucksformen aufgelistet werden, kann potenziell uneindeutiger Code unterdrückt werden, z. B.

object o = new int[3][1];

Diese Zeile würde ansonsten interpretiert werden als

object o = (new int[3])[1];

### Literale

Eine primary-expression, die aus einem literal (§2.4.4) besteht, wird als ein Wert klassifiziert.

### Einfache Namen

Ein simple-name besteht aus einem Bezeichner, auf den optional eine Typargumentliste folgt:

simple-name:  
identifier type-argument-listopt

Ein simple-name weist entweder die Form I oder die Form I<A1, ..., AK> auf, wobei I ein einzelner Bezeichner und <A1, ..., AK> eine optionale type-argument-list ist. Wenn keine type-argument-list angegeben ist, wird K als 0 (null) betrachtet. Der simple-name wird wie folgt ausgewertet und klassifiziert:

* Wenn K 0 (null) ist, der simple-name in einem block vorkommt und der Deklarationsabschnitt lokaler Variablen (§3.3) des block (oder eines einschließenden block) eine lokale Variable, einen Parameter oder eine Konstante mit dem Namen I enthält, dann verweist der simple-name auf diese lokale Variable, diesen Parameter oder diese Konstante und ist als Variable oder Wert klassifiziert.
* Wenn K 0 (null) ist, der simple-name im Text einer generischen Methodendeklaration steht und diese Deklaration einen Typparameter mit dem Namen I enthält, dann verweist der simple-name auf diesen Typparameter.
* Andernfalls gilt Folgendes für jeden Instanztyp T (§10.3.1) ab dem Instanztyp der unmittelbar einschließenden Typdeklaration über den Instanztyp aller einschließenden Klassen- oder Strukturdeklarationen (sofern vorhanden):
* Wenn K 0 (null) ist und die Deklaration von T einen Typparameter mit dem Namen I enthält, dann verweist der simple-name auf diesen Typparameter.
* Wenn andernfalls eine Membersuche (§7.4) von I in T mit K Typargumenten zu einer Übereinstimmung führt:
* Wenn T der Instanztyp des unmittelbar einschließenden Klassen- oder Strukturtyps ist und die Suche eine oder mehrere Methoden ermittelt, ist das Ergebnis eine Methodengruppe mit dem zugeordneten Instanzausdruck this. Wenn eine Typargumentliste angegeben ist, wird diese beim Aufrufen einer generischen Methode (§7.6.5.1) verwendet.
* Wenn T andernfalls der Instanztyp des unmittelbar einschließenden Klassen- oder Strukturtyps ist, die Suche einen Instanzmember ermittelt und der Verweis im block eines Instanzkonstruktors, einer Instanzmethode oder eines Instanzaccessors steht, entspricht das Ergebnis einem Memberzugriff (§7.6.4) der Form this.I. Dies ist nur möglich, wenn K 0 (null) ist.
* Andernfalls entspricht das Ergebnis einem Memberzugriff (§7.6.4) der Form T.I oder T.I<A1, ..., AK>. In diesem Fall wird ein Bindungsfehler verursacht, wenn der simple-name auf einen Instanzmember verweist.
* Andernfalls werden für jeden Namespace N ab dem Namespace, in dem der simple-name vorkommt, über jeden einschließenden Namespace (sofern vorhanden) bis zum globalen Namespace die folgenden Schritte ausgewertet, bis eine Entität gefunden wird:
* Wenn K 0 (null) ist und I der Name eines Namespaces in N ist, gilt Folgendes:
* Wenn die Position, an der der simple-name auftritt, von einer Namespacedeklaration für N eingeschlossen ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive enthält, die den Namen I einem Namespace oder Typ zuordnet, dann ist der simple-name mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls verweist der simple-name auf den Namespace mit dem Namen I in N.
* Wenn N andernfalls einen zugreifbaren Typ mit dem Namen I und K Typparametern enthält, dann gilt Folgendes:
* Wenn K 0 (null) ist und die Position, an der der simple-name auftritt, von einer Namespacedeklaration für N eingeschlossen ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive, die den Namen I einem Namespace oder Typ zuordnet, dann ist der simple-name mehrdeutig, und es tritt ein Kompilierungsfehler auf.
* Andernfalls verweist der namespace-or-type-name auf den mit den angegebenen Typargumenten konstruierten Typ.
* Wenn andernfalls die Position, an der der simple-name auftritt, in eine Namespacedeklaration für N eingeschlossen ist, dann gilt Folgendes:
* Wenn K 0 (null) ist und die Namespacedeklaration eine extern-alias-directive oder using-alias-directive enthält, die den Namen I mit einem importierten Namespace oder Typ verknüpft, verweist der simple-name auf diesen Namespace oder Typ.
* Wenn andernfalls die über die using-namespace-directives der Namespacedeklaration importierten Namespaces genau einen Typ mit dem Namen I und K Typparametern enthalten, dann verweist der simple-name auf den mit den angegebenen Typargumenten konstruierten Typ.
* Wenn andernfalls die über die using-namespace-directives der Namespacedeklaration importierten Namespaces mehr als einen Typ mit dem Namen I und K Typparametern enthalten, dann ist der simple-name mehrdeutig, und es tritt ein Fehler auf.

Beachten Sie, dass der gesamte Schritt genau dem entsprechenden Schritt in der Verarbeitung eines namespace-or-type-name (§3.8) entspricht.

* Andernfalls ist der simple-name nicht definiert, und es tritt ein Kompilierungsfehler auf.

#### Unveränderliche Bedeutung in Blöcken

Bei jedem Vorkommen eines bestimmten Bezeichners als vollständiger simple-name (ohne Typargumentliste) in einem Ausdruck oder Deklarator innerhalb des Deklarationsabschnitts für lokale Variablen (§3.3), der dieses Vorkommnis unmittelbar einschließt, muss jedes weitere Vorkommnis desselben Bezeichners als vollständiger simple-name in einem Ausdruck oder Deklarator auf dieselbe Entität verweisen. Mit dieser Regel wird sichergestellt, dass die Bedeutung eines Namens innerhalb eines Blocks, Schalterblocks, einer for-, foreach- oder using-Anweisung oder einer anonymen Funktion gleich bleibt.

In dem Beispiel

class Test  
{  
 double x;

void F(bool b) {  
 x = 1.0;  
 if (b) {  
 int x;  
 x = 1;  
 }  
 }  
}

verursacht einen Kompilierungsfehler, weil x auf andere Entitäten innerhalb des äußeren Blocks (einschließlich des geschachtelten Blocks in der if-Anweisung) verweist. Dagegen ist das Beispiel

class Test  
{  
 double x;

void F(bool b) {  
 if (b) {  
 x = 1.0;  
 }  
 else {  
 int x;  
 x = 1;  
 }  
 }  
}

zulässig, da der Name x nicht im äußeren Block verwendet wird.

Beachten Sie, dass die Regel der unveränderlichen Bedeutung nur für einfache Namen gilt. Es ist kein Verstoß gegen diese Regel, wenn derselbe Bezeichner als einfacher Name eine andere Bedeutung hat als ein rechter Operand eines Memberzugriffs (§7.6.4). Beispiel:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Das Beispiel oben zeigt eine typische Verwendung von Feldnamen als Parameternamen in einem Instanzkonstruktor. In dem Beispiel verweisen die einfachen Namen x und y auf die Parameter, wodurch jedoch nicht verhindert wird, dass die Memberzugriffsausdrücke this.x und this.y auf die Felder zugreifen.

### Ausdrücke in Klammern

Eine parenthesized-expression besteht aus einer in Klammern eingeschlossenen expression.

parenthesized-expression:  
( expression )

Eine parenthesized-expression wird durch Auswertung der in Klammern eingeschlossenen expression verarbeitet. Bezeichnet die in Klammern eingeschlossene expression einen Namespace oder Typ, tritt ein Kompilierungsfehler auf. Ansonsten entspricht das Ergebnis der parenthesized-expression dem Ergebnis der Auswertung der enthaltenen expression.

### Memberzugriff

Ein member-access besteht aus einer primary-expression, einem predefined-type oder einem qualified-alias-member, gefolgt vom Token ".", einem identifier und optional einer type-argument-list.

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
qualified-alias-member . identifier type-argument-listopt

predefined-type: one of  
bool byte char decimal double float int long  
object sbyte short string uint ulong ushort

Die Produktion von qualified-alias-member ist in §9.7 definiert.

Ein member-access weist entweder die Form E.I oder die Form E.I<A1, ..., AK> auf, wobei E eine primary-expression, I ein einzelner Bezeichner und <A1, ..., AK> eine optionale type-argument-list ist. Wenn keine type-argument-list angegeben ist, wird K als 0 (null) betrachtet.

Ein member-access mit einer primary-expression vom Typ dynamic wird dynamisch gebunden (§7.2.2). In diesem Fall klassifiziert der Compiler den Memberzugriff als Eigenschaftenzugriff vom Typ dynamic. Die folgenden Regeln zur Ermittlung der Bedeutung des member-access werden dann zur Laufzeit angewendet, wobei der Laufzeittyp anstelle des Kompilierungstyps der primary-expression verwendet wird. Wenn diese Laufzeitklassifikation zu einer Methodengruppe führt, muss der Memberzugriff die primary-expression einer invocation-expression sein.

Der member-access wird wie folgt ausgewertet und klassifiziert:

* Wenn K 0 (null) ist, E ein Namespace ist und E einen geschachtelten Namespace mit dem Namen I enthält, dann ist das Ergebnis dieser Namespace.
* Wenn andernfalls E ein Namespace ist und E einen zugreifbaren Typ mit dem Namen I und K Typparametern enthält, dann ist das Ergebnis dieser mit den angegebenen Typargumenten konstruierte Typ.
* Wenn E ein predefined-type oder eine als Typ klassifizierte primary-expression ist, E kein Typparameter ist und eine Membersuche (§7.4) nach I in E mit K Typparametern eine Übereinstimmung ergibt, dann wird E.I wie folgt ausgewertet und klassifiziert:
* Wenn I einen Typ identifiziert, dann ist das Ergebnis dieser mit den angegebenen Typargumenten konstruierte Typ.
* Wenn I eine oder mehrere Methoden erkennt, ist das Ergebnis eine Methodengruppe ohne einen zugehörigen Instanzausdruck. Wenn eine Typargumentliste angegeben ist, wird diese beim Aufrufen einer generischen Methode (§7.6.5.1) verwendet.
* Wenn I eine static-Eigenschaft identifiziert, ist das Ergebnis ein Eigenschaftenzugriff ohne einen zugehörigen Instanzausdruck.
* Wenn I ein static-Feld identifiziert, dann gilt:
* Wenn das Feld vom Typ readonly ist und der Verweis außerhalb des statischen Konstruktors der Klasse oder Struktur auftritt, in der das Feld deklariert ist, wird als Ergebnis ein Wert, und zwar der Wert des statischen Felds I in E, zurückgegeben.
* Andernfalls ist das Ergebnis eine Variable, und zwar das statische Feld I in E.
* Wenn I ein static-Ereignis identifiziert, dann gilt:
* Wenn der Verweis innerhalb der Klasse oder Struktur auftritt, in der das Ereignis deklariert ist und das Ereignis ohne event-accessor-declarations (§10.8) deklariert wurde, wird E.I so verarbeitet, als ob I ein statisches Feld wäre.
* Andernfalls ist das Ergebnis ein Ereigniszugriff ohne einen zugehörigen Instanzausdruck.
* Wenn I eine Konstante identifiziert, wird als Ergebnis ein Wert, und zwar der Wert dieser Konstante, zurückgegeben.
* Wenn I einen Enumerationsmember identifiziert, wird als Ergebnis ein Wert, und zwar der Wert dieses Enumerationsmembers, zurückgegeben.
* In allen anderen Fällen ist E.I ein ungültiger Memberverweis, und es wird ein Kompilierungsfehler ausgelöst.
* Wenn E ein Eigenschaftenzugriff, ein Indexerzugriff, eine Variable oder ein Wert vom Typ T ist und eine Membersuche (§7.4) nach I in T mit K Typargumenten eine Übereinstimmung ergibt, wird E.I wie folgt ausgewertet und klassifiziert:
* Wenn E eine Eigenschaft oder ein Indexerzugriff ist, wird der Wert der Eigenschaft oder des Indexerzugriffs abgerufen (§7.1.1) und E als Wert neu klassifiziert.
* Wenn I eine oder mehrere Methoden erkennt, ist das Ergebnis eine Methodengruppe mit einem zugehörigen Instanzausdruck von E. Wenn eine Typargumentliste angegeben ist, wird diese beim Aufrufen einer generischen Methode (§7.6.5.1) verwendet.
* Wenn I eine Instanzeigenschaft identifiziert, ist das Ergebnis ein Eigenschaftenzugriff mit einem zugehörigen Instanzausdruck E.
* Wenn T ein class-type ist und I ein Instanzfeld dieses class-type identifiziert, dann gilt:
* Wenn der Wert von E gleich null ist, wird eine System.NullReferenceException ausgelöst.
* Wenn das Feld jedoch readonly ist und der Verweis außerhalb eines Instanzkonstruktors der Klasse auftritt, in der das Feld deklariert ist, wird als Ergebnis ein Wert zurückgegeben, und zwar der Wert des Felds I in dem Objekt, auf das durch E verwiesen wird.
* Andernfalls ist das Ergebnis eine Variable, und zwar das Feld I in dem Objekt, auf das E verweist.
* Wenn T ein struct-type ist und I ein Instanzfeld dieses struct-type identifiziert, dann gilt:
* Wenn E ein Wert oder das Feld readonly ist und der Verweis außerhalb eines Instanzkonstruktors der Struktur auftritt, in der das Feld deklariert ist, wird als Ergebnis ein Wert zurückgegeben, und zwar der Wert des Felds I in der durch E festgelegten Strukturinstanz.
* Andernfalls ist das Ergebnis eine Variable, und zwar das Feld I in der durch E festgelegten Strukturinstanz.
* Wenn I ein Instanzereignis identifiziert, dann gilt:
* Wenn der Verweis innerhalb einer Klasse oder Struktur auftritt, in der das Ereignis deklariert ist, das Ereignis ohne event-accessor-declarations (§10.8) deklariert wurde und der Verweis nicht als linke Seite eines +=- oder -=-Operators auftritt, wird E.I genauso verarbeitet, als ob I ein Instanzfeld wäre.
* Andernfalls ist das Ergebnis ein Ereigniszugriff mit einem zugehörigen Instanzausdruck von E.
* Ansonsten wird versucht, E.I als Aufruf einer Erweiterungsmethode (§7.6.5.2) zu verarbeiten. Schlägt dies fehl, ist E.I ein ungültiger Memberverweis, und es tritt ein Bindungsfehler auf.

#### Identische einfache Namen und Typnamen

In einem Memberzugriff der Form E.I gilt: Wenn E ein einzelner Bezeichner ist und die Bedeutung von E als simple-name (§7.6.2) eine Konstante, ein Feld, eine Eigenschaft, eine lokale Variable oder ein Parameter desselben Typs ist, wie die Bedeutung von E als type-name (§3.8), sind beide Bedeutungen von E zulässig. Beide Bedeutungen von E.I sind stets eindeutig, weil I notwendigerweise in beiden Fällen ein Member des Typs E sein muss. Die Regel ermöglicht anders ausgedrückt den Zugriff auf die statischen Member und geschachtelten Typen von E, wobei in allen anderen Fällen ein Kompilierungsfehler aufgetreten wäre. Beispiel:

struct Color  
{  
 public static readonly Color White = new Color(...);  
 public static readonly Color Black = new Color(...);

public Color Complement() {...}  
}

class A  
{  
 public Color Color; // Field Color of type Color

void F() {  
 Color = Color.Black; // References Color.Black static member  
 Color = Color.Complement(); // Invokes Complement() on Color field  
 }

static void G() {  
 Color c = Color.White; // References Color.White static member  
 }  
}

Innerhalb der A-Klasse sind die Color-Bezeichner, die auf den Color-Typ verweisen, unterstrichen, die Verweise auf das Color-Feld hingegen sind nicht unterstrichen.

#### Grammatikalische Mehrdeutigkeiten

Die Produktionen für simple-name (§7.6.2) und member-access (§7.6.4) können zu Mehrdeutigkeiten in der Grammatik von Ausdrücken führen. Beispielsweise kann die Anweisung:

F(G<A,B>(7));

als Aufruf von F mit den beiden Argumenten G < A und B > (7) aufgefasst werden. Alternativ kann sie als Aufruf von F mit einem Argument, nämlich dem Aufruf der generischen Methode G mit zwei Typargumenten und einem normalen Argument, aufgefasst werden.

Wenn eine Folge von Token (im Kontext) als simple-name (§7.6.2), member-access (§7.6.4) oder pointer-member-access (§18.5.2), der mit einer type-argument-list endet (§4.4.1), analysiert werden kann, wird das auf das schließende Token > unmittelbar folgende Token untersucht. Wenn dies eines von

( ) ] } : ; , . ? == != | ^

ist, wird die type-argument-list als Teil des simple-name, member-access oder pointer-member-access beibehalten, und alle anderen möglichen Analysen der Tokenfolge werden verworfen. Andernfalls wird die type-argument-list nicht als Teil des simple-name, member-access oder pointer-member-access betrachtet, selbst wenn keine andere Analyse der Tokenfolge möglich ist. Beachten Sie, dass diese Regeln beim Analysieren einer type-argument-list in einem namespace-or-type-name (§3.8) nicht angewendet werden. Die Anweisung

F(G<A,B>(7));

wird entsprechend dieser Regel als Aufruf von F mit einem Argument, nämlich dem Aufruf der generischen Methode G mit zwei Typargumenten und einem normalen Argument, interpretiert. Die Anweisungen

F(G < A, B > 7);  
F(G < A, B >> 7);

werden jeweils als Aufruf von F mit zwei Argumenten interpretiert. Die Anweisung

x = F < A > +y;

wird als Kleiner-als-Operator, Größer-als-Operator und unärer Plus-Operator interpretiert, als lautete die Anweisung x = (F < A) > (+y), und nicht als simple-name mit einer type-argument-list, gefolgt von einem binären Plus-Operator. In der Anweisung

x = y is C<T> + z;

werden die Token C<T> als namespace-or-type-name mit einer type-argument-list interpretiert.

### Aufrufausdrücke

invocation-expression werden zum Aufrufen von Methoden verwendet.

invocation-expression:  
primary-expression ( argument-listopt )

Eine invocation-expression wird dynamisch gebunden (§7.2.2), wenn mindestens eine der folgenden Bedingungen zutrifft:

* Die primary-expression weist den Kompilierungstyp dynamic auf.
* Mindestens ein Argument der optionalen argument-list weist den Kompilierungstyp dynamic auf, und die primary-expression weist keinen Delegattyp auf.

In diesem Fall klassifiziert der Compiler die invocation-expression als Wert vom Typ dynamic. Die folgenden Regeln zur Ermittlung der Bedeutung der invocation-expression werden dann zur Laufzeit angewendet, wobei der Laufzeittyp anstelle des Kompilierungstyps der primary-expression und der Argumente verwendet wird, die den Kompilierungstyp dynamic aufweisen. Wenn die primary-expression nicht den Kompilierungstyp dynamic aufweist, durchläuft der Methodenaufruf eine beschränkte Prüfung während der Kompilierung, wie in §7.5.4 beschrieben.

Die primary-expression einer invocation-expression muss eine Methodengruppe oder ein Wert eines delegate-type sein. Wenn die primary-expression einer Methodengruppe entspricht, ist die invocation-expression ein Methodenaufruf (§7.6.5.1). Wenn die primary-expression einem Wert eines delegate-type entspricht, ist die invocation-expression ein Delegataufruf (§7.6.5.3). Wenn die primary-expression weder eine Methodengruppe noch ein Wert eines delegate-type ist, tritt ein Bindungsfehler auf.

Die optionale argument-list (§7.5.1) enthält für die Parameter der Methode Werte oder Variablenverweise.

Das Auswertungsergebnis einer invocation-expression wird folgendermaßen klassifiziert:

* Wenn die invocation-expression eine Methode oder einen Delegaten aufruft, die void zurückgeben, wird kein Ergebnis zurückgegeben. Ein als nothing klassifizierter Ausdruck ist nur im Kontext einer statement-expression (§8.6) oder als Text einer lambda-expression (§7.15) zulässig. Andernfalls wird ein Bindungsfehler ausgelöst.
* In allen anderen Fällen wird als Ergebnis ein Wert des Typs zurückgegeben, der von der Methode bzw. dem Delegaten ermittelt wurde.

#### Methodenaufrufe

Für einen Methodenaufruf muss die primary-expression der invocation-expression einer Methodengruppe entsprechen. Die Methodengruppe erkennt die aufzurufende Methode bzw. die Gruppe überladener Methoden, aus denen die aufzurufende Methode ausgewählt wird. Im letzteren Fall basiert die Auswahl der aufzurufenden Methode auf dem Kontext, der durch die Argumenttypen der argument-list vorgegeben wird.

Die Verarbeitung eines Methodenaufrufs zur Bindungszeit der Form M(A), wobei M eine Methodengruppe (ggf. mit einer type-argument-list) und A eine optionale argument-list ist, umfasst die folgenden Schritte:

* Erstellen der potenziellen Methoden für den Methodenaufruf. Für jede Methode F, die der Methodengruppe M zugeordnet ist, gilt:
* Wenn F nicht generisch ist, kommt F unter folgenden Bedingungen infrage:
* M besitzt keine Typargumentliste, und
* F ist in Bezug auf A anwendbar (§7.5.3.1).
* Wenn F generisch ist und M keine Typargumentliste besitzt, kommt F unter folgenden Bedingungen infrage:
* Der Typrückschluss (§7.5.2) ist erfolgreich und leitet eine Liste von Typargumenten für den Aufruf ab, und:
* Nach dem Ersetzen der abgeleiteten Typargumente durch die entsprechenden Methodentypparameter erfüllen alle konstruierten Typen in der Parameterliste von F ihre Einschränkungen (§4.4.4), und die Parameterliste von F ist in Bezug auf A anwendbar (§7.5.3.1).
* Wenn F generisch ist und M eine Typargumentliste besitzt, kommt F unter folgenden Bedingungen infrage:
* F besitzt die gleiche Anzahl von Methodentypparametern, wie in der Typargumentliste übergeben wurden, und:
* Nach dem Ersetzen der abgeleiteten Typargumente durch die entsprechenden Methodentypparameter erfüllen alle konstruierten Typen in der Parameterliste von F ihre Einschränkungen (§4.4.4), und die Parameterliste von F ist in Bezug auf A anwendbar (§7.5.3.1).
* Die Menge der infrage kommenden Methoden wird auf die Methoden der am stärksten abgeleiteten Typen beschränkt: Für jede Methode C.F in der Menge, wobei C der Typ ist, in dem die Methode F deklariert ist, werden alle in einem Basistyp von C deklarierten Methoden aus der Menge entfernt. Wenn C außerdem ein Klassentyp ist, der nicht object ist, werden alle in einem Schnittstellentyp deklarierten Methoden aus der Menge entfernt. (Diese zuletzt genannte Regel hat nur dann eine Auswirkung, wenn die Methodengruppe das Ergebnis einer Membersuche nach einem Typparameter ist, der eine effektive Basisklasse, die nicht object ist, und eine nicht leere Schnittstellengruppe besitzt.)
* Wenn der Ergebnissatz von infrage kommenden Methoden leer ist, wird die weitere Verarbeitung in den nächsten Schritte abgebrochen und stattdessen versucht, den Aufruf als den Aufruf einer Erweiterungsmethode (§7.6.5.2) zu verarbeiten. Wenn dies fehlschlägt, sind keine anwendbaren Methoden vorhanden, und es tritt ein Bindungsfehler auf.
* Die beste aus den vorhandenen potenziellen Methoden wird anhand der in §7.5.3 beschriebenen Überladungsauflösungsregeln ermittelt. Wenn es nicht möglich ist, eine beste Methode zu ermitteln, ist der Methodenaufruf nicht eindeutig, und es tritt ein Bindungsfehler auf. Bei der Durchführung der Überladungsauflösung werden die Parameter einer generischen Methode berücksichtigt, nachdem die entsprechenden Methodentypparameter durch die (angegebenen oder abgeleiteten) Typargumente ersetzt wurden.
* Es wird die endgültige Validierung der ausgewählten besten Methode ausgeführt:
* Die Methode wird im Kontext der Methodengruppe überprüft: Wenn die beste Methode eine statische Methode ist, muss die Methodengruppe das Ergebnis eines simple-name oder eines member-access über einen Typ sein. Wenn die beste Methode eine Instanzmethode ist, muss die Methodengruppe aus einem simple-name oder einem member-access durch eine Variable bzw. einen Wert oder aus einem base-access resultieren. Wenn keine dieser Bedingungen erfüllt ist, tritt ein Bindungsfehler auf.
* Wenn die beste Methode eine generische Methode ist, werden die (angegebenen oder abgeleiteten) Typargumente auf die für die generische Methode deklarierten Einschränkungen (§4.4.4) hin überprüft. Wenn ein beliebiges Typargument die entsprechenden Einschränkungen für den Typparameter nicht erfüllt, tritt ein Bindungsfehler auf.

Nachdem eine Methode bei der Bindung anhand der oben beschriebenen Schritte ausgewählt und überprüft wurde, wird der eigentliche Laufzeitaufruf entsprechend den Regeln des in §7.5.4 beschriebenen Funktionsmemberaufrufs verarbeitet.

Die oben beschriebenen Auflösungsregeln wirken sich intuitiv wie folgt aus: Um eine bestimmte durch einen Methodenaufruf aufgerufene Methode zu lokalisieren, beginnen Sie mit dem durch den Methodenaufruf vorgegebenen Typ und durchlaufen die Vererbungskette, bis mindestens eine anwendbare, verfügbare Deklaration für Nicht-Überschreibungsmethoden gefunden wird. Führen Sie dann den Typrückschluss und die Überladungsauflösung für die in diesem Typ deklarierten anwendbaren, verfügbaren Nicht-Überschreibungsmethoden aus, und rufen Sie die so ausgewählte Methode auf. Wenn keine Methode gefunden werden kann, verarbeiten Sie stattdessen den Aufruf als den Aufruf einer Erweiterungsmethode.

#### Erweiterungsmethodenaufrufe

Für einen Methodenaufruf (§7.5.5.1) einer der folgenden Formen

expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )

gilt: Wenn bei der normalen Verarbeitung des Aufrufs keine anwendbaren Methoden gefunden werden, wird versucht, das Konstrukt als einen Erweiterungsmethodenaufruf zu verarbeiten. Wenn expr oder eines der args den Kompilierungstyp dynamic aufweist, werden keine Erweiterungsmethoden angewendet.

Ziel ist das Ermitteln des besten type-name C, damit der zugehörige Aufruf der statischen Methode durchgeführt werden kann:

C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )

Eine Erweiterungsmethode Ci.Mj ist freigegeben, wenn:

* Ci eine nicht generische, nicht verschachtelte Klasse ist.
* Der Name von Mj identifier ist.
* auf Mj zugegriffen werden kann und Mj angewendet werden kann, wenn sie als statische Methode auf die Argumente angewendet wird, wie oben dargestellt.
* Eine implizite Identitäts-, Verweis- oder Boxingkonvertierung von expr in den Typ des ersten Parameters von Mj. vorhanden ist.

Die Suche nach C läuft folgendermaßen ab:

* Beginnend bei der nächsten einschließenden Namespacedeklaration wird nacheinander in den weiteren einschließenden Namespacedeklarationen bis zur enthaltenden Kompilationseinheit versucht, einen infrage kommenden Satz von Erweiterungsmethoden zu finden:
* Wenn der angegebene Namespace oder die Kompilationseinheit direkt nicht generische Typdeklarationen Ci mit den möglichen Erweiterungsmethoden Mj enthält, kommt dieser Satz von Erweiterungsmethoden infrage.
* Wenn mithilfe von Namespacedirektiven importierte Namespaces im angegebenen Namespace oder der angegebenen Kompilationseinheit direkt nicht generische Typdeklarationen Ci mit den möglichen Erweiterungsmethoden Mj enthält, kommt dieser Satz von Erweiterungsmethoden infrage.
* Wenn in keiner der einschließenden Namespacedeklarationen oder der Kompilationseinheit ein infrage kommender Satz gefunden wird, tritt während der Kompilierung ein Fehler auf.
* Andernfalls wird wie in (§7.5.3) beschrieben eine Überladungsauflösung auf den infrage kommenden Satz angewendet. Wird keine einzelne optimale Methode gefunden, tritt während der Kompilierung ein Fehler auf.
* C ist der Typ, in dem die optimale Methode als Erweiterungsmethode deklariert ist.

Mit C als Ziel wird dann der Methodenaufruf als Aufruf einer statischen Methode (§7.5.4) durchgeführt.

Die vorherigen Regeln sagen aus, dass Instanzmethoden eine höhere Priorität als Erweiterungsmethoden haben, dass Erweiterungsmethoden, die innerhalb der inneren Namespacedeklarationen stehen, eine höhere Priorität als Erweiterungsmethoden in äußeren Namespacedeklarationen haben und dass Erweiterungsmethoden, die direkt in einem Namespace deklariert sind, eine höhere Priorität als Erweiterungsmethoden haben, die mithilfe einer Namespacedirektive in diesen Namespace importiert wurden. Beispiel:

public static class E  
{  
 public static void F(this object obj, int i) { }

public static void F(this object obj, string s) { }  
}

class A { }

class B  
{  
 public void F(int i) { }  
}

class C  
{  
 public void F(object obj) { }  
}

class X  
{  
 static void Test(A a, B b, C c) {  
 a.F(1); // E.F(object, int)  
 a.F("hello"); // E.F(object, string)

b.F(1); // B.F(int)  
 b.F("hello"); // E.F(object, string)

c.F(1); // C.F(object)  
 c.F("hello"); // C.F(object)  
 }  
}

In diesem Beispiel hat die Methode von B eine höhere Priorität als die erste Erweiterungsmethode, und die Methode von C hat eine höhere Priorität als die anderen beiden Erweiterungsmethoden.

public static class C  
{  
 public static void F(this int i) { Console.WriteLine("C.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("C.G({0})", i); }  
 public static void H(this int i) { Console.WriteLine("C.H({0})", i); }  
}

namespace N1  
{  
 public static class D  
 {  
 public static void F(this int i) { Console.WriteLine("D.F({0})", i); }  
 public static void G(this int i) { Console.WriteLine("D.G({0})", i); }  
 }  
}

namespace N2  
{  
 using N1;

public static class E  
 {  
 public static void F(this int i) { Console.WriteLine("E.F({0})", i); }  
 }

class Test  
 {  
 static void Main(string[] args)  
 {  
 1.F();  
 2.G();  
 3.H();  
 }  
 }  
}

Die Ausgabe dieses Beispiels lautet:

E.F(1)  
D.G(2)  
C.H(3)

D.G hat eine höhere Priorität als C.G, und E.F hat eine höhere Priorität als D.F und C.F.

#### Delegataufrufe

Für einen Delegataufruf muss die primary-expression der invocation-expression einem Wert eines delegate-type entsprechen. Wenn der delegate-type ein Funktionsmember mit derselben Parameterliste wie der delegate-type ist, muss der delegate-type hinsichtlich der argument-list der invocation-expression (§7.5.3.1) anwendbar sein.

Die Laufzeitverarbeitung eines Delegataufrufs der Form D(A), wobei D einer primary-expression eines delegate-type und A einer optionalen argument-list entspricht, setzt sich aus den folgenden Schritten zusammen:

* D wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Der Wert von D wird auf seine Gültigkeit überprüft. Wenn der Wert von D null ist, wird eine System.NullReferenceException ausgelöst und keine weiteren Schritte ausgeführt.
* In allen anderen Fällen ist D ein Verweis auf eine Delegatinstanz. Funktionsmemberaufrufe (§7.5.4) werden für jede einzelne aufrufbare Entität in der Aufrufliste des Delegattyps ausgeführt. Bei aufrufbaren Entitäten, die aus einer Instanz und einer Instanzmethode bestehen, wird für den Aufruf die Instanz verwendet, die in der aufrufbaren Entität enthalten ist.

### Elementzugriff

Ein element-access besteht aus einer primary-no-array-creation-expression, gefolgt von einem "["-Token, einer argument-list und einem "]"-Token. Die argument-list besteht aus einem oder mehreren arguments, die durch Trennzeichen getrennt sind.

element-access:  
primary-no-array-creation-expression [ argument-list ]

Die argument-list eines element-access darf keine ref- oder out-Argumente enthalten.

Ein element-access wird dynamisch gebunden (§7.2.2), wenn mindestens eine der folgenden Bedingungen zutrifft:

* Die primary-no-array-creation-expression weist den Kompilierungstyp dynamic auf.
* Mindestens ein Ausdruck der argument-list weist den Kompilierungstyp dynamic auf, und die primary-no-array-creation-expression weist keinen Arraytyp auf.

In diesem Fall klassifiziert der Compiler den element-access als Wert vom Typ dynamic. Die folgenden Regeln zur Ermittlung der Bedeutung des element-access werden dann zur Laufzeit angewendet, wobei der Laufzeittyp anstelle des Kompilierungstyps der Ausdrücke primary-no-array-creation-expression und argument-list verwendet wird, die den Kompilierungstyp dynamic aufweisen. Wenn die primary-no-array-creation-expression nicht den Kompilierungstyp dynamic aufweist, durchläuft der Elementzugriff eine beschränkte Prüfung während der Kompilierung, wie in §7.5.4 beschrieben.

Wenn die primary-no-array-creation-expression eines element-access ein Wert eines array-type ist, entspricht der element-access einem Arrayzugriff (§7.6.6.1). Andernfalls muss die primary-no-array-creation-expression eine Variable oder ein Wert eines Klassen-, Struktur- oder eines Schnittstellentyps sein, der über einen oder mehrere Indexmember verfügt. In diesem Fall entspricht der element-access einem Indexerzugriff (§7.6.6.2).

#### Arrayzugriff

Für einen Arrayzugriff muss der primary-no-array-creation-expression des element-access einem Wert eines array-type entsprechen. Die argument-list eines Arrayzugriffs darf außerdem keine benannten Argumente enthalten. Die Anzahl von Ausdrücken in der argument-list muss mit dem Rang des array-type übereinstimmen, und alle Ausdrücke müssen den Typ int, uint, long, ulong aufweisen oder implizit in einen oder mehrere dieser Typen konvertiert werden können.

Das Ergebnis der Überprüfung eines Arrayzugriffs ist eine Variable des Elementtyps des Arrays, und zwar des Arrayelements, das durch den bzw. die Werte der Ausdrücke in der argument-list festgelegt wurde.

Die Laufzeitverarbeitung eines Arrayzugriffs der Form P[A], wobei P eines primary-no-array-creation-expression einer array-type und A einer argument-list entspricht, setzt sich aus den folgenden Schritten zusammen:

* P wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Die Indexausdrücke der argument-list werden in der Reihenfolge von links nach rechts ausgewertet. Nach der Auswertung der einzelnen Ausdrücke wird eine implizite Konvertierung (§6.1) in einen der folgenden Typen ausgeführt: int, uint, long, ulong. Dabei wird der erste Typ in der Liste ausgewählt, für den eine implizite Konvertierung vorhanden ist. Wenn beispielsweise der Indexausdruck vom Typ short ist, wird eine implizite Konvertierung in int ausgeführt, da implizite Konvertierungen von short in int und von short in long möglich sind. Wenn die Auswertung eines Indexausdrucks oder einer nachfolgenden impliziten Konvertierung eine Ausnahme verursacht, werden weder weitere Indexausdrücke ausgewertet noch weitere Schritte ausgeführt.
* Der Wert von P wird auf seine Gültigkeit überprüft. Wenn der Wert von P null ist, wird eine System.NullReferenceException ausgelöst und keine weiteren Schritte ausgeführt.
* Der Wert der Ausdrücke in der argument-list wird mit den tatsächlichen Grenzen aller Dimensionen der Arrayinstanz verglichen, auf die von P verwiesen wird. Wenn sich mindestens ein Wert außerhalb des gültigen Bereichs befindet, wird eine System.IndexOutOfRangeException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Die Position des vom Indexausdruck festgelegten Arrayelements wird berechnet und als Ergebnis des Arrayzugriffs ausgegeben.

#### Indexerzugriff

Für einen Indexerzugriff muss die primary-no-array-creation-expression des element-access eine Variable oder ein Wert des Klassen-, Struktur- oder Schnittstellentyps sein. Dieser Typ muss seinerseits einen oder mehrere Indexer implementieren, die hinsichtlich der argument-list des element-access anwendbar sind.

Die Bindungsverarbeitung eines Indexerzugriffs der Form P[A], wobei P einer primary-no-array-creation-expression eines Klassen-, Struktur- oder Schnittstellentyps T und A einer argument-list entspricht, setzt sich aus den folgenden Schritten zusammen:

* Die von T bereitgestellte Gruppe von Indexern wird erstellt. Die Gruppe besteht aus allen in T oder einem Basistyp von T deklarierten Indexern, bei denen es sich nicht um override-Deklarationen handelt und auf die im aktuellen Kontext (§3.5) zugegriffen werden kann.
* Die Gruppe der Indexer wird auf diejenigen reduziert, die anwendbar und nicht von anderen Indexern verdeckt sind. Die folgenden Regeln gelten für alle S.I-Indexer in der Gruppe, wobei S der Typ ist, in dem der Indexer I deklariert wird:
* Wenn I in Bezug auf A (§7.5.3.1) nicht anwendbar ist, wird I aus der Gruppe entfernt.
* Wenn I in Bezug auf A (§7.5.3.1) anwendbar ist, werden alle in einem Basistyp von S deklarierten Indexer aus der Gruppe entfernt.
* Wenn I in Bezug auf A (§7.5.3.1) anwendbar ist und S ein Klassentyp ist, der nicht object ist, werden alle in einer Schnittstelle deklarierten Indexer aus der Gruppe entfernt.
* Wenn die resultierende Gruppe der potenziellen Indexer leer ist, sind keine anwendbaren Indexer vorhanden, und es tritt ein Bindungsfehler auf.
* Der beste Indexer aus der Gruppe potenzieller Indexer wird entsprechend den in §7.5.3 beschriebenen Überladungsauflösungsregeln ermittelt. Wenn es nicht möglich ist, einen besten Indexer zu ermitteln, ist der Indexerzugriff nicht eindeutig, und es tritt ein Bindungsfehler auf.
* Die Indexausdrücke der argument-list werden in der Reihenfolge von links nach rechts ausgewertet. Das Auswertungsergebnis des Indexerzugriffs ist ein als Indexerzugriff klassifizierter Ausdruck. Der Indexzugriffsausdruck verweist auf den im vorigen Schritt ermittelten Indexer und verfügt über einen zugehörigen Instanzausdruck von P sowie eine zugehörige Argumentenliste von A.

Je nachdem, in welchem Kontext der Indexerzugriff verwendet wird, verursacht er einen Aufruf des get-accessor oder des set-accessor des Indexers. Wenn der Indexerzugriff das Ziel einer Zuweisung ist, wird der set-accessor aufgerufen, um einen neuen Wert zuzuweisen (§7.17.1). In allen anderen Fällen wird der get-accessor aufgerufen, um den aktuellen Wert abzurufen (§7.1.1).

### This-Zugriff

Ein this-access besteht aus dem reservierten Wort this.

this-access:  
this

Ein this-access ist nur in dem block eines Instanzkonstruktors, einer Instanzmethode oder eines Instanzaccessors zulässig. Der this-Zugriff hat eine der folgenden Bedeutungen:

* Wenn this in einer primary-expression innerhalb eines Instanzkonstruktors einer Klasse verwendet wird, wird es als Wert klassifiziert. Der Werttyp entspricht dem Instanztyp (§10.3.1) der Klasse, in der this verwendet wird, und der Wert ist ein Verweis auf das Objekt, welches erstellt wird.
* Wenn this in einer primary-expression innerhalb einer Instanzmethode oder eines Instanzaccessors einer Klasse verwendet wird, wird es als Wert klassifiziert. Der Werttyp entspricht dem Instanztyp (§10.3.1) der Klasse, in der this verwendet wird, und der Wert ist ein Verweis auf das Objekt, für das die Methode oder der Accessor aufgerufen wurde.
* Wenn this in einer primary-expression innerhalb eines Instanzkonstruktors einer Struktur verwendet wird, wird es als Variable klassifiziert. Der Variablentyp entspricht dem Instanztyp (§10.3.1) der Struktur, in der this verwendet wird, und die Variable verweist auf die Struktur, welche erstellt wird. Die this-Variable eines Instanzkonstruktors einer Struktur verhält sich genauso wie ein out-Parameter des Strukturtyps, d. h., die Variable muss in allen Ausführungspfaden des Instanzkonstruktors definitiv zugewiesen sein.
* Wenn this in einer primary-expression innerhalb einer Instanzmethode oder eines Instanzaccessors einer Struktur verwendet wird, wird es als Variable klassifiziert. Der Variablentyp entspricht dem Instanztyp (§10.3.1) der die Struktur, in der this verwendet wird.
* Wenn die Methode oder der Accessor kein Iterator (§10.14) ist, stellt die this-Variable die Struktur dar, für die die Methode oder der Accessor aufgerufen wurde und verhält sich genauso wie ein ref-Parameter des Strukturtyps.
* Wenn die Methode oder der Accessor ein Iterator ist, stellt die this-Variable eine Kopie der Struktur dar, für die die Methode oder der Accessor aufgerufen wurde und verhält sich genauso wie ein value-Parameter des Strukturtyps.

Die Verwendung von this in einer primary-expression in einem anderen als den oben aufgeführten Kontexten verursacht einen Kompilierungsfehler. Insbesondere darf in einer statischen Methode, einem statischen Eigenschaftenaccessor oder der variable-initializer einer Felddeklaration nicht auf this verwiesen werden.

### Basiszugriff

Ein base-access besteht aus dem reservierten Wort base, gefolgt von einem "."-Token und einem Bezeichner oder einer in eckige Klammern eingeschlossenen argument-list:

base-access:  
base . identifier  
base [ argument-list ]

Ein base-access wird verwendet, um Basisklassenmember aufzurufen, die in der aktuellen Klasse oder Struktur von ähnlich bezeichneten Membern verdeckt sind. Ein base-access ist nur in dem block eines Instanzkonstruktors, einer Instanzmethode oder eines Instanzaccessors zulässig. Wenn base.I in einer Klasse oder Struktur auftritt, muss I einen Member der Basisklasse dieser Klasse oder Struktur bezeichnen. Analog dazu muss, wenn base[E] in einer Klasse auftritt, ein anwendbarer Indexer in der Basisklasse vorhanden sein.

Während der Bindung werden base-access-Ausdrücke der Form base.I und base[E] so ausgewertet, als hätten sie die Form ((B)this).I und ((B)this)[E]. Dabei entspricht B der Basisklasse in der Klasse oder Struktur, in der das Konstrukt auftritt. Somit entsprechen base.I und base[E] den Ausdrücken this.I und this[E], wobei this als Instanz der Basisklasse angesehen wird.

Wenn ein base-access auf einen virtuellen Funktionsmember verweist (eine Methode, Eigenschaft oder ein Indexer), ändert sich die Festlegung des zur Laufzeit aufzurufenden Funktionsmembers (§7.5.4). Der aufzurufende Funktionsmember wird ermittelt, indem die am weitesten abgeleitete Implementierung (§10.6.3) des Funktionsmembers in Bezug auf B (an Stelle des Laufzeittyps von this, wie sonst bei einem Nicht-Basiszugriff üblich) bestimmt wird. Somit kann innerhalb eines override eines virtual-Funktionsmembers mittels eines base-access die vererbte Implementierung des Funktionsmembers aufgerufen werden. Wenn der Funktionsmember, auf den ein base-access verweist, abstrakt ist, tritt ein Bindungsfehler auf.

### Postfix-Inkrement- und -Dekrementoperatoren

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

Der Operand einer Postfix-Inkrementoperation oder einer Postfix-Dekrementoperation muss ein Ausdruck sein, der als Variable, Eigenschaftenzugriff oder Indexerzugriff klassifiziert ist. Das Ergebnis der Operation ist ein Wert vom Typ des Operanden.

Wenn die primary-expression den Kompilierungstyp dynamic aufweist, wird der Operator dynamisch gebunden (§7.2.2), die post-increment-expression oder post-decrement-expression weist den Typ dynamic auf und die folgenden Regeln werden zur Laufzeit unter Verwendung des Laufzeittyps derb primary-expression angewendet.

Wenn der Operand einer Postfix-Inkrementoperation oder Postfix-Dekrementoperation eine Eigenschaft oder ein Indexzugriff ist, muss die Eigenschaft oder der Indexer über einen get-Accessor und einen set-Accessor verfügen. Ist dies nicht der Fall, tritt während der Bindung ein Fehler auf.

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die vordefinierten Operatoren ++ und -- bestehen für die folgenden Typen: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal und alle Enumerationstypen. Die vordefinierten Operatoren ++ geben den Wert zurück, der entsteht, wenn zum Operanden 1 addiert wird. Die vordefinierten Operatoren -- geben den Wert zurück, der entsteht, wenn vom Operanden 1 subtrahiert wird. Wenn in einem checked-Kontext das Ergebnis dieser Addition oder Subtraktion außerhalb des Bereichs für den Ergebnistyp liegt und der Ergebnistyp ein Ganzzahl- oder Enumerationstyp ist, wird System.OverflowException ausgelöst.

Die Laufzeitverarbeitung einer Postfix-Inkrementoperation oder einer Postfix-Dekrementoperation der Form x++ oder x-- besteht aus den folgenden Schritten:

* Bei Klassifizierung von x als Variable:
* x wird zur Erzeugung der Variablen ausgewertet.
* Der Wert von x wird gespeichert.
* Der ausgewählte Operator wird mit dem gespeicherten Wert von x als Argument aufgerufen.
* Der vom Operator zurückgegebene Wert wird in dem Speicherort gespeichert, der durch die Auswertung von x vorgegeben wird.
* Der gespeicherte Wert von x ist das Ergebnis der Operation.
* Bei Klassifizierung von x als Eigenschaft oder Indexzugriff:
* Der mit x verknüpfte Instanzausdruck (wenn x nicht static ist) sowie die mit x verknüpfte Argumentenliste (wenn x ein Indexzugriff ist) werden ausgewertet und die Ergebnisse in den darauffolgenden Accessoraufrufen get und set verwendet.
* Der get-Accessor von x wird aufgerufen und der zurückgegebene Wert gespeichert.
* Der ausgewählte Operator wird mit dem gespeicherten Wert von x als Argument aufgerufen.
* Der set-Accessor von x wird mit dem vom Operator zurückgegebenen Wert als value-Argument aufgerufen.
* Der gespeicherte Wert von x ist das Ergebnis der Operation.

Die Operatoren ++ und -- unterstützen auch die Präfixnotation (§7.7.5). In der Regel entspricht das Ergebnis von x++ oder x-- dem Wert von x vor der Operation, wogegen das Ergebnis von ++x oder --x dem Wert von x nach der Operation entspricht. In beiden Fällen hat x nach der Operation denselben Wert.

Die Implementierung von operator ++ oder operator -- kann mithilfe der Postfix- oder Präfixnotation aufgerufen werden. Einzelne Operatorimplementierungen für beide Notationen sind nicht möglich.

### Der new-Operator

Der new-Operator wird zum Erstellen neuer Instanzen von Typen verwendet.

Es gibt drei mögliche Formen von new-Ausdrücken:

* Mit Objekterstellungsausdrücken werden neue Instanzen von Klassentypen und Werttypen erstellt.
* Arrayerstellungsausdrücke werden zum Erstellen neuer Instanzen von Arraytypen verwendet.
* Mithilfe von Delegaterstellungsausdrücken werden neue Instanzen von Delegattypen erstellt.

Der new-Operator impliziert die Erstellung einer Instanz eines Typs, nicht jedoch zwangsläufig die dynamische Speicherbelegung. Insbesondere Instanzen von Werttypen erfordern über die Variablen hinaus, in denen sie gespeichert werden, keinen zusätzlichen Speicher. Wenn mit new Instanzen von Werttypen erstellt werden, wird keine dynamische Speicherzuweisung ausgeführt.

#### Objekterstellungsausdrücke

Eine object-creation-expression wird verwendet, um eine neue Instanz eines class-type oder eines value-type zu erstellen.

object-creation-expression:  
new type ( argument-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

object-or-collection-initializer:  
object-initializer  
collection-initializer

Der Typ einer object-creation-expression muss ein class-type, ein value-type oder ein type-parameter sein. Der type darf kein abstract-class-type sein.

Die optionale argument-list (§7.5.1) ist nur zulässig, wenn der type ein class-type oder ein struct-type ist.

Ein Objekterstellungsausdruck darf die Konstruktorargumentliste und einschließende Klammern auslassen, wenn er einen Objektinitialisierer oder einen Auflistungsinitialisierer enthält. Das Auslassen von Konstruktorargumentliste und einschließenden Klammern entspricht der Angabe einer leeren Argumentliste.

Die Verarbeitung eines Objekterstellungsausdrucks, der einen Objektinitialisierer oder Auflistungsinitialisierer enthält, umfasst zunächst das Verarbeiten des Instanzkonstruktors und anschließend das Verarbeiten der Member- oder Elementinitialisierer, die vom Objektinitialisierer (§7.6.10.2) oder vom Auflistungsinitialisierer (§7.6.10.3) angegeben werden.

Wenn eines der Argumente in der optionalen *argument-list* den Kompilierungstyp dynamic aufweist, wird die object-creation-expression dynamisch gebunden (§7.2.2) und die folgenden Regeln werden zur Laufzeit unter Verwendung des Laufzeittyps der Argumente der argument-list angewendet, die den Kompilierungstyp dynamic aufweisen. Die Objekterstellung durchläuft jedoch eine beschränkte Prüfung während der Kompilierung, wie in §7.5.4 beschrieben.

Die Bindungsverarbeitung einer object-creation-expression der Form new T(A), wobei T ein class-type bzw. ein value-type ist und A einer optionalen argument-list entspricht, setzt sich aus den folgenden Schritten zusammen:

* Wenn T ein value-type und A nicht vorhanden ist:
* Die object-creation-expression ist ein standardmäßiger Konstruktoraufruf. Das Ergebnis der object-creation-expression ist ein Wert vom Typ T, und zwar der Standardwert für T (siehe Definition in §4.1.1).
* Wenn andernfalls T ein type-parameter und A nicht vorhanden ist:
* Wenn keine Werttypeinschränkung oder Konstruktoreinschränkung (§10.1.5) für T angegeben wurde, tritt während der Bindung ein Fehler auf.
* Das Ergebnis der object-creation-expression ist ein Wert des Laufzeittyps, an den der Typparameter gebunden ist, also das Ergebnis des Aufrufs des Standardkonstruktors für diesen Typ. Der Laufzeittyp kann ein Verweistyp oder ein Werttyp sein.
* Wenn T ein class-type oder ein struct-type ist, gilt andernfalls:
* Wenn T ein abstract class-type ist, tritt ein Kompilierungsfehler auf.
* Der aufzurufende Instanzkonstruktor wird entsprechend den in §7.5.3 beschriebenen Überladungsauflösungsregeln bestimmt. Die Gruppe der potenziellen Instanzkonstruktoren besteht aus allen in T deklarierten, verfügbaren Instanzkonstruktoren, die hinsichtlich A (§7.5.3.1) anwendbar sind. Wenn die Gruppe potenzieller Instanzkonstruktoren leer ist oder kein einzelner optimaler Instanzkonstruktor identifiziert werden kann, tritt während der Bindung ein Fehler auf.
* Das Ergebnis der object-creation-expression ist der Wert des Typs T, der sich durch den Aufruf des im oberen Schritt bestimmten Instanzkonstruktors ergibt.
* In allen anderen Fällen ist die object-creation-expression ungültig, und es tritt ein Bindungsfehler auf.

Auch wenn die object-creation-expression dynamisch gebunden wird, bleibt der Kompilierungstyp T.

Die Laufzeitverarbeitung einer object-creation-expression der Form new T(A), wobei T ein class-type bzw. ein struct-type ist und A einer optionalen argument-list entspricht, setzt sich aus den folgenden Schritten zusammen:

* Wenn T ein class-type ist:
* Eine neue Instanz der Klasse T wird zugewiesen. Wenn nicht genügend Speicher für die neue Instanz belegt werden kann, wird eine System.OutOfMemoryException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Alle Felder der neuen Instanz werden mit ihren Standardwerten (§5.2) initialisiert.
* Der Instanzkonstruktor wird entsprechend den in §7.5.4 beschriebenen Regeln zum Funktionsmemberaufruf aufgerufen. Ein Verweis auf die neu zugewiesene Instanz wird automatisch an den Instanzkonstruktor übergeben, und auf die Instanz kann aus dem Konstruktor als this zugegriffen werden.
* Wenn T ein struct-type ist:
* Eine Instanz vom Typ T wird erstellt, indem eine temporäre lokale Variable zugewiesen wird. Da ein Instanzkonstruktor eines struct-type erforderlich ist, um allen Feldern der zu erstellenden Instanz definitiv einen Wert zuzuweisen, ist keine Initialisierung der temporären Variablen erforderlich.
* Der Instanzkonstruktor wird entsprechend den in §7.5.4 beschriebenen Regeln zum Funktionsmemberaufruf aufgerufen. Ein Verweis auf die neu zugewiesene Instanz wird automatisch an den Instanzkonstruktor übergeben, und auf die Instanz kann aus dem Konstruktor als this zugegriffen werden.

#### Objektinitialisierer

Ein Objektinitialisierer gibt Werte für kein oder mehrere Felder oder Eigenschaften eines Objekts an.

object-initializer:  
{ member-initializer-listopt }  
{ member-initializer-list , }

member-initializer-list:  
member-initializer  
member-initializer-list , member-initializer

member-initializer:  
identifier = initializer-value

initializer-value:  
expression  
object-or-collection-initializer

Ein Objektinitialisierer besteht aus einer Reihe von Memberinitialisierern, die von den Token "{" und "}" eingeschlossen und durch ein Komma getrennt werden. Jeder Memberinitialisierer muss ein zugreifbares Feld oder eine zugreifbare Eigenschaft des zu initialisierenden Objekts gefolgt von einem Gleichheitszeichen und einem Ausdruck oder einem Objektinitialisierer bzw. Auflistungsinitialisierer angeben. Es tritt ein Fehler im Objektinitialisierer auf, wenn er für dasselbe Feld oder dieselbe Eigenschaft mehrere Memberinitialisierer enthält. Der Objektinitialisierer kann nicht auf das von ihm initialisierte neu erstellte Objekt verweisen.

Ein Memberinitialisierer, der nach dem Gleichheitszeichen einen Ausdruck angibt, wird wie eine Zuweisung (§7.17.1) zu dem Feld oder der Eigenschaft verarbeitet.

Gibt ein Memberinitialisierer nach dem Gleichheitszeichen einen Objektinitialisierer an, ist er ein geschachtelter Objektinitialisierer, d. h., ein Initialisierer für ein eingebettetes Objekt. Anstelle der Zuweisung eines neuen Werts zu dem Feld oder der Eigenschaft werden die Zuweisungen im geschachtelten Objektinitialisierer wie Zuweisungen zu Membern des Felds oder der Eigenschaft behandelt. Geschachtelte Objektinitialisierer können weder auf Eigenschaften mit einem Werttyp noch auf schreibgeschützte Felder mit einem Werttyp angewendet werden.

Gibt ein Memberinitialisierer nach dem Gleichheitszeichen einen Auflistungsinitialisierer an, handelt es sich um einen Initialisierer für eine eingebettete Auflistung. Anstelle der Zuweisung einer neuen Auflistung zu dem Feld oder der Eigenschaft werden die im Initialisierer angegebenen Elemente der Auflistung hinzugefügt, auf die das Feld oder die Eigenschaft verweist. Das Feld bzw. die Eigenschaft muss einen Auflistungstyp aufweisen, der die in §7.6.10.3 angegebenen Bedingungen erfüllt.

Die folgende Klasse stellt einen Punkt mit zwei Koordinaten dar:

public class Point  
{  
 int x, y;

public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Ein Instanz von Point kann folgendermaßen erstellt und initialisiert werden:

Point a = new Point { X = 0, Y = 1 };

hat dieselben Auswirkungen wie

Point \_\_a = new Point();  
\_\_a.X = 0;  
\_\_a.Y = 1;   
Point a = \_\_a;

Dabei ist \_\_a eine ansonsten unsichtbare temporäre Variable, auf die nicht zugegriffen werden kann. Die folgende Klasse stellt ein Rechteck dar, das aus zwei Punkten erstellt wird:

public class Rectangle  
{  
 Point p1, p2;

public Point P1 { get { return p1; } set { p1 = value; } }  
 public Point P2 { get { return p2; } set { p2 = value; } }  
}

Ein Instanz von Rectangle kann folgendermaßen erstellt und initialisiert werden:

Rectangle r = new Rectangle {  
 P1 = new Point { X = 0, Y = 1 },  
 P2 = new Point { X = 2, Y = 3 }  
};

hat dieselben Auswirkungen wie

Rectangle \_\_r = new Rectangle();  
Point \_\_p1 = new Point();  
\_\_p1.X = 0;  
\_\_p1.Y = 1;  
\_\_r.P1 = \_\_p1;  
Point \_\_p2 = new Point();  
\_\_p2.X = 2;  
\_\_p2.Y = 3;  
\_\_r.P2 = \_\_p2;   
Rectangle r = \_\_r;

Dabei sind \_\_r, \_\_p1 und \_\_p2 temporäre Variablen, die ansonsten unsichtbar sind und auf die nicht zugegriffen werden kann.

Wenn der Konstruktor von Rectangle die beiden eingebetteten Instanzen von Point zuordnet:

public class Rectangle  
{  
 Point p1 = new Point();  
 Point p2 = new Point();

public Point P1 { get { return p1; } }  
 public Point P2 { get { return p2; } }  
}

kann folgendes Konstrukt zum Initialisieren der eingebetteten Point-Instanzen verwendet werden, sodass keine neuen Instanzen zugewiesen werden müssen:

Rectangle r = new Rectangle {  
 P1 = { X = 0, Y = 1 },  
 P2 = { X = 2, Y = 3 }  
};

hat dieselben Auswirkungen wie

Rectangle \_\_r = new Rectangle();  
\_\_r.P1.X = 0;  
\_\_r.P1.Y = 1;  
\_\_r.P2.X = 2;  
\_\_r.P2.Y = 3;  
Rectangle r = \_\_r;

#### Auflistungsinitialisierer

Ein Auflistungsinitialisierer gibt die Elemente einer Auflistung an.

collection-initializer:  
{ element-initializer-list }  
{ element-initializer-list , }

element-initializer-list:  
element-initializer  
element-initializer-list , element-initializer

element-initializer:  
non-assignment-expression  
{ expression-list }

expression-list:  
expression  
expression-list , expression

Ein Auflistungsinitialisierer besteht aus einer Reihe von Elementinitialisierern, die von den Token "{" und "}" eingeschlossen und durch ein Komma getrennt werden. Jeder Elementinitialisierer gibt ein Element an, dass dem zu initialisierenden Auflistungsobjekt hinzugefügt werden soll und aus einer Liste von Ausdrücken besteht, die von den Token "{" und "}" eingeschlossen und durch ein Komma getrennt werden. Ein Elementinitialisierer, der nur aus einem Ausdruck besteht, kann ohne geschweifte Klammern geschrieben werden, er darf dann aber keinem Zuweisungsausdruck zugeordnet werden, um Mehrdeutigkeiten bei Memberinitialisierern zu vermeiden. Die non-assignment-expression-Produktion wird in §7.18 definiert.

Im Folgenden sehen Sie ein Beispiel für einen Objekterstellungsausdruck mit einem Auflistungsinitialisierer:

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Der Typ des Auflistungsobjekts, auf das ein Auflistungsinitialisierer angewendet wird, muss System.Collections.IEnumerable implementieren, andernfalls tritt während der Kompilierung ein Fehler auf. Der Auflistungsinitialisierer ruft für jedes angegebene Element in der angegebenen Reihenfolge eine Add-Methode im Zielobjekt auf. Dabei wird die Ausdrucksliste des Elementinitialisierers als Argumentliste verwendet, und bei jedem Aufruf wird eine normale Überladungsauflösung angewendet. Daher muss das Auflistungsobjekt eine anwendbare Add-Methode für jeden Elementinitialisierer enthalten.

Die folgende Klasse stellt einen Kontakt durch einen Namen und eine Liste von Telefonnummern dar:

public class Contact  
{  
 string name;  
 List<string> phoneNumbers = new List<string>();

public string Name { get { return name; } set { name = value; } }

public List<string> PhoneNumbers { get { return phoneNumbers; } }  
}

Eine List List<Contact> kann folgendermaßen erstellt und initialisiert werden:

var contacts = new List<Contact> {  
 new Contact {  
 Name = "Chris Smith",  
 PhoneNumbers = { "206-555-0101", "425-882-8080" }  
 },  
 new Contact {  
 Name = "Bob Harris",  
 PhoneNumbers = { "650-555-0199" }  
 }  
};

hat dieselben Auswirkungen wie

var \_\_clist = new List<Contact>();  
Contact \_\_c1 = new Contact();  
\_\_c1.Name = "Chris Smith";  
\_\_c1.PhoneNumbers.Add("206-555-0101");  
\_\_c1.PhoneNumbers.Add("425-882-8080");  
\_\_clist.Add(\_\_c1);  
Contact \_\_c2 = new Contact();  
\_\_c2.Name = "Bob Harris";  
\_\_c2.PhoneNumbers.Add("650-555-0199");  
\_\_clist.Add(\_\_c2);  
var contacts = \_\_clist;

Dabei sind \_\_clist, \_\_c1 und \_\_c2 temporäre Variablen, die ansonsten unsichtbar sind und auf die nicht zugegriffen werden kann.

#### Arrayerstellungsausdrücke

Eine array-creation-expression wird verwendet, um eine neue Instanz eines array-type zu erstellen.

array-creation-expression:  
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

Ein Arrayerstellungsausdruck der ersten Form weist eine Arrayinstanz des Typs zu, der sich aus dem Löschen der einzelnen Ausdrücke aus der Ausdrucksliste ergibt. Der Arrayerstellungsausdruck new int[10, 20] ergibt z. B. eine Arrayinstanz des Typs int[,] und der Arrayerstellungsausdruck new int[10][,] ergibt ein Array des Typs int[][,]. Alle Ausdrücke in der Ausdrucksliste müssen vom Typ int, uint, long oder ulong sein oder implizit in einen oder mehrere dieser Typen konvertiert werden können. Der Wert der Ausdrücke legt die Länge der jeweiligen Dimension in der neu zugewiesenen Arrayinstanz fest. Da die Länge einer Arraydimension nicht negativ sein darf, wird durch Angabe einer constant-expression mit einem negativen Wert in der Ausdrucksliste ein Kompilierungsfehler verursacht.

Außer in einem unsicheren Kontext (§18.1) ist das Layout von Arrays unspezifisch.

Wenn ein Ausdruck zur Arrayerstellung der ersten Form einen Arrayinitialisierer enthält, müssen alle Ausdrücke in der Ausdrucksliste Konstanten sein, und der durch die Ausdrucksliste festgelegte Rang sowie die Dimensionslängen müssen denen des Arrayinitialisierers entsprechen.

In einem Arrayerstellungsausdruck der zweiten oder dritten Form muss der Rang des festgelegten Arraytyps oder der Rangspezifizierer dem des Arrayinitialisierers entsprechen. Die einzelnen Dimensionslängen werden aus der Anzahl der Elemente der jeweiligen Schachtelungsebenen des Arrayinitialisierers abgeleitet. Folglich stimmt der Ausdruck

new int[,] {{0, 1}, {2, 3}, {4, 5}}

genau mit dem folgenden Ausdruck überein

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}

Ein Arrayerstellungsausdruck der dritten Form wird als implizit typisierter Arrayerstellungsausdruck bezeichnet. Er entspricht denen der zweiten Form mit der Ausnahme, dass der Elementtyp des Arrays nicht explizit angegeben wird, sondern als der optimale Typ (§7.5.2.14) für den Satz von Ausdrücken im Arrayinitialisierer bestimmt wird. Bei einem mehrdimensionalen Array, d. h., bei dem der rank-specifier mindestens ein Komma enthält, setzt sich dieser Satz als sämtlichen expressions in den geschachtelten array-initializers zusammen.

Arrayinitialisierer werden in §12.6 genauer beschrieben.

Das Ergebnis der Auswertung eines Arrayerstellungsausdrucks wird als Wert klassifiziert, nämlich als Verweis auf die neu zugewiesene Arrayinstanz. Die Laufzeitverarbeitung eines Arrayerstellungsausdrucks besteht aus den folgenden Schritten:

* Die Ausdrücke für die Dimensionslänge der expression-list werden in der Reihenfolge von links nach rechts ausgewertet. Nach der Auswertung der einzelnen Indexausdrücke wird eine implizite Konvertierung (§6.1) in einen der folgenden Typen ausgeführt: int, uint, long, ulong. Dabei wird der erste Typ in der Liste ausgewählt, für den eine implizite Konvertierung vorhanden ist. Wenn die Auswertung eines Ausdrucks oder einer nachfolgenden impliziten Konvertierung eine Ausnahme verursacht, werden weder weitere Ausdrücke ausgewertet noch weitere Schritte ausgeführt.
* Die errechneten Werte für die Dimensionslängen werden wie folgt ausgewertet: Wenn einer oder mehrere Werte kleiner als 0 sind, wird System.OverflowException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Eine Arrayinstanz mit den gegebenen Dimensionslängen wird zugewiesen. Wenn nicht genügend Speicher für die neue Instanz belegt werden kann, wird eine System.OutOfMemoryException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Alle Elemente der neuen Arrayinstanz werden mit ihren Standardwerten (§5.2) initialisiert.
* Wenn der Arrayerstellungsausdruck einen Arrayinitialisierer enthält, werden die einzelnen Ausdrücke im Arrayinitialisierer ausgewertet und den zugehörigen Arrayelementen zugewiesen. Die Auswertungen und Zuweisungen werden in der Reihenfolge ausgeführt, in der die Ausdrücke in den Arrayinitialisierer geschrieben wurden, d. h. die Elemente werden in aufsteigender Indexreihenfolge initialisiert, wobei die am weitesten rechts stehende Dimension zuerst erhöht wird. Wenn die Auswertung eines gegebenen Ausdrucks oder einer nachfolgenden Zuweisung zum jeweiligen Arrayelement eine Ausnahme verursacht, werden keine weiteren Elemente initialisiert (die verbleibenden Elemente behalten daher die Standardwerte).

Ein Arrayerstellungsausdruck ermöglicht die Instanziierung eines Arrays mit Elementen eines Arraytyps, wobei die Elemente dieses Arrays jedoch manuell initialisiert werden müssen. Die Anweisung

int[][] a = new int[100][];

erstellt z. B. ein eindimensionales Array mit 100 Elementen vom Typ int[]. Der Anfangswert der einzelnen Elemente ist null. Derselbe Arrayerstellungsausdruck kann nicht gleichzeitig die untergeordneten Arrays instanziieren, und die Anweisung

int[][] a = new int[100][5]; // Error

führt zu einem Kompilierungsfehler. Die Instanziierung untergeordneter Arrays muss stattdessen manuell ausgeführt werden, so z. B. im folgenden Beispiel:

int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];

Wenn ein zusammengesetztes Array eine „rechteckige“ Form aufweist, also alle untergeordneten Arrays dieselbe Länge haben, ist es sinnvoller, ein mehrdimensionales Array zu verwenden. Im obigen Beispiel werden durch die Instanziierung des zusammengesetzten Arrays 101 Objekte erstellt: ein äußeres Array und 100 untergeordnete Arrays. Im Gegensatz dazu erstellt

int[,] = new int[100, 5];

nur ein einzelnes Objekt, ein zweidimensionales Array, und vervollständigt die Zuweisung in eine einzelne Anweisung.

Im Folgenden sind einige Beispiele für implizit typisierte Arrayerstellungsausdrücke aufgeführt:

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { 1, 1.5, 2, 2.5 }; // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" }; // Error

Der letzte Ausdruck führt während der Kompilierung zu einem Fehler, da weder int noch string implizit in den jeweils anderen Typ konvertiert werden können, sodass kein optimaler Typ festgelegt werden kann. In diesem Fall muss ein explizit typisierter Arrayerstellungsausdruck verwendet werden, bei dem z. B. als Typ object[] angegeben wird. Es ist jedoch auch möglich, eines der Elemente in einen gemeinsamen Basistyp umzuwandeln, der dann der hergeleitete Elementtyp ist.

Implizit typisierte Arrayerstellungsausdrücke können mit anonymen Objektinitialisierern (§7.6.10.6) kombiniert werden, um anonym typisierte Datenstrukturen zu erstellen. Beispiel:

var contacts = new[] {  
 new {  
 Name = "Chris Smith",  
 PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }  
 },  
 new {  
 Name = "Bob Harris",  
 PhoneNumbers = new[] { "650-555-0199" }  
 }  
};

#### Delegaterstellungsausdrücke

Eine delegate-creation-expression wird verwendet, um eine neue Instanz eines delegate-type zu erstellen.

delegate-creation-expression:  
new delegate-type ( expression )

Das Argument eines Delegaterstellungsausdrucks muss eine Methodengruppe, eine anonyme Funktion oder ein Wert mit dem Kompilierungstyp dynamic bzw. ein delegate-type sein. Wenn das Argument eine Methodengruppe ist, wird die Methode und bei einer Instanzmethode das Objekt erkannt, für das ein Delegat erstellt werden soll. Wenn das Argument eine anonyme Funktion ist, definiert es direkt die Parameter und den Methodentext des Delegatziels. Wenn das Argument ein Wert ist, wird eine Delegatinstanz identifiziert, von der eine Kopie erstellt werden soll.

Wenn die expression den Kompilierungstyp dynamic aufweist, wird die delegate-creation-expression dynamisch gebunden (§7.2.2), und die folgenden Regeln werden zur Laufzeit unter Verwendung des Laufzeittyps der expression angewendet. Andernfalls werden die Regeln während der Kompilierung angewendet.

Die Verarbeitung einer delegate-creation-expression der Form new D(E) während der Bindung (wobei D ein delegate-type und E ein Ausdruck ist) setzt sich aus den folgenden Schritten zusammen:

* Wenn E eine Methodengruppe ist, wird der Delegaterstellungsausdruck wie eine Methodengruppenkonvertierung (§6.6) von E in D verarbeitet.
* Wenn E eine anonyme Funktion ist, wird der Delegaterstellungsausdruck auf dieselbe Weise wie eine Konvertierung einer anonymen Funktion (§6.5) von E in D verarbeitet.
* Wenn E ein Wert ist, muss E mit (§15.1) mit D kompatibel sein, und das Ergebnis ist ein Verweis auf einen neu erstellten Delegaten vom Typ D, der auf dieselbe Aufrufliste wie E verweist. Wenn E mit D nicht kompatibel ist, tritt ein Kompilierungsfehler auf.

Die Verarbeitung einer delegate-creation-expression der Form new D(E) während der Laufzeit (wobei D ein delegate-type und E ein Ausdruck ist) setzt sich aus den folgenden Schritten zusammen:

* Wenn E eine Methodengruppe ist, wird der Delegaterstellungsausdruck wie eine Methodengruppenkonvertierung (§6.6) von E in Dausgewertet.
* Wenn E eine anonyme Funktion ist, wird der Delegaterstellungsausdruck auf dieselbe Weise wie eine Konvertierung einer anonymen Funktion von E nach D (§6.5) ausgewertet.
* Wenn E ein Wert eines delegate-type ist:
* E wird ausgewertet. Wenn diese Auswertung eine Ausnahme verursacht, werden keine weiteren Schritte ausgeführt.
* Wenn der Wert von E gleich null ist, wird System.NullReferenceException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Eine neue Instanz des Delegattyps D wird zugewiesen. Wenn nicht genügend Speicher für die neue Instanz belegt werden kann, wird eine System.OutOfMemoryException ausgelöst, und es werden keine weiteren Schritte ausgeführt.
* Die neue Delegatinstanz wird mit derselben Aufrufliste initialisiert wie die von E vorgegebene Delegatinstanz.

Die Aufrufliste eines Delegaten wird bei der Instanziierung des Delegaten festgelegt und bleibt für die gesamte Lebensdauer des Delegaten konstant. Anders ausgedrückt heißt dies, dass die aufrufbaren Zielentitäten eines Delegaten nach der Erstellung nicht geändert werden können. Wenn zwei Delegaten miteinander verbunden sind oder einer davon entfernt wird (§15.1), entsteht ein neuer Delegat, wobei jedoch der Inhalt eines vorhandenen Delegaten nicht verändert wird.

Das Erstellen eines Delegaten mit einem Verweis auf eine Eigenschaft, einen Indexer, einen benutzerdefinierten Operator, einen Instanzkonstruktor, einen Destruktor oder einen statischen Konstruktor ist nicht möglich.

Wenn ein Delegat aus einer Methodengruppe erstellt wird, bestimmen die Liste formaler Parameter und der Rückgabetyp des Delegaten, welche Überladungsmethode ausgewählt wird (siehe Beschreibung weiter oben). In dem Beispiel

delegate double DoubleFunc(double x);

class A  
{  
 DoubleFunc f = new DoubleFunc(Square);

static float Square(float x) {  
 return x \* x;  
 }

static double Square(double x) {  
 return x \* x;  
 }  
}

wird das A.f-Feld mit einem Delegaten initialisiert, der auf die zweite Square-Methode verweist, da diese Methode genau mit der Liste formaler Parameter übereinstimmt und den Typ von DoubleFunc zurückgibt. Wenn die zweite Square-Methode nicht vorhanden ist, tritt während der Kompilierung ein Fehler auf.

#### Anonyme Objekterstellungsausdrücke

Eine anonymous-object-creation-expression wird verwendet, um ein Objekt eines anonymen Typs zu erstellen.

anonymous-object-creation-expression:  
new anonymous-object-initializer

anonymous-object-initializer:  
{ member-declarator-listopt }  
{ member-declarator-list , }

member-declarator-list:  
member-declarator  
member-declarator-list , member-declarator

member-declarator:  
simple-name  
member-access  
base-access  
identifier = expression

Ein anonymer Objektinitialisierer deklariert einen anonymen Typ und gibt eine Instanz dieses Typs zurück. Ein anonymer Typ ist ein Klassentyp ohne Namen, der direkt von object erbt. Die Member eines anonymen Typs bilden eine Folge von schreibgeschützten Eigenschaften, für die vom anonymen Objektinitialisierer ein Rückschluss durchgeführt wird und die zum Erstellen einer Instanz des Typs verwendet werden. Ein anonymer Objektinitialisierer der Form

new { p1 = e1 , p2 = e2 , … pn = en }

deklariert einen anonymen Typ der Form

class \_\_Anonymous1  
{  
 private readonly T1 f1 ;  
 private readonly T2 f2 ;  
 …  
 private readonly Tn fn ;

public \_\_Anonymous1(T1 a1, T2 a2,…, Tn an) {  
 f1 = a1 ;  
 f2 = a2 ;  
 …  
 fn = an ;  
 }

public T1 p1 { get { return f1 ; } }  
 public T2 p2 { get { return f2 ; } }  
 …  
 public Tn pn { get { return fn ; } }

public override bool Equals(object \_\_o) { … }  
 public override int GetHashCode() { … }  
}

Dabei ist jedes Tx der Typ des zugehörigen Ausdrucks ex. Der in einem member-declarator verwendete Ausdruck muss über einen Typ verfügen. Daher tritt während der Kompilierung ein Fehler auf, wenn ein Ausdruck in einem member-declarator gleich NULL oder eine anonyme Funktion ist. Es tritt auch dann ein Fehler während der Kompilierung auf, wenn der Ausdruck einen unsicheren Typ aufweist.

Der Name eines anonymen Typs und des Paramenters für die zugehörige Equals-Methode wird automatisch vom Compiler generiert, sodass im Programmtext nicht darauf verwiesen werden kann.

Wenn in einem Programm zwei anonyme Objektinitialisierer eine Folge von Eigenschaften mit denselben Namen und Kompilierungstypen in derselben Reihenfolge angeben, werden auch Instanzen desselben anonymen Typs erstellt.

In dem Beispiel

var p1 = new { Name = "Lawnmower", Price = 495.00 };  
var p2 = new { Name = "Shovel", Price = 26.95 };  
p1 = p2;

ist die Zuweisung auf der letzten Zeile zulässig, da p1 und p2 denselben anonymen Typ aufweisen.

Die Equals-Methode und die GetHashcode-Methode überschreiben bei anonymen Typen die von object geerbten Methoden und werden als Equals und GetHashcode der Eigenschaften definiert, sodass zwei Instanzen desselben anonymen Typs gleich sind, wenn lediglich alle ihre Eigenschaften gleich sind.

Ein Memberdeklarator kann mit einem einfachen Namen (§7.5.2), einem Memberzugriff (§7.5.4) oder einem Basiszugriff (§7.6.8) abgekürzt werden. Dies wird als projection initializer bezeichnet und stellt die Kurzform für das Deklarieren einer Eigenschaft und das Zuweisen zu einer Eigenschaft desselben Namens dar. Memberdeklaratoren der Formen

identifier expr . identifier

sind jeweils mit Folgendem identisch:

identifer = identifier identifier = expr . identifier

Daher wählt der identifier bei einem Projektionsinitialisierer sowohl den Wert als auch das Feld bzw. die Eigenschaft für die Wertzuweisung aus. Daher projiziert ein Projektionsinitialisierer offensichtlich nicht nur einen Wert, sondern auch den Namen des Werts.

### Der typeof-Operator

Der typeof-Operator wird verwendet, um das System.Type-Objekt für einen Typ abzurufen.

typeof-expression:  
typeof ( type )  
typeof ( unbound-type-name )  
typeof ( void )

unbound-type-name:  
identifier generic-dimension-specifieropt  
identifier :: identifier generic-dimension-specifieropt  
unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:  
< commasopt >

commas:  
,  
commas ,

Die erste Form der typeof-expression-expression besteht aus dem typeof-Schlüsselwort, gefolgt von einem type in Klammern. Das Ergebnis eines Ausdrucks dieser Form ist das System.Type-Objekt für den angegebenen Typ. Für jeden gegebenen Typ ist jeweils nur ein System.Type-Objekt vorhanden. Das bedeutet, dass der Ausdruck typeof(T) == typeof(T) für den Typ T stets zutrifft. Der type kann nicht dynamic sein.

Die zweite Form der typeof-expression-expression besteht aus einem typeof-Schlüsselwort, gefolgt von einem in Klammern stehenden unbound-type-name. Ein unbound-type-name gleicht stark einem type-name (§3.8), mit der Ausnahme, dass unbound-type-name generic-dimension-specifiers enthält, wohingegen ein type-name type-argument-lists enthält. Wenn der Operand einer typeof-expression eine Folge von Token ist, die sowohl der Grammatik von unbound-type-name als auch der von type-name entspricht, d. h., wenn er weder einen generic-dimension-specifier noch eine type-argument-list enthält, wird die Folge von Token als type-name betrachtet. Die Bedeutung eines unbound-type-name wird folgendermaßen bestimmt:

* Konvertieren der Folge von Token in einen type-name, indem jeder generic-dimension-specifier durch eine type-argument-list mit der gleichen Anzahl von Kommas und dem object-Schlüsselwort für jedes type-argument ersetzt wird.
* Auswerten des resultierenden type-name, wobei alle Typparametereinschränkungen ignoriert werden.
* Der unbound-type-name wird in den ungebundenen generischen Typ aufgelöst, der dem resultierenden konstruierten Typ (§4.4.3) zugeordnet ist.

Das Ergebnis der typeof-expression ist das System.Type-Objekt für den resultierenden ungebundenen generischen Typ.

Die dritte Form von typeof-expression besteht aus einem typeof-Schlüsselwort, gefolgt von einem in Klammern stehenden void-Schlüsselwort. Das Ergebnis eines Ausdrucks dieser Form ist das System.Type-Objekt, das für einen nicht vorhandenen Typ steht. Das von typeof(void) zurückgegebene Typobjekt unterscheidet sich von dem für einen beliebigen Typ zurückgegebenen Typobjekt. Dieses besondere Typobjekt ist in Klassenbibliotheken hilfreich, die eine Reflektion für Methoden in der Sprache zulassen, in der diese Methoden eine Möglichkeit haben sollen, die Rückgabetypen einer beliebigen Methode (einschließlich leerer Methoden) mit einer Instanz von System.Type zu repräsentieren.

Der typeof-Operator kann für einen Typparameter verwendet werden. Das Ergebnis ist das System.Type-Objekt für den Laufzeittyp, der an den Typparameter gebunden wurde. Der typeof-Operator kann auch für einen konstruierten Typ oder einen nicht gebundenen generischen Typ (§4.4.3) verwendet werden. Das System.Type-Objekt für einen ungebundenen generischen Typ entspricht nicht dem System.Type-Objekt des Instanztyps. Der Instanztyp ist zur Laufzeit immer ein geschlossener konstruierter Typ, sodass dessen System.Type-Objekt von den verwendeten Laufzeit-Typargumenten abhängig ist, wohingegen der ungebundene generische Typ keine Typargumente besitzt.

In dem Beispiel

using System;

class X<T>  
{  
 public static void PrintTypes() {  
 Type[] t = {  
 typeof(int),  
 typeof(System.Int32),  
 typeof(string),  
 typeof(double[]),  
 typeof(void),  
 typeof(T),  
 typeof(X<T>),  
 typeof(X<X<T>>),  
 typeof(X<>)  
 };  
 for (int i = 0; i < t.Length; i++) {  
 Console.WriteLine(t[i]);  
 }  
 }  
}

class Test  
{  
 static void Main() {  
 X<int>.PrintTypes();  
 }  
}

erzeugt die folgende Ausgabe:

System.Int32  
System.Int32  
System.String  
System.Double[]  
System.Void  
System.Int32  
X`1[System.Int32]  
X`1[X`1[System.Int32]]  
X`1[T]

Beachten Sie, dass int und System.Int32 denselben Typ haben.

Beachten Sie außerdem, dass das Ergebnis von typeof(X<>) im Gegensatz zum Ergebnis von typeof(X<T>) nicht vom Typargument abhängig ist.

### Der checked-Operator und der unchecked-Operator

Der checked-Operator und der unchecked-Operator werden verwendet, um den Kontext der Überlaufprüfung für arithmetische Ganzzahloperationen und Konvertierungen zu steuern.

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

Der checked-Operator wertet den enthaltenen Ausdruck im Kontext der Überlaufprüfung aus, der unchecked-Operator wertet den enthaltenen Ausdruck in einem Kontext ohne Überlaufprüfung aus. Eine checked-expression oder eine unchecked-expression entspricht genau einer parenthesized-expression (§7.6.3), mit der Ausnahme, dass der enthaltene Ausdruck im gegebenen Kontext mit Überlaufprüfung ausgewertet wird.

Der Kontext mit Überlaufprüfung kann auch über die Anweisungen checked und unchecked (§8.11) gesteuert werden.

Die folgenden Operationen werden durch den Kontext mit Überlaufprüfung beeinflusst, der durch die Operatoren und Anweisungen checked und unchecked erstellt wird.

* Die vordefinierten unären Operatoren ++ und -- (§7.6.9 und §7.7.5), wenn der Operand ein ganzzahliger Typ ist.
* Der vordefinierte unäre Operator - (§7.7.2), wenn der Operand ein ganzzahliger Typ ist.
* Die vordefinierten binären Operatoren +, -, \* und / (§7.8), wenn beide Operanden ganzzahlige Typen sind.
* Explizite numerische Konvertierungen (§6.2.1) von einem ganzzahligen Typ in einen anderen ganzzahligen Typ oder von float bzw. double in einen ganzzahligen Typ.

Wenn eine der aufgeführten Operationen ein Ergebnis ausgibt, das im Zieltyp aufgrund seiner Größe nicht dargestellt werden kann, steuert der Kontext, in dem die Operation ausgeführt wird, das Verhalten:

* In einem checked-Kontext tritt ein Kompilierungsfehler auf, wenn die Operation ein konstanter Ausdruck (§7.19) ist. In allen anderen Fällen wird, wenn die Operation zur Laufzeit ausgeführt wird, System.OverflowException ausgelöst.
* In einem unchecked-Kontext wird das Ergebnis gekürzt, indem alle höherwertigen Bits, die nicht in den Zieltyp passen, verworfen werden.

Für nicht konstante Ausdrücke (Ausdrücke, die zur Laufzeit ausgewertet werden), die nicht von den Operatoren oder Anweisungen checked oder unchecked eingeschlossen sind, ist der Standardkontext mit Überlaufprüfung unchecked, vorausgesetzt externe Faktoren (wie Compilerschalter und Konfigurationen der Ausführungsumgebung) rufen keine checked-Auswertung auf.

Für konstante Ausdrücke (Ausdrücke, die während der Kompilierung vollständig ausgewertet werden können) ist der Standardkontext mit Überlaufprüfung stets checked. Überläufe während der Kompilierungsauswertung eines Ausdrucks verursachen nur dann einen Fehler während der Kompilierung, wenn kein konstanter Ausdruck explizit in einen unchecked-Kontext eingefügt wird.

Auf den Text einer anonymen Funktion hat der checked-Kontext oder der unchecked-Kontext, in dem die anonyme Funktion auftritt, keine Auswirkung.

In dem Beispiel

class Test  
{  
 static readonly int x = 1000000;  
 static readonly int y = 1000000;

static int F() {  
 return checked(x \* y); // Throws OverflowException  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Depends on default  
 }  
}

werden keine Fehler während der Kompilierung ausgegeben, da keiner der Ausdrücke während der Kompilierung ausgewertet werden kann. Zur Laufzeit löst die F-Methode eine System.OverflowException aus, und die G-Methode gibt den Wert –727379968 zurück (die niedrigsten 32 Bits des Ergebnisses außerhalb des Bereichs). Das Verhalten der H-Methode hängt vom Standardkontext mit Überlaufprüfung für die Kompilierung ab, ist jedoch mit dem der Methoden F oder G identisch.

In dem Beispiel

class Test  
{  
 const int x = 1000000;  
 const int y = 1000000;

static int F() {  
 return checked(x \* y); // Compile error, overflow  
 }

static int G() {  
 return unchecked(x \* y); // Returns -727379968  
 }

static int H() {  
 return x \* y; // Compile error, overflow  
 }  
}

verursachen die bei der Auswertung der konstanten Ausdrücke in F und H auftretenden Überläufe Fehler während der Kompilierung, die gemeldet werden müssen, da die Ausdrücke in einem checked-Kontext ausgewertet werden. Ein Überlauf tritt auch beim Auswerten des konstanten Ausdrucks in G auf, da die Überprüfung jedoch in einem unchecked-Kontext stattfindet, wird der Überlauf nicht gemeldet.

Die Operatoren checked und unchecked betreffen nur den Kontext der Überlaufprüfung für die Operationen, die im Text innerhalb der Token "(" und ")" enthalten sind. Die Operatoren haben keine Auswirkung auf Funktionsmember, die aufgrund der Auswertung des enthaltenen Ausdrucks aufgerufen werden. In dem Beispiel

class Test  
{  
 static int Multiply(int x, int y) {  
 return x \* y;  
 }

static int F() {  
 return checked(Multiply(1000000, 1000000));  
 }  
}

wirkt sich die Verwendung von checked in F nicht auf die Auswertung von x \* y in Multiply aus. Daher wird x \* y im Standardkontext der Überlaufprüfung ausgewertet.

Der unchecked-Operator eignet sich besonders, wenn Konstanten von Ganzzahltypen mit Vorzeichen in Hexadezimalschreibweise erfasst werden. Beispiel:

class Test  
{  
 public const int AllBits = unchecked((int)0xFFFFFFFF);

public const int HighBit = unchecked((int)0x80000000);  
}

Beide der obigen hexadezimalen Konstanten sind vom Typ uint. Da die Konstanten außerhalb des int-Bereichs liegen, verursachen Typumwandlungen in int ohne den unchecked-Operator Fehler während der Kompilierung.

Über die Operatoren checked und unchecked sowie Anweisungen können Programmierer bestimmte Aspekte einiger numerischer Berechnungen steuern. Das Verhalten einiger numerischer Operatoren richtet sich jedoch nach den Datentypen ihrer Operanden. Das Multiplizieren zweier Dezimalzahlen resultiert z. B. immer in einer Überlaufausnahme, selbst in einem explizit als unchecked deklarierten Konstrukt. Entsprechend kann aber das Multiplizieren zweier Gleitkommazahlen nie zu einer Überlaufausnahme führen, selbst nicht in einem explizit als checked deklarierten Konstrukt. Darüber hinaus wirkt sich der Überprüfungsmodus nie auf andere Operatoren aus, unabhängig davon, ob er standardmäßig oder explizit ist.

### Standardwertausdrücke

Mithilfe eines Standardwertausdrucks wird der Standardwert (§5.2) eines Typs abgerufen. Im Allgemeinen wird ein Standardwertausdruck für Typparameter verwendet, da nicht unbedingt bekannt ist, ob es sich bei dem Typparameter um einen Werttyp oder um einen Verweistyp handelt. (Die Konvertierung vom null-Literal in einen Typparameter ist nicht möglich, es sei denn, der Typparameter ist als Verweistyp bekannt.)

default-value-expression:  
default ( type )

Wenn der type in einer default-value-expression zur Laufzeit als Verweistyp ausgewertet wird, ist das Ergebnis null und wird in den betreffenden Typ konvertiert. Wenn der type in einer default-value-expression zur Laufzeit als Werttyp ausgewertet wird, ist das Ergebnis der Standardwert für den value-type (§4.1.2).

Eine default-value-expression ist ein konstanter Ausdruck (§7.19), wenn es sich bei dem Typ um einen Verweistyp oder einen Typparameter handelt, der als Verweistyp bekannt ist (§10.1.5). Außerdem ist eine default-value-expression ein konstanter Ausdruck, wenn der Typ einen der folgenden Werttypen darstellt: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool oder ein beliebiger Enumerationstyp.

### Ausdrücke für anonyme Methoden

Eine anonymous-method-expression ist eine von zwei Möglichkeiten, anonyme Funktionen zu definieren. Diese werden in §7.15 genauer beschrieben.

## Unäre Operatoren

Die Operatoren +, -, !, ~, ++, --, await sowie Umwandlungsoperatoren werden als unäre Operatoren bezeichnet.

unary-expression:  
primary-expression  
+ unary-expression  
- unary-expression  
! unary-expression  
~ unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression  
await-expression

Wenn der Operand einer unary-expression den Kompilierungstyp dynamic aufweist, wird er dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp der unary-expression dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps des Operanden.

### Unärer Plus-Operator

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird auf Operationen der Form +x angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Der Operand wird in den Parametertyp des ausgewählten Operators konvertiert, wobei der Ergebnistyp dem Rückgabetyp des Operators entspricht. Folgende unäre Plus-Operatoren sind vordefiniert:

int operator +(int x);  
uint operator +(uint x);  
long operator +(long x);  
ulong operator +(ulong x);  
float operator +(float x);  
double operator +(double x);  
decimal operator +(decimal x);

Das Ergebnis dieser Operatoren entspricht dem Wert des Operanden.

### Unärer Minus-Operator

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird auf Operationen der Form –x angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Der Operand wird in den Parametertyp des ausgewählten Operators konvertiert, wobei der Ergebnistyp dem Rückgabetyp des Operators entspricht. Folgende Negationsoperatoren sind vordefiniert:

* Ganzzahlige Negation:

int operator –(int x);  
long operator –(long x);

Zur Errechnung des Ergebnisses wird x von 0 (null) subtrahiert. Wenn der Wert von x der kleinste darstellbare Wert des Operandentyp ist (−231 für int oder −263 für long)), dann ist die mathematische Negation von x innerhalb des Operandentyps nicht darstellbar. Falls dies in einem checked-Kontext auftritt, wird eine System.OverflowException ausgelöst; falls dies in einem unchecked-Kontext auftritt, wird der Wert des Operanden als Ergebnis verwendet und der Überlauf nicht gemeldet.

Wenn der Operand des Negationsoperators vom Typ uint ist, wird dieser in den Typ long konvertiert, und das Ergebnis ist vom Typ long. Eine Ausnahme bildet die Regel, nach der der int-Wert −2147483648 (−231) als dezimales ganzzahliges Literal (§2.4.4.2) geschrieben werden kann.

Wenn der Operand des Negationsoperators vom Typ ulong ist, tritt ein Kompilierungsfehler auf. Eine Ausnahme bildet die Regel, nach der der long-Wert −9223372036854775808 (−263) als dezimales ganzzahliges Literal (§2.4.4.2) geschrieben werden kann.

* Negation mit Gleitkommawerten:

float operator –(float x);  
double operator –(double x);

Das Ergebnis ist der Wert von x mit umgekehrtem Vorzeichen. Wenn x gleich NaN ist, ist das Ergebnis ebenfalls NaN.

* Dezimale Negation:

decimal operator –(decimal x);

Zur Errechnung des Ergebnisses wird x von 0 (null) subtrahiert. Die dezimale Negation ist äquivalent zur Verwendung unärer Minusoperatoren vom Typ System.Decimal.

### Logischer Negationsoperator

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird auf Operationen der Form !x angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Der Operand wird in den Parametertyp des ausgewählten Operators konvertiert, wobei der Ergebnistyp dem Rückgabetyp des Operators entspricht. Es ist nur ein logischer Negationsoperator vordefiniert:

bool operator !(bool x);

Dieser Operator errechnet die logische Negation des Operanden: Wenn der Operand true ist, ist das Ergebnis false. Wenn der Operand false ist, ist das Ergebnis true.

### Bitweiser Komplementoperator

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird auf Operationen der Form ~x angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Der Operand wird in den Parametertyp des ausgewählten Operators konvertiert, wobei der Ergebnistyp dem Rückgabetyp des Operators entspricht. Folgende bitweise Komplementoperatoren sind vordefiniert:

int operator ~(int x);  
uint operator ~(uint x);  
long operator ~(long x);  
ulong operator ~(ulong x);

Das Operationsergebnis zu diesen Operatoren entspricht dem bitweisen Komplement von x.

Der Enumerationstyp E enthält implizit den folgenden bitweisen Komplementoperator:

E operator ~(E x);

Das Ergebnis der Auswertung von ~x (wobei x ein Ausdruck eines Enumerationstyps E mit einem zugrundeliegenden Typ U ist) entspricht genau dem Auswertungsergebnis von (E)(~(U)x), mit dem Unterschied, dass die Konvertierung in E immer wie in einem unchecked-Kontext ausgeführt wird (§7.6.12).

### Präfix-Inkrement- und -Dekrementoperatoren

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

Der Operand einer Präfix-Inkrementoperation oder eine Präfix-Dekrementoperation muss ein Ausdruck sein, der als Variable, Eigenschaftenzugriff oder Indexerzugriff klassifiziert ist. Das Ergebnis der Operation ist ein Wert vom Typ des Operanden.

Wenn der Operand einer Präfix-Inkrementoperation oder Präfix-Dekrementoperation eine Eigenschaft oder ein Indexzugriff ist, muss die Eigenschaft oder der Indexer über einen get-Accessor und einen set-Accessor verfügen. Ist dies nicht der Fall, tritt während der Bindung ein Fehler auf.

Die Überladungsauflösung unärer Operatoren (§7.3.3) wird angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die vordefinierten Operatoren ++ und -- bestehen für die folgenden Typen: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal und alle Enumerationstypen. Die vordefinierten Operatoren ++ geben den Wert zurück, der entsteht, wenn zum Operanden 1 addiert wird. Die vordefinierten Operatoren -- geben den Wert zurück, der entsteht, wenn vom Operanden 1 subtrahiert wird. Wenn in einem checked-Kontext das Ergebnis dieser Addition oder Subtraktion außerhalb des Bereichs für den Ergebnistyp liegt und der Ergebnistyp ein Ganzzahl- oder Enumerationstyp ist, wird System.OverflowException ausgelöst.

Die Laufzeitverarbeitung einer Präfix-Inkrementoperation oder einer Präfix-Dekrementoperation der Form ++x oder --x besteht aus den folgenden Schritten:

* Bei Klassifizierung von x als Variable:
* x wird zur Erzeugung der Variablen ausgewertet.
* Der ausgewählte Operator wird mit dem Wert von x als Argument aufgerufen.
* Der vom Operator zurückgegebene Wert wird in dem Speicherort gespeichert, der durch die Auswertung von x vorgegeben wird.
* Der vom Operator zurückgegebene Wert ist das Ergebnis der Operation.
* Bei Klassifizierung von x als Eigenschaft oder Indexzugriff:
* Der mit x verknüpfte Instanzausdruck (wenn x nicht static ist) sowie die mit x verknüpfte Argumentenliste (wenn x ein Indexzugriff ist) werden ausgewertet und die Ergebnisse in den darauffolgenden Accessoraufrufen get und set verwendet.
* Der get-Accessor von x wird aufgerufen.
* Der ausgewählte Operator wird mit dem vom get-Accessor zurückgegebenen Wert als Argument aufgerufen.
* Der set-Accessor von x wird mit dem vom Operator zurückgegebenen Wert als value-Argument aufgerufen.
* Der vom Operator zurückgegebene Wert ist das Ergebnis der Operation.

Die Operatoren ++ und -- unterstützen auch die Postfixnotation (§7.6.9). In der Regel entspricht das Ergebnis von x++ oder x-- dem Wert von x vor der Operation, wogegen das Ergebnis von ++x oder --x dem Wert von x nach der Operation entspricht. In beiden Fällen hat x nach der Operation denselben Wert.

Die Implementierung von operator ++ oder operator -- kann mithilfe der Postfix- oder Präfixnotation aufgerufen werden. Einzelne Operatorimplementierungen für beide Notationen sind nicht möglich.

### Umwandlungsausdrücke

Eine cast-expression wird verwendet, um einen Ausdruck explizit in einen bestimmten Typ zu konvertieren.

cast-expression:  
( type ) unary-expression

Eine cast-expression der Form (T)E, wobei T ein type und E eine unary-expression ist, führt eine explizite Konvertierung (§6.2) des Werts von E in den Typ T durch. Wenn keine explizite Konvertierung von E in T vorhanden ist, wird ein Kompilierungsfehler ausgelöst. Andernfalls ist das Ergebnis der von der expliziten Konvertierung erzeugte Wert. Das Ergebnis wird stets als Wert klassifiziert, auch wenn E eine Variable bezeichnet.

Die Grammatik einer cast-expression impliziert bestimmte syntaktische Mehrdeutigkeiten. Der Ausdruck (x)–y kann als eine cast-expression (Umwandlung von –y in den Typ x) oder als eine mit einer parenthesized-expression kombinierte additive-expression (zur Berechnung des Werts von x – y)) interpretiert werden.

Für die Auflösung der Uneindeutigkeit der cast-expression gilt folgende Regel: Eine Folge von einem oder mehreren, in Klammern eingeschlossenen tokens (§2.3.3) wird nur dann als Anfang einer cast-expression interpretiert, wenn mindestens eine der folgenden Bedingungen wahr ist:

* Die Folge der Token entspricht der korrekten Grammatik eines type, nicht jedoch der einer expression.
* Die Folge der Token entspricht der korrekten Grammatik eines type und das direkt auf die schließende Klammer folgende Token ist das Token "~", das Token "!", das Token "(", identifier (§2.4.1), a literal (§2.4.4) oder ein beliebiges keyword (§2.4.3) außer as und is.

Der Begriff „korrekte Grammatik“ besagt, dass die Reihe von Token der jeweiligen grammatischen Bildung entsprechen muss. Die tatsächliche Bedeutung der einzelnen Bezeichner wird dabei nicht betrachtet. Beispiel: Gegeben sind die Bezeichner x und y, damit bildet x.y eine korrekte Grammatik für einen Typ, auch wenn x.y keinen Typ bezeichnet.

Aus den Regeln zur Sicherstellung der Eindeutigkeit folgt: Wenn x und y Bezeichner sind, dann sind (x)y, (x)(y) und (x)(-y) cast-expressions, (x)-y dagegen nicht, auch wenn x einen Typ bezeichnet. Wenn jedoch x ein Schlüsselwort ist, das einen vordefinierten Typ bezeichnet (z. B. int), handelt es sich bei allen vier Formen um cast-expressions (weil ein solches Schlüsselwort an sich keinen Ausdruck bilden kann).

### Await-Ausdrücke

Mit dem Await-Operator wird die Auswertung der einschließenden Async-Funktion angehalten, bis der durch den Operanden dargestellte asynchrone Vorgang abgeschlossen ist.

await-expression:  
await unary-expression

Eine await-expression ist nur im Text einer Async-Funktion (§10.14) zulässig. Innerhalb der nächsten einschließenden Async-Funktion darf eine await-expression nicht an den folgenden Stellen auftreten:

* Innerhalb einer geschachtelten (nicht asynchronen) anonymen Funktion
* In einem catch- oder finally-Block eines try-statement
* Innerhalb des Blocks eines lock-statement
* In einem unsicheren Kontext

Beachten Sie, dass eine await-expression an den meisten Stellen innerhalb einer query-expression nicht auftreten darf, da diese zur Verwendung von nicht asynchronen Lambda-Ausdrücken syntaktisch umgewandelt wird.

Innerhalb einer Async-Funktion darf await nicht als Bezeichner verwendet werden. Daher gibt es keine syntaktische Mehrdeutigkeit zwischen Await-Ausdrücken und verschiedenen Ausdrücken mit Bezeichnern. Außerhalb von Async-Funktionen fungiert await als normaler Bezeichner.

Der Operand einer await-expression wird als Aufgabe bezeichnet. Dadurch wird eine asynchrone Operation dargestellt, die zum Zeitpunkt der Auswertung der await-expression abgeschlossen oder auch nicht abgeschlossen wird. Der Zweck des Await-Operators besteht darin, die Ausführung der einschließenden Async-Funktion anzuhalten, bis die erwartete Aufgabe abgeschlossen ist, und dann das Ergebnis abzurufen.

#### Awaitable-Ausdrücke

Die Aufgabe eines Await-Ausdrucks muss awaitable sein. Ein Ausdruck *t* ist awaitable, wenn eine der folgenden Bedingungen zutrifft:

* *t* is of compile time type dynamic
* *t* has an accessible instance or extension method called GetAwaiter with no parameters and no type parameters, and a return type *A* for which all of the following hold:
* *A* implementiert die Schnittstelle System.Runtime.CompilerServices.INotifyCompletion (im Folgenden der Kürze wegen als INotifyCompletion bezeichnet).
* *A* verfügt über eine zugängliche, lesbare Instanzeigenschaft IsCompleted vom Typ bool.
* *A* verfügt über eine zugängliche Instanzmethode GetResult ohne Parameter oder Typparameter.

Der Zweck der GetAwaiter-Methode besteht darin, einen Awaiter für die Aufgabe abzurufen. Der Typ *A* wird als Awaiter-Typ für den Await-Ausdruck bezeichnet.

Der Zweck der IsCompleted-Eigenschaft besteht darin zu bestimmen, ob die Aufgabe bereits abgeschlossen ist. In diesem Fall muss die Auswertung nicht angehalten werden.

Der Zweck der INotifyCompletion.OnCompleted-Methode besteht darin, eine "Fortsetzung" für die Aufgabe zu registrieren, d. h. einen Delegaten (vom Typ System.Action), der bei Abschluss der Aufgabe aufgerufen wird.

Der Zweck der GetResult-Methode besteht darin, das Ergebnis der Aufgabe nach deren Abschluss abzurufen. Dieses Ergebnis kann kein erfolgreicher Abschluss sein, eventuell mit einem Ergebniswert, oder es kann sich um eine Ausnahme handeln, die von der GetResult-Methode ausgelöst wird.

#### Klassifizierungen von Await-Ausdrücken

Der Ausdruck await *t* wird auf dieselbe Weise klassifiziert wie der Ausdruck (*t*).GetAwaiter().GetResult(). Wenn daher der Rückgabetyp von GetResult void ist, wird die await-expression als "nothing" klassifiziert. Wenn der Rückgabetyp *T* nicht "void" ist, wird die await-expression als Wert vom Typ *T* klassifiziert.

#### Laufzeitauswertung von Await-Ausdrücken

Zur Laufzeit wird der Ausdruck await *t* wie folgt ausgewertet:

* Ein Awaiter *a* wird durch Auswertung des Ausdrucks (*t*).GetAwaiter() abgerufen.
* Ein bool *b* wird durch Auswertung des Ausdrucks (*a*).IsCompleted abgerufen.
* Wenn *b* den Wert false hat, ist die Auswertung davon abhängig, ob *a* die Schnittstelle System.Runtime.CompilerServices.ICriticalNotifyCompletion implementiert (im Folgenden der Kürze wegen als ICriticalNotifyCompletion bezeichnet). Die Überprüfung wird während der Bindung vorgenommen, bzw. zur Laufzeit, wenn *a* den Kompilierungstyp dynamic aufweist, und andernfalls während der Kompilierung. *r* kennzeichnet den Wiederaufnahmedelegaten (§10.14):
* Wird ICriticalNotifyCompletion von *a* nicht implementiert, wird der Ausdruck   
  (*a* as (INotifyCompletion)).OnCompleted(*r*) ausgewertet.
* Wird ICriticalNotifyCompletion von *a* nicht implementiert, wird der Ausdruck   
  (*a* as (ICriticalNotifyCompletion)).UnsafeOnCompleted(*r*) ausgewertet.
* Die Auswertung wird angehalten, und die Steuerung wird an den aktuellen Aufrufer der Async-Funktion zurückgegeben.
* Entweder direkt danach (wenn *b* true war) oder beim späteren Aufruf des Wiederaufnahmedelegaten (wenn *b* false war) wird der Ausdruck (*a*).GetResult() ausgewertet. Wenn ein Wert zurückgegeben wird, ist dieser Wert das Ergebnis der *await-expression*. Andernfalls ist das Ergebnis "nothing".

Die Implementierung der Schnittstellenmethoden INotifyCompletion.OnCompleted und ICriticalNotifyCompletion.UnsafeOnCompleted durch einen Awaiter sollte dazu führen, dass der Delegat *r* höchstens einmal aufgerufen wird. Andernfalls ist das Verhalten einschließenden Async-Funktion nicht definiert.

## Arithmetische Operatoren

Die Operatoren \*, /, %, + und – werden arithmetische Operatoren genannt.

multiplicative-expression:  
unary-expression  
multiplicative-expression \* unary-expression  
multiplicative-expression / unary-expression  
multiplicative-expression % unary-expression

additive-expression:  
multiplicative-expression  
additive-expression + multiplicative-expression  
additive-expression – multiplicative-expression

Wenn der Operand eines arithmetischen Operators den Kompilierungstyp dynamic aufweist, wird der Ausdruck dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Ausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps derjenigen Operanden, die den Kompilierungstyp dynamic aufweisen.

### Multiplikationsoperator

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x \* y angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Multiplikationsoperatoren werden im Folgenden aufgeführt: Alle Operatoren berechnen das Produkt aus x und y.

* Ganzzahlige Multiplikation:

int operator \*(int x, int y);  
uint operator \*(uint x, uint y);  
long operator \*(long x, long y);  
ulong operator \*(ulong x, ulong y);

Wenn in einem checked-Kontext das Produkt außerhalb des Bereichs für den Ergebnistyp liegt, wird System.OverflowException ausgelöst. In einem unchecked-Kontext werden keine Überläufe gemeldet, und alle signifikanten höherwertigen Bits außerhalb des Bereichs des Ergebnistyps werden verworfen.

* Multiplikation mit Gleitkommawerten:

float operator \*(float x, float y);  
double operator \*(double x, double y);

Das Produkt wird entsprechend den arithmetischen Regeln nach IEEE 754 berechnet. In der folgenden Tabelle werden die Ergebnisse aller möglichen Kombinationen aus endlichen Werten ungleich 0, Nullwerten, unendlichen Werten und NaNs aufgeführt. In der Tabelle sind x und y positive endliche Werte, und z ist das Ergebnis aus x \* y. Wenn das Ergebnis für den Zieltyp zu groß ist, ist z unendlich. Wenn das Ergebnis für den Zieltyp zu klein ist, ist z gleich 0 (null).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +0 | –0 | +∞ | –∞ | NaN |
| –x | –z | +z | –0 | +0 | –∞ | +∞ | NaN |
| +0 | +0 | –0 | +0 | –0 | NaN | NaN | NaN |
| –0 | –0 | +0 | –0 | +0 | NaN | NaN | NaN |
| +∞ | +∞ | –∞ | NaN | NaN | +∞ | –∞ | NaN |
| –∞ | –∞ | +∞ | NaN | NaN | –∞ | +∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Dezimale Multiplikation:

decimal operator \*(decimal x, decimal y);

Wenn der Ergebniswert für die Darstellung im decimal-Format zu groß ist, wird eine System.OverflowException ausgelöst. Wenn der Ergebniswert für die Darstellung im decimal-Format zu klein ist, ist das Ergebnis gleich 0. Die Dezimalstellenanzahl des Ergebnisses vor dem Runden entspricht der Summe der Dezimalstellen beider Operanden.

Die dezimale Multiplikation ist äquivalent zur Verwendung des Multiplikationsoperators vom Typ System.Decimal.

### Divisionsoperator

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x / y angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Divisionsoperatoren werden im Folgenden aufgeführt: Alle Operatoren berechnen den Quotienten aus x und y.

* Ganzzahlige Division:

int operator /(int x, int y);  
uint operator /(uint x, uint y);  
long operator /(long x, long y);  
ulong operator /(ulong x, ulong y);

Wenn der Wert des rechten Operanden gleich 0 (null) ist, wird eine System.DivideByZeroException ausgelöst.

Die Division rundet das Ergebnis ab. Der absolute Ergebniswert ist die größtmögliche ganze Zahl, die kleiner oder gleich dem absoluten Wert des Quotienten aus den beiden Operanden ist. Das Ergebnis ist 0 oder positiv, wenn beide Operanden dasselbe Vorzeichen haben, bzw. null oder negativ, wenn beide Operanden gegensätzliche Vorzeichen haben.

Wenn der linke Operand der in int oder long kleinste darstellbare Wert ist und der rechte Operand –1 beträgt, tritt ein Überlauf auf. In einem checked-Kontext führt dies zum Auslösen einer System.ArithmeticException (oder einer entsprechenden Unterklasse). In einem unchecked-Kontext richtet sich nach der Implementierung, ob eine System.ArithmeticException (oder eine entsprechende Unterklasse) ausgelöst wird oder der Überlauf nicht gemeldet wird und der Ergebniswert der Wert des linken Operanden ist.

* Division mit Gleitkommawerten:

float operator /(float x, float y);  
double operator /(double x, double y);

Der Quotient wird entsprechend den arithmetischen Regeln nach IEEE 754 berechnet. In der folgenden Tabelle werden die Ergebnisse aller möglichen Kombinationen aus endlichen Werten ungleich 0, Nullwerten, unendlichen Werten und NaNs aufgeführt. In der Tabelle sind x und y positive endliche Werte, und z ist das Ergebnis aus x / y. Wenn das Ergebnis für den Zieltyp zu groß ist, ist z unendlich. Wenn das Ergebnis für den Zieltyp zu klein ist, ist z gleich 0 (null).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | –z | +∞ | –∞ | +0 | –0 | NaN |
| –x | –z | +z | –∞ | +∞ | –0 | +0 | NaN |
| +0 | +0 | –0 | NaN | NaN | +0 | –0 | NaN |
| –0 | –0 | +0 | NaN | NaN | –0 | +0 | NaN |
| +∞ | +∞ | –∞ | +∞ | –∞ | NaN | NaN | NaN |
| –∞ | –∞ | +∞ | –∞ | +∞ | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Dezimale Division:

decimal operator /(decimal x, decimal y);

Wenn der Wert des rechten Operanden gleich 0 (null) ist, wird eine System.DivideByZeroException ausgelöst. Wenn der Ergebniswert für die Darstellung im decimal-Format zu groß ist, wird eine System.OverflowException ausgelöst. Wenn der Ergebniswert für die Darstellung im decimal-Format zu klein ist, ist das Ergebnis gleich 0. Die Dezimalstellen des Ergebnisses sind die kleinsten Dezimalstellen in dem Ergebnis, das dem darstellbaren Dezimalwert entspricht und dem wahren mathematischen Ergebnis am nächsten kommt.

Die dezimale Division ist äquivalent zur Verwendung des Divisionsoperators vom Typ System.Decimal.

### Restoperator

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x % y angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Restwertoperatoren werden im Folgenden aufgeführt: Alle Operatoren berechnen den Restwert der Division aus x und y.

* Ganzzahliger Restwert:

int operator %(int x, int y);  
uint operator %(uint x, uint y);  
long operator %(long x, long y);  
ulong operator %(ulong x, ulong y);

Das Ergebnis aus x % y ist der Wert, der sich aus x – (x / y) \* y ergibt. Wenn y gleich 0 ist, wird eine System.DivideByZeroException-Ausnahme ausgelöst.

Wenn der linke Operand der kleinste int-Wert oder long-Wert ist und der rechte Operand -1 ist, wird eine System.OverflowException ausgelöst. Zu keiner Zeit wird durch x % y eine Ausnahme ausgelöst, wenn x / y keine Ausnahme auslöst.

* Gleitkommarest:

float operator %(float x, float y);  
double operator %(double x, double y);

In der folgenden Tabelle werden die Ergebnisse aller möglichen Kombinationen aus endlichen Werten ungleich 0, Nullwerten, unendlichen Werten und NaNs aufgeführt. In der Tabelle sind x und y positive endliche Werte. z ist das Ergebnis von x % y und wird als x – n \* y berechnet, wobei n die größte mögliche ganze Zahl ist, die kleiner oder gleich x / y ist. Diese Methode der Restwertberechnung ist analog zu der für ganzzahlige Operanden, unterscheidet sich jedoch von der Definition in IEEE 754 (wobei n dem ganzzahligen Wert entspricht, der dem Ergebnis von x / y am nächsten liegt).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | +y | –y | +0 | –0 | +∞ | –∞ | NaN |
| +x | +z | +z | NaN | NaN | x | x | NaN |
| –x | –z | –z | NaN | NaN | –x | –x | NaN |
| +0 | +0 | +0 | NaN | NaN | +0 | +0 | NaN |
| –0 | –0 | –0 | NaN | NaN | –0 | –0 | NaN |
| +∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| –∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Dezimaler Restwert:

decimal operator %(decimal x, decimal y);

Wenn der Wert des rechten Operanden gleich 0 (null) ist, wird eine System.DivideByZeroException ausgelöst. Die Dezimalstellen des Ergebnisses entsprechen vor dem Runden den höheren Dezimalstellen beider Operanden, und das Vorzeichen des Ergebnisses entspricht, falls es ungleich 0 ist, dem Vorzeichen von x.

Der dezimale Restwert ist äquivalent zur Verwendung des Restwertoperators vom Typ System.Decimal.

### Additionsoperator

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x + y angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Additionsoperatoren werden im Folgenden aufgeführt: Für numerische und Enumerationstypen berechnen die vordefinierten Additionsoperatoren die Summe aus beiden Operanden. Wenn mindestens ein Operand den Typ string aufweist, verketten die vordefinierten Additionsoperatoren die Darstellung der Operandenzeichenfolge.

* Ganzzahlige Addition:

int operator +(int x, int y);  
uint operator +(uint x, uint y);  
long operator +(long x, long y);  
ulong operator +(ulong x, ulong y);

Wenn in einem checked-Kontext die Summe außerhalb des Bereichs für den Ergebnistyp liegt, wird System.OverflowException ausgelöst. In einem unchecked-Kontext werden keine Überläufe gemeldet, und alle signifikanten höherwertigen Bits außerhalb des Bereichs des Ergebnistyps werden verworfen.

* Addition von Gleitkommazahlen:

float operator +(float x, float y);  
double operator +(double x, double y);

Die Summe wird entsprechend den arithmetischen Regeln nach IEEE 754 berechnet. In der folgenden Tabelle werden die Ergebnisse aller möglichen Kombinationen aus endlichen Werten ungleich 0, Nullwerten, unendlichen Werten und NaNs aufgeführt. In der Tabelle sind x und y endliche Werte ungleich 0, und z ist das Ergebnis aus x + y. Wenn x und y gleiche Werte mit gegensätzlichen Vorzeichen haben, ist z positiv 0. Wenn x + y zur Darstellung im Zieltyp zu groß ist, ist z unendlich und hat dasselbe Vorzeichen wie x + y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | +∞ | –∞ | NaN |
| +0 | y | +0 | +0 | +∞ | –∞ | NaN |
| –0 | y | +0 | –0 | +∞ | –∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | +∞ | NaN | NaN |
| –∞ | –∞ | –∞ | –∞ | NaN | –∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Dezimale Addition:

decimal operator +(decimal x, decimal y);

Wenn der Ergebniswert für die Darstellung im decimal-Format zu groß ist, wird eine System.OverflowException ausgelöst. Die Dezimalstellenanzahl des Ergebnisses vor dem Runden entspricht der höheren Dezimalstellenanzahl eines der beiden Operanden.

Die dezimale Addition ist äquivalent zur Verwendung des Additionsoperators vom System.Decimal.

* Enumerationsaddition. Alle Enumerationstypen enthalten implizit die folgenden vordefinierten Operatoren, wobei E der Enumerationstyp und U der zugrunde liegende Typ von E ist.

E operator +(E x, U y);  
E operator +(U x, E y);

Zur Laufzeit werden diese Operatoren genau in der Form (E)((U)x + (U)y) ausgewertet.

* Zeichenfolgenverkettung:

string operator +(string x, string y);  
string operator +(string x, object y);  
string operator +(object x, string y);

Diese Überladungen des binären +-Operators führen eine Zeichenfolgenverkettung aus. Wenn ein Operand der Zeichenfolgenverkettung gleich null ist, wird eine leere Zeichenfolge ersetzt. Andernfalls werden alle Argumente, die keine Zeichenfolge darstellen, in die Zeichenfolgendarstellung konvertiert, indem die virtuelle ToString-Methode aufgerufen wird, die vom Typ object vererbt wurde. Wenn ToString den Wert null zurückgibt, wird eine leere Zeichenfolge ersetzt.

using System;

class Test  
{  
 static void Main() {  
 string s = null;  
 Console.WriteLine("s = >" + s + "<"); // displays s = ><  
 int i = 1;  
 Console.WriteLine("i = " + i); // displays i = 1  
 float f = 1.2300E+15F;  
 Console.WriteLine("f = " + f); // displays f = 1.23E+15  
 decimal d = 2.900m;  
 Console.WriteLine("d = " + d); // displays d = 2.900  
 }  
}

Das Ergebnis des Operators der Zeichenfolgenverkettung ist eine Zeichenfolge, die aus den Zeichen des linken Operanden gefolgt von den Zeichen des rechten Operanden besteht. Der Operator der Zeichenfolgenverkettung gibt stets einen Wert ungleich null aus. Wenn nicht genügend Speicher für die Ergebniszeichenfolge reserviert werden kann, wird ggf. System.OutOfMemoryException ausgelöst.

* Delegatkombination. Alle Delegattypen enthalten implizit den folgenden vordefinierten Operator, wobei D dem Delegattyp entspricht:

D operator +(D x, D y);

Der binäre +-Operator führt eine Delegatkombination aus, wenn einer oder beide Operanden vom Delegattyp D sind. (Wenn die Operanden unterschiedliche Delegattypen aufweisen, tritt ein Bindungsfehler auf.) Wenn der erste Operand gleich null ist, ist das Operationsergebnis der Wert des zweiten Operanden (auch wenn der Operand ebenfalls gleich null ist). Andernfalls gilt: Wenn der zweite Operand gleich null ist, ist das Operationsergebnis mit dem Wert des ersten Operanden identisch. In allen anderen Fällen ist das Operationsergebnis eine neue Delegatinstanz, die bei ihrem Aufruf zuerst den ersten Operanden und dann den zweiten Operanden aufruft. Beispiele zu Delegatkombinationen finden Sie in §7.8.5 und §15.4. Da System.Delegate kein Delegattyp ist, wird operator + nicht für ihn definiert.

### Subtraktionsoperator

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x – y angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Subtraktionsoperatoren werden im Folgenden aufgeführt: Alle Operatoren subtrahieren dabei y von x.

* Ganzzahlige Subtraktion:

int operator –(int x, int y);  
uint operator –(uint x, uint y);  
long operator –(long x, long y);  
ulong operator –(ulong x, ulong y);

Wenn in einem checked-Kontext die Differenz außerhalb des Bereichs für den Ergebnistyp liegt, wird System.OverflowException ausgelöst. In einem unchecked-Kontext werden keine Überläufe gemeldet, und alle signifikanten höherwertigen Bits außerhalb des Bereichs des Ergebnistyps werden verworfen.

* Subtraktion von Gleitkommazahlen:

float operator –(float x, float y);  
double operator –(double x, double y);

Die Differenz wird entsprechend den arithmetischen Regeln nach IEEE 754 berechnet. In der folgenden Tabelle werden die Ergebnisse aller möglichen Kombinationen aus endlichen Werten ungleich 0, Nullwerten, unendlichen Werten und NaNs aufgeführt. In der Tabelle sind x und y endliche Werte ungleich 0, und z ist das Ergebnis aus x – y. Falls x und y gleich sind, ist z positiv 0. Wenn x – y zur Darstellung im Zieltyp zu groß ist, ist z unendlich und hat dasselbe Vorzeichen wie x – y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | y | +0 | –0 | +∞ | –∞ | NaN |
| x | z | x | x | –∞ | +∞ | NaN |
| +0 | –y | +0 | +0 | –∞ | +∞ | NaN |
| –0 | –y | –0 | +0 | –∞ | +∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | NaN | +∞ | NaN |
| –∞ | –∞ | –∞ | –∞ | –∞ | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

* Dezimale Subtraktion:

decimal operator –(decimal x, decimal y);

Wenn der Ergebniswert für die Darstellung im decimal-Format zu groß ist, wird eine System.OverflowException ausgelöst. Die Dezimalstellenanzahl des Ergebnisses vor dem Runden entspricht der höheren Dezimalstellenanzahl eines der beiden Operanden.

Die dezimale Subtraktion ist äquivalent zur Verwendung des Subtraktionsoperators vom Typ System.Decimal.

* Enumerationssubtraktion. Alle Enumerationstypen enthalten implizit der folgende vordefinierte Operator, wobei E der Enumerationstyp und U der zugrunde liegende Typ von E ist.

U operator –(E x, E y);

Dieser Operator wird genau in der Form (U)((U)x – (U)y) ausgewertet. Das bedeutet, der Operator berechnet die Differenz aus den Ordinalwerten von x und y, und der Ergebnistyp ist der zugrunde liegende Enumerationstyp.

E operator –(E x, U y);

Dieser Operator wird genau in der Form (E)((U)x – y) ausgewertet. D. h. der Operator subtrahiert einen Wert von dem zugrunde liegenden Enumerationstyp und erhält so einen Enumerationswert.

* Delegatentfernung. Alle Delegattypen enthalten implizit den folgenden vordefinierten Operator, wobei D dem Delegattyp entspricht:

D operator –(D x, D y);

Der binäre –-Operator führt eine Delegatentfernung aus, wenn einer oder beide Operanden vom Delegattyp D sind. Wenn die Operanden unterschiedliche Delegattypen aufweisen, tritt ein Bindungsfehler auf. Wenn der erste Operand gleich null ist, ist auch das Operationsergebnis null. Andernfalls gilt: Wenn der zweite Operand gleich null ist, ist das Operationsergebnis mit dem Wert des ersten Operanden identisch. Ist dies nicht der Fall, repräsentieren beide Operanden Aufruflisten (§15.1), die mindestens einen Eintrag enthalten. Das Ergebnis ist eine neue Aufrufliste, die aus der Liste des ersten Operanden besteht, aus der die Einträge des zweiten Operanden entfernt sind, vorausgesetzt, die Liste des zweiten Operanden ist eine zusammenhängende untergeordnete Teilmenge des ersten Operanden. (Um die Gleichheit einer Teilliste zu bestimmen, werden die entsprechenden Einträge wie beim Delegatgleichheitsoperator (§7.10.8) verglichen.) In allen anderen Fällen ist das Ergebnis mit dem Wert des linken Operanden identisch. Bei diesem Vorgang wird keine der Operandenlisten verändert. Wenn die Liste des zweiten Operanden mit mehreren Teillisten zusammenhängender Einträge der ersten Operandenliste übereinstimmt, wird die am weitesten rechts stehende übereinstimmende Teilliste zusammenhängender Einträge entfernt. Wenn durch das Entfernen eine leere Liste entsteht, ist das Ergebnis gleich null. Beispiel:

delegate void D(int x);

class C  
{  
 public static void M1(int i) { /\* … \*/ }  
 public static void M2(int i) { /\* … \*/ }  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1);  
 D cd2 = new D(C.M2);  
 D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1; // => M1 + M2 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd2; // => M2 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd2; // => M1 + M1

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd2 + cd1; // => M1 + M2

cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1  
 cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1  
 }  
}

## Verschiebeoperatoren

Die Operatoren << und >> werden zur Ausführung von Bitverschiebungsoperationen verwendet.

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

Wenn der Operand einer shift-expression den Kompilierungstyp dynamic aufweist, wird der Ausdruck dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Ausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps derjenigen Operanden, die den Kompilierungstyp dynamic aufweisen.

Die Überladungsauflösung binärer Operatoren (§7.3.4) wird auf Operationen der Form x << count or x >> count angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Beim Deklarieren eines überladenen Schiebeoperators muss der Typ des ersten Operanden stets die Klasse oder Struktur sein, welche die Operatordeklaration enthält. Der Typ des zweiten Operanden muss immer int sein.

Die vordefinierten Schiebeoperatoren werden im Folgenden aufgeführt:

* Verschiebung nach links:

int operator <<(int x, int count);  
uint operator <<(uint x, int count);  
long operator <<(long x, int count);  
ulong operator <<(ulong x, int count);

Der <<-Operator verschiebt x um die ermittelte Bitanzahl nach links (siehe folgende Beschreibung).

Die höherwertigen Bits außerhalb des Bereichs des Ergebnistyps von x werden verworfen, die verbleibenden Bits nach links verschoben und die geringerwertigen leeren Bitpositionen auf 0 gesetzt.

* Verschiebung nach rechts:

int operator >>(int x, int count);  
uint operator >>(uint x, int count);  
long operator >>(long x, int count);  
ulong operator >>(ulong x, int count);

Der >>-Operator verschiebt x um die ermittelte Bitanzahl nach rechts (siehe folgende Beschreibung).

Wenn x vom Typ int oder long ist, werden die geringerwertigen Bits von x verworfen, die verbleibenden Bits nach rechts verschoben und die höherwertigen leeren Bitpositionen auf 0 gesetzt, wenn x nicht negativ und bei negativem x auf eins gesetzt ist.

Wenn x vom Typ uint oder ulong ist, werden die geringerwertigen Bits von x verworfen, die verbleibenden Bits nach rechts verschoben und die höherwertigen leeren Bitpositionen auf 0 gesetzt.

Für die vordefinierten Operatoren wird die Anzahl der zu verschiebenden Bits wie folgt berechnet:

* Wenn x vom Typ int oder uint ist, wird die Verschiebungszahl durch die fünf geringerwertigen Bits von count bestimmt. Die Verschiebungszahl errechnet sich also aus count & 0x1F.
* Wenn x vom Typ long oder ulong ist, wird die Verschiebungszahl durch die sechs geringerwertigen Bits von count bestimmt. Die Verschiebungszahl errechnet sich also aus count & 0x3F.

Wenn die Berechnung der Verschiebungszahl 0 ergibt, geben die Schiebeoperatoren einfach den Wert von x zurück.

Schiebeoperatoren verursachen keinen Überlauf und ergeben in den checked- und unchecked-Kontexten dieselben Ergebnisse.

Wenn der linke Operand des >>-Operators ein vorzeichenbehafteter Ganzzahltyp ist, führt der Operator eine arithmetische Verschiebung nach rechts aus, wobei der Wert des höchstwertigen Bits (Vorzeichenbit) des Operanden an die höherwertigen leeren Bitpositionen übergeben wird. Wenn der linke Operand des >>-Operators ein vorzeichenloser Ganzzahltyp ist, führt der Operator eine logische Verschiebung nach rechts aus, wobei höherwertige leere Bitpositionen stets auf 0 festgelegt werden. Um die entgegengesetzte Operation zu der vom Operandentyp hergeleiteten auszuführen, können explizite Typumwandlungen verwendet werden. Wenn x z. B. eine Variable vom Typ int ist, führt die Operation unchecked((int)((uint)x >> y)) eine logische Verschiebung von x nach rechts aus.

## Relationale und Typtestoperatoren

Die Operatoren ==, !=, <, >, <=, >=, is und as werden relationale und Typtestoperatoren genannt.

relational-expression:  
shift-expression  
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression  
relational-expression is type  
relational-expression as type

equality-expression:  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression

Der is-Operator wird in §7.10.10 und der as-Operator in §7.10.11 beschrieben.

Die Operatoren ==, !=, <, >, <= und >= sind Vergleichsoperatoren.

Wenn der Operand eines Vergleichsoperators den Kompilierungstyp dynamic aufweist, wird der Ausdruck dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Ausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps derjenigen Operanden, die den Kompilierungstyp dynamic aufweisen.

Auf Operationen der Form x op y, bei denen op ein Vergleichsoperator ist, wird die Überladungsauflösung (§7.3.4) angewendet, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten Vergleichsoperatoren werden in den folgenden Abschnitten beschrieben. Wie aus der folgenden Tabelle hervorgeht, geben alle vordefinierten Vergleichsoperatoren ein Ergebnis vom Typ bool zurück.

|  |  |
| --- | --- |
| **Operation** | **Ergebnis** |
| x == y | true, wenn x gleich y ist, andernfalls false |
| x != y | true, wenn x ungleich y ist, andernfalls false |
| x < y | true, wenn x kleiner als y ist, andernfalls false |
| x > y | true, wenn x größer als y ist, andernfalls false |
| x <= y | true, wenn x kleiner als oder gleich y ist, andernfalls false |
| x >= y | true, wenn x größer als oder gleich y ist, andernfalls false |

### Vergleichsoperatoren für ganze Zahlen

Folgende Operatoren für den Ganzzahlvergleich sind vordefiniert:

bool operator ==(int x, int y);  
bool operator ==(uint x, uint y);  
bool operator ==(long x, long y);  
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);  
bool operator !=(uint x, uint y);  
bool operator !=(long x, long y);  
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);  
bool operator <(uint x, uint y);  
bool operator <(long x, long y);  
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);  
bool operator >(uint x, uint y);  
bool operator >(long x, long y);  
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);  
bool operator <=(uint x, uint y);  
bool operator <=(long x, long y);  
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);  
bool operator >=(uint x, uint y);  
bool operator >=(long x, long y);  
bool operator >=(ulong x, ulong y);

Alle Operatoren vergleichen die numerischen Werte von zwei Ganzzahloperanden und geben einen bool-Wert zurück, der angibt, ob die jeweilige Beziehung true oder false ist.

### Vergleichsoperatoren für Gleitkommazahlen

Folgende Operatoren für den Gleitkommavergleich sind vordefiniert:

bool operator ==(float x, float y);  
bool operator ==(double x, double y);

bool operator !=(float x, float y);  
bool operator !=(double x, double y);

bool operator <(float x, float y);  
bool operator <(double x, double y);

bool operator >(float x, float y);  
bool operator >(double x, double y);

bool operator <=(float x, float y);  
bool operator <=(double x, double y);

bool operator >=(float x, float y);  
bool operator >=(double x, double y);

Die Operatoren vergleichen die Operanden in Übereinstimmung mit den Regeln nach Standard IEEE 754:

* Wenn einer der beiden Operanden NaN ist, ist das Ergebnis für alle Operatoren false (außer für !=, hier lautet das Ergebnis true). Für zwei beliebige Operanden ergibt x != y stets dasselbe Ergebnis wie !(x == y). Wenn jedoch einer oder beide Operanden NaN sind, ergeben die Operatoren <, >, <= und >= nicht dieselben Ergebnisse wie die logische Negation des entgegengesetzten Operators. Wenn z. B. x oder y vom Typ NaN ist, dann ist x < y gleich false, !(x >= y) jedoch true.
* Wenn keiner der Operanden NaN ist, vergleichen die Operatoren die Gleitkommawerte der beiden Operanden hinsichtlich deren Anordnung.

–∞ < –max < ... < –min < –0.0 == +0.0 < +min < ... < +max < +∞

Dabei sind min und max der kleinste bzw. der größte positive und endliche Wert, der im gegebenen Gleitkommaformat dargestellt werden kann. Diese Anordnung wirkt sich folgendermaßen aus:

* Negative und positive Nullen werden als gleichwertig betrachtet.
* Ein negativer unendlicher Wert gilt als kleiner als alle anderen Werte, aber gleichwertig einem anderen negativen unendlichen Wert.
* Ein positiver unendlicher Wert gilt als größer als alle anderen Werte, aber gleichwertig einem anderen positiven unendlichen Wert.

### Vergleichsoperatoren für Dezimalzahlen

Folgende Operatoren für den Dezimalvergleich sind vordefiniert:

bool operator ==(decimal x, decimal y);

bool operator !=(decimal x, decimal y);

bool operator <(decimal x, decimal y);

bool operator >(decimal x, decimal y);

bool operator <=(decimal x, decimal y);

bool operator >=(decimal x, decimal y);

Alle Operatoren vergleichen die numerischen Werte von zwei dezimalen Operanden und geben einen bool-Wert zurück, der angibt, ob die jeweilige Beziehung true oder false ist. Jeder dezimale Vergleich ist äquivalent zur Verwendung des entsprechenden relationalen oder Gleichheitsoperators vom Typ System.Decimal.

### Boolesche Gleichheitsoperatoren

Folgende boolesche Gleichheitsoperatoren sind vordefiniert:

bool operator ==(bool x, bool y);

bool operator !=(bool x, bool y);

Das Ergebnis von == ist true, wenn sowohl x als auch y true sind oder wenn sowohl x als auch y false sind. Andernfalls ist das Ergebnis false.

Das Ergebnis von != ist false, wenn x und y gleich true oder x und y false sind. Andernfalls ist das Ergebnis true. Wenn die Operanden vom Typ bool sind, ergibt der !=-Operator dasselbe Ergebnis wie der ^-Operator.

### Vergleichsoperatoren für Enumerationen

Alle Enumerationstypen stellen implizit die folgenden vordefinierten Vergleichsoperatoren zur Verfügung:

bool operator ==(E x, E y);

bool operator !=(E x, E y);

bool operator <(E x, E y);

bool operator >(E x, E y);

bool operator <=(E x, E y);

bool operator >=(E x, E y);

Das Ergebnis der Auswertung von x op y, wobei x und y Ausdrücke eines Enumerationstyps E mit einem zugrunde liegenden Typ U sind und op einer der Vergleichsoperatoren ist, entspricht genau dem Auswertungsergebnis von ((U)x) op ((U)y). Die Operatoren für den Enumerationsvergleich vergleichen also nur die zugrunde liegenden ganzzahligen Werte der beiden Operanden.

### Gleichheitsoperatoren für Verweistypen

Folgende Gleichheitsoperatoren für Verweistypen sind vordefiniert:

bool operator ==(object x, object y);

bool operator !=(object x, object y);

Die Operatoren geben das Ergebnis zurück, das sich aus dem Überprüfen der beiden Verweise auf Gleichheit oder Ungleichheit ergibt.

Da die vordefinierten Gleichheitsoperatoren für Verweistypen Operanden des Typs object akzeptieren, gelten sie für alle Typen, die keine anwendbaren Member vom Typ operator == und operator != deklarieren. Umgekehrt gilt, dass alle anwendbaren benutzerdefinierten Gleichheitsoperatoren die vordefinierten Gleichheitsoperatoren für Verweistypen effektiv verdecken.

Für vordefinierten Gleichheitsoperatoren muss eine der folgenden Bedingungen zutreffen:

* Beide Operanden sind Werte eines Typs, der als reference-type oder das Literal null bekannt ist. Außerdem muss eine explizite Verweiskonvertierung (§6.2.4) vom Typ des einen Operanden in den Typ des anderen Operanden erfolgen.
* Ein Operand ist ein Wert vom Typ T, wobei T ein type-parameter ist, und der andere Operand ist das Literal null. Darüber hinaus darf T darf keine Werteinschränkung besitzen.

Ein Fehler während der Bindung tritt auf, wenn keine dieser Bedingungen true ist. Diese Regeln wirken sich folgendermaßen aus:

* Die Verwendung der vordefinierten Gleichheitsoperatoren für Verweistypen zum Vergleich von zwei Verweisen, die bei der Bindung bekanntermaßen unterschiedlich sind, verursacht einen Bindungsfehler. Wenn z. B. die Bindungstypen der Operanden zwei Klassentypen A und B sind und weder A noch B aus dem jeweils anderen Typ abgeleitet wurde, können nicht beide Operanden auf dasselbe Objekt verweisen. Darum bewirkt die Operation einen Fehler während der Bindung.
* Die vordefinierten Gleichheitsoperatoren für Verweistypen lassen keinen Vergleich von Werttypoperanden zu. Der Vergleich von Werten dieses Strukturtyps ist also nur möglich, wenn ein Strukturtyp eigene Gleichheitsoperatoren deklariert.
* Die vordefinierten Gleichheitsoperatoren für Verweistypen verursachen für ihre Operanden keine Boxingoperationen. Die Ausführung solcher Boxingoperationen wäre unsinnig, da die Verweise auf neu zugewiesene, mittels Boxing gepackte Instanzen notwendigerweise von allen anderen Verweisen abweichen würden.
* Falls ein Operand eines Typparametertyps T mit null verglichen wird und der Laufzeittyp von T ein Werttyp ist, ist das Ergebnis des Vergleichs false.

Im folgenden Beispiel wird überprüft, ob das Argument eines nicht eingeschränkten Typparametertyps null ist.

class C<T>  
{  
 void F(T x) {  
 if (x == null) throw new ArgumentNullException();  
 ...  
 }  
}

Das Konstrukt x == null ist zulässig, obwohl T einen Werttyp darstellen könnte, und das Ergebnis ist einfach als false definiert, wenn T ein Werttyp ist.

Bei Operationen der Form x == y oder x != y, für die ein anwendbarer operator == oder operator != vorhanden ist, wählen die Überladungsauflösungsregeln eines Operators (§7.3.4) anstelle des vordefinierten Gleichheitsoperators für Verweistypen diesen Operator aus. Es ist jedoch immer möglich, den vordefinierten Gleichheitsoperator für Verweistypen auszuwählen. Dafür müssen einer oder beide Operanden in den Typ object umgewandelt werden. In dem Beispiel

using System;

class Test  
{  
 static void Main() {  
 string s = "Test";  
 string t = string.Copy(s);  
 Console.WriteLine(s == t);  
 Console.WriteLine((object)s == t);  
 Console.WriteLine(s == (object)t);  
 Console.WriteLine((object)s == (object)t);  
 }  
}

erzeugt die Ausgabe

True  
False  
False  
False

Die Variablen s und t verweisen auf zwei unterschiedliche string-Instanzen mit denselben Zeichen. Der erste Vergleich erzeugt True, da der vordefinierte Gleichheitsoperator für Zeichenfolgen (§7.10.7) ausgewählt wird, wenn beide Operanden vom Typ string sind. Alle restlichen Vergleiche ergeben False, da der vordefinierte Gleichheitsoperator für Verweistypen ausgewählt wird, wenn einer oder beide Operanden vom Typ object sind.

Beachten Sie, dass dieses Vorgehen nicht für Werttypen geeignet ist. In dem Beispiel

class Test  
{  
 static void Main() {  
 int i = 123;  
 int j = 123;  
 System.Console.WriteLine((object)i == (object)j);  
 }  
}

gibt False aus, da die Umwandlungen Verweise auf zwei unterschiedliche Instanzen von mittels Boxing gepackten int-Werten erzeugen.

### Gleichheitsoperatoren für Zeichenfolgen

Folgende Gleichheitsoperatoren für Zeichenfolgen sind vordefiniert:

bool operator ==(string x, string y);

bool operator !=(string x, string y);

Zwei string-Werte werden als gleich bewertet, wenn eine der folgenden Bedingungen wahr ist:

* Beide Werte sind null.
* Beide Werte sind von null verschiedene Verweise auf Instanzen von Zeichenfolgen mit identischen Längen und identischen Zeichen in jeder Zeichenposition.

Die Gleichheitsoperatoren für Zeichenfolgen vergleichen nicht die Verweise auf Zeichenfolgen, sondern die Werte von Zeichenfolgen. Wenn zwei unterschiedliche Instanzen von Zeichenfolgen genau dieselbe Anordnung von Zeichen enthalten, sind die Zeichenfolgenwerte gleich, die Verweise jedoch unterschiedlich. Die Gleichheitsoperatoren für Verweistypen können anstelle der Zeichenfolgenwerte die Zeichenfolgenverweise vergleichen (siehe Beschreibung in §7.10.6).

### Gleichheitsoperatoren für Delegaten

Alle Delegattypen enthalten implizit die folgenden vordefinierten Gleichheitsoperatoren:

bool operator ==(System.Delegate x, System.Delegate y);

bool operator !=(System.Delegate x, System.Delegate y);

Zwei Delegatinstanzen werden in folgenden Fällen gleich bewertet:

* Wenn eine der Delegatinstanzen null ist, sind die Delegatinstanzen nur dann gleich, wenn auch die zweite Instanz null ist.
* Wenn die Delegaten unterschiedliche Laufzeittypen besitzen, sind sie niemals gleich.
* Wenn beide Delegatinstanzen eine Aufrufliste (§15.1) mit mindestens zwei Einträgen enthalten, sind die Delegatinstanzen nur dann identisch, wenn ihre Aufruflisten dieselbe Länge haben und die einzelnen Einträge in einer Aufrufliste (wie oben definiert) dem entsprechenden Eintrag in der anderen Aufrufliste auch in ihrer Reihenfolge entsprechen.

Folgende Regeln gelten für die Gleichheit von Einträgen in Aufruflisten:

* Wenn zwei Aufruflisteneinträge auf dieselbe statische Methode verweisen, sind sie gleich.
* Wenn zwei Aufruflisteneinträge auf dieselbe nicht statische Methode in demselben Zielobjekt (das durch die Verweisgleichheitsoperatoren definiert wird) verweisen, sind die Einträge gleich.
* Aufruflisteneinträge, die durch die Auswertung von semantisch identischen anonymous-function-expressions mit derselben (möglicherweise leeren) Gruppe von erfassten äußeren Variableninstanzen erstellt werden, können (müssen jedoch nicht) gleich sein.

### Gleichheitsoperatoren und NULL

Bei den Operatoren == und != darf einer der Operanden ein Wert eines Typs sein, der NULL-Werte zulässt, und der andere das null-Literal. Dies gilt auch, wenn für die Operation weder ein vordefinierter noch ein benutzerdefinierter Operator (heraufgestuft oder nicht) vorhanden ist.

Bei einer Operation der Formen

x == null null == x x != null null != x

mit x als Ausdruck eines Typs, der NULL-Werte zulässt, wird, wenn bei der Auflösung der Operatorüberladung (§7.2.4) kein anwendbarer Operator gefunden werden kann, das Ergebnis stattdessen aus der HasValue-Eigenschaft von x berechnet. Speziell die ersten beiden Formen werden in !x.HasValue übersetzt, und die letzten beiden Formen werden in x.HasValue übersetzt.

### Der is-Operator

Mit dem is-Operator wird auf dynamische Weise überprüft, ob der Laufzeittyp eines Objekts mit einem angegebenen Typ kompatibel ist. Das Ergebnis der Operation E is T, wobei E ein Ausdruck und T ein Typ ist, ist ein boolescher Wert, der angibt, ob E durch eine Verweiskonvertierung, eine Boxingkonvertierung oder eine Unboxingkonvertierung in den Typ T umgewandelt werden kann. Die Operation wird folgendermaßen ausgewertet, nachdem die Typargumente für alle Typparameter ersetzt wurden:

* Wenn E eine anonyme Funktion ist, tritt ein Kompilierungsfehler auf.
* Wenn E eine Methodengruppe oder das null-Literal ist oder wenn der Typ von E ein Verweistyp oder ein Typ ist, der NULL-Werte zulässt, und der Wert von E NULL ist, lautet das Ergebnis false.
* Wenn andernfalls D den dynamischen Typ von E darstellt:
* Wenn der Typ von E ein Verweistyp ist, ist D der Laufzeittyp der Instanz, auf die E verweist.
* Wenn der Typ von E NULL-Werte zulässt, ist D der zugrunde liegende Typ dieses Typs, der NULL-Werte zulässt.
* Wenn der Typ von E ein Werttyp ist, der keine NULL-Werte zulässt, ist D der Typ von E.
* Das Ergebnis der Operation ist folgendermaßen von D und T abhängig:
* Wenn T ein Verweistyp ist, ist das Ergebnis true, wenn D und T von demselben Typ sind, wenn D ein Verweistyp ist und eine implizite Konvertierung von D in T erfolgt, oder wenn D ein Werttyp ist und es eine Boxingkonvertierung von D in T gibt.
* Wenn T ein Typ ist, der NULL-Werte zulässt, ist das Ergebnis true, wenn D der zugrunde liegende Typ von T ist.
* Wenn T ein Typ ist, der keine NULL-Werte zulässt, ist das Ergebnis true, wenn D und T von demselben Typ sind.
* In allen anderen Fällen ist das Ergebnis false.

Beachten Sie, dass benutzerdefinierte Konvertierung vom is-Operator nicht betrachtet werden.

### Der as-Operator

Der as-Operator wird verwendet, um einen Wert in einen festgelegten Verweistyp oder einen Typ, der NULL-Werte zulässt, zu konvertieren. Im Gegensatz zu einem Typumwandlungsausdruck (§7.7.6) löst der as-Operator in keinem Fall eine Ausnahme aus. Wenn die angegebene Konvertierung nicht möglich ist, wird stattdessen der Wert null ausgegeben.

Bei einer Operation der Form E as T, muss E ein Ausdruck und T ein Verweistyp, ein Typparameter, der als Verweistyp gilt, oder ein Typ sein, der NULL-Werte zulässt. Außerdem muss mindestens eine der folgenden Bedingungen zutreffen, andernfalls tritt ein Kompilierungsfehler auf:

* Es gibt eine Identitätskonvertierung (§6.1.1), eine implizite Konvertierung, die NULL-Werte zulässt (§6.1.4), eine implizite Verweiskonvertierung (§6.1.6), eine Boxingkonvertierung (§6.1.7), eine explizite Konvertierung, die NULL-Werte zulässt (§6.2.3), eine explizite Verweiskonvertierung (§6.2.4) oder eine Unboxingkonvertierung (§6.2.5) von E in T.
* Der Typ von E oder T ist ein offener Typ.
* E ist das null-Literal.

Wenn der Kompilierungstyp von E nicht dynamic ist, erzeugt die Operation E as T dasselbe Ergebnis wie

E is T ? (T)(E) : (T)null

mit der Ausnahme, dass E nur einmal ausgewertet wird. Der Compiler wird E as T optimieren, um maximal eine dynamische Typüberprüfung durchführen, und nicht zwei, wie bei dem oben stehenden Ausdruck.

Wenn der Kompilierungstyp von E dynamic ist, wird der as-Operator im Gegensatz zum Umwandlungsoperator nicht dynamisch gebunden (§7.2.2). Deshalb ist die Erweiterung in diesem Fall:

E is T ? (T)(object)(E) : (T)null

Beachten Sie, dass einige Konvertierungen, z. B. benutzerdefinierte, für den as-Operator nicht zulässig sind und stattdessen mit Typumwandlungsausdrücken verwendet werden sollten.

In dem Beispiel

class X  
{

public string F(object o) {  
 return o as string; // OK, string is a reference type  
 }

public T G<T>(object o) where T: Attribute {  
 return o as T; // Ok, T has a class constraint  
 }

public U H<U>(object o) {  
 return o as U; // Error, U is unconstrained   
 }  
}

ist vom Typparameter T von G bekannt, dass er ein Verweistyp ist, da er über eine Klasseneinschränkung verfügt. Für den Typparameter U von H gilt dies jedoch nicht, sodass die Verwendung des as-Operator in H nicht zulässig ist.

## Logische Operatoren

Die Operatoren &, ^ und | werden logische Operatoren genannt.

and-expression:  
equality-expression  
and-expression & equality-expression

exclusive-or-expression:  
and-expression  
exclusive-or-expression ^ and-expression

inclusive-or-expression:  
exclusive-or-expression  
inclusive-or-expression | exclusive-or-expression

Wenn ein Operand eines logischen Operators den Kompilierungstyp dynamic aufweist, wird der Ausdruck dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Ausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps derjenigen Operanden, die den Kompilierungstyp dynamic aufweisen.

Die Überladungsauflösung (§7.3.4) wird auf Operationen der Form x op y angewendet, wobei op ein logischer Operator ist, um eine bestimmte Operatorimplementierung auszuwählen. Die Operanden werden in die Parametertypen des ausgewählten Operators konvertiert, und der Ergebnistyp entspricht dem Rückgabetyp des Operators.

Die vordefinierten logischen Operatoren werden in den folgenden Abschnitten beschrieben.

### Logische Operatoren für ganze Zahlen

Folgende logische Operatoren für ganze Zahlen sind vordefiniert:

int operator &(int x, int y);  
uint operator &(uint x, uint y);  
long operator &(long x, long y);  
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);  
uint operator |(uint x, uint y);  
long operator |(long x, long y);  
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);  
uint operator ^(uint x, uint y);  
long operator ^(long x, long y);  
ulong operator ^(ulong x, ulong y);

Der &-Operator berechnet das bitweise logische AND der beiden Operanden, der |-Operator berechnet das bitweise logische OR der beiden Operanden, und der ^-Operator berechnet das bitweise logische exklusive OR der beide Operanden. Diese Operationen verursachen keine Überläufe.

### Logische Operatoren für Enumerationen

Alle Enumerationstypen E enthalten implizit die folgenden vordefinierten logischen Operatoren:

E operator &(E x, E y);  
E operator |(E x, E y);  
E operator ^(E x, E y);

Das Ergebnis der Auswertung von x op y, wobei x und y Ausdrücke eines Enumerationstyps E mit einem zugrunde liegenden Typ U sind und op einer der logischen Operatoren ist, entspricht genau dem Auswertungsergebnis von (E)((U)x op (U)y). Die logischen Operatoren für Enumerationen führen also nur die logische Operation für den zugrunde liegenden Typ der beiden Operanden aus.

### Boolesche logische Operatoren

Folgende boolesche logische Operatoren sind vordefiniert:

bool operator &(bool x, bool y);

bool operator |(bool x, bool y);

bool operator ^(bool x, bool y);

Das Ergebnis von x & y ist true, wenn sowohl x und y als auch true sind. Andernfalls ist das Ergebnis false.

Das Ergebnis von x | y ist true, wenn entweder x oder y true ist. Andernfalls ist das Ergebnis false.

Das Ergebnis von x ^ y ist true, wenn x true ist und y falseist oder x false ist und y true ist. Andernfalls ist das Ergebnis false. Wenn die Operanden vom Typ bool sind, ergibt der ^-Operator dasselbe Ergebnis wie der !=-Operator.

### Boolesche logische Operatoren, die NULL-Werte zulassen

Der boolesche Typ bool?, der NULL-Werte zulässt, kann die drei Werte true, false und null darstellen und stimmt konzeptionell mit dem in SQL für boolesche Ausdrücke verwendeten Typ mit drei Werten überein. Um sicherzustellen, dass die durch die Operatoren & und | erstellten Ergebnisse für Operanden vom Typ bool? der SQL-Logik für drei Werte entsprechen, werden die folgenden vordefinierten Operatoren bereitgestellt:

bool? operator &(bool? x, bool? y);

bool? operator |(bool? x, bool? y);

Die folgende Tabelle enthält die Ergebnisse dieser Operatoren für alle Kombinationen der Werte true, false und null.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | x & y | x | y |
| true | true | true | true |
| true | false | false | true |
| true | null | null | true |
| false | true | false | true |
| false | false | false | false |
| false | null | false | null |
| null | true | null | true |
| null | false | false | null |
| null | null | null | null |

## Logische Bedingungsoperatoren

Die Operatoren && und || werden bedingte logische Operatoren genannt. Sie sind auch unter dem Namen logische „Kurzschlussoperatoren“ bekannt.

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

Die Operatoren && und || sind konditionale Versionen der Operatoren & und |:

* Die Operation x && y entspricht der Operation x & y, mit der Ausnahme, dass y nur dann ausgewertet wird, wenn x nicht false ist.
* Die Operation x || y entspricht der Operation x | y, mit der Ausnahme, dass y nur dann ausgewertet wird, wenn x nicht true ist.

Wenn der Operand eines bedingten logischen Operators den Kompilierungstyp dynamic aufweist, wird der Ausdruck dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Ausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit unter Verwendung des Laufzeittyps derjenigen Operanden, die den Kompilierungstyp dynamic aufweisen.

Eine Operation der Form x && y oder x || y wird durch Anwenden der Überladungsauflösung (§7.3.4) genauso verarbeitet, als wenn die Operation x & y oder x | y lauten würde. Es gilt Folgendes:

* Wenn die Überladungsauflösung keinen am besten geeigneten Operator findet oder einen vordefinierten logischen Operator für ganze Zahlen auswählt, tritt ein Bindungsfehler auf.
* Andernfalls gilt: Wenn der ausgewählte Operator ein vordefinierter logischer boolescher Operator (§7.11.3) ist oder ein logischer boolescher Operator, der NULL-Werte zulässt, (§7.11.4), wird die Operation entsprechend der Beschreibung in §7.12.1 verarbeitet.
* In allen anderen Fällen ist der ausgewählte Operator ein benutzerdefinierter Operator, und die Operation wird gemäß der Beschreibung in §7.12.2 verarbeitet.

Ein direkter Überlauf der logischen Bedingungsoperatoren ist nicht möglich. Da die bedingten logischen Operatoren jedoch in Bezug auf die regulären logischen Operatoren ausgewertet werden, werden Überläufe der regulären logischen Operatoren, mit bestimmten Einschränkungen, gleichzeitig als Überläufe von bedingten logischen Operatoren gewertet. Dies wird in §7.12.2 näher erläutert.

### Boolesche logische Bedingungsoperatoren

Wenn die Operanden der Operatoren && oder || vom Typ bool sind oder von einem Typ, der keinen anwendbaren operator & oder operator | deklariert, jedoch implizite Konvertierungen in bool definieren, wird die Operation folgendermaßen verarbeitet:

* Die Operation x && y wird als x ? y : false ausgewertet. Zuerst wird also x ausgewertet und in den Typ bool konvertiert. Wenn x gleich true ist, wird y ausgewertet und in den Typ bool konvertiert. Dies wird dann als Ergebnis der Operation ausgegeben. Andernfalls ist das Ergebnis der Operation false.
* Die Operation x || y wird als x ? true : y ausgewertet. Zuerst wird also x ausgewertet und in den Typ bool konvertiert. Wenn x gleich true ist, dann ist auch das Ergebnis der Operation true. Andernfalls wird y ausgewertet und in den Typ bool konvertiert. Dies wird dann als Ergebnis der Operation ausgegeben.

### Benutzerdefinierte logische Bedingungsoperatoren

Wenn die Operanden der Operatoren && oder || einen Typ annehmen, der einen anwendbaren, benutzerdefinierten operator & oder operator | deklariert, müssen beide der folgenden Bedingungen zutreffen (T ist dabei der Typ, in dem der ausgewählte Operator deklariert ist):

* Der Rückgabetyp und der Typ der einzelnen Parameter des ausgewählten Operators muss T sein. Der Operator muss also die logische AND-Verknüpfung oder die logische OR-Verknüpfung der beiden Operanden vom Typ T berechnen und ein Ergebnis vom Typ T zurückgeben.
* T muss die Deklarationen von operator true und operator false enthalten.

Wenn eine dieser Voraussetzungen nicht erfüllt ist, tritt während der Bindung ein Fehler auf. Andernfalls wird die &&- oder ||-Operation berechnet, indem der benutzerdefinierte operator true-Operator oder operator false-Operator mit dem ausgewählten benutzerdefinierten Operator kombiniert wird:

* Die Operation x && y wird als T.false(x) ? x : T.&(x, y) ausgewertet, wobei T.false(x) ein Aufruf von dem in T deklarierten operator false ist und T.&(x, y) ein Aufruf des ausgewählten operator &. Zuerst wird also x ausgewertet und operator false aufgerufen, um zu prüfen, ob x definitiv falsch (false) ergibt. Wenn x definitiv falsch (false) ist, entspricht das Ergebnis der Operation dem zuvor für x berechneten Wert. Andernfalls wird y ausgewertet, und der ausgewählte operator & wird für den zuvor für x und den für y berechneten Wert aufgerufen, um das Ergebnis der Operation zu ermitteln.
* Die Operation x || y wird als T.true(x) ? x : T.|(x, y) ausgewertet, wobei T.true(x) ein Aufruf von dem in T deklarierten operator true ist und T.|(x, y) ein Aufruf des ausgewählten operator |. Zuerst wird also x ausgewertet und operator true aufgerufen, um zu prüfen, ob x definitiv wahr (true) ergibt. Wenn x definitiv wahr (true) ist, entspricht das Ergebnis der Operation dem zuvor für x berechneten Wert. Andernfalls wird y ausgewertet, und der ausgewählte operator | wird für den zuvor für x und den für y berechneten Wert aufgerufen, um das Ergebnis der Operation zu ermitteln.

In beiden Operationen wird der durch x vorgegebene Ausdruck nur einmal ausgewertet, und der durch y vorgegebene Ausdruck wird nicht oder genau einmal ausgewertet.

Ein Beispiel für einen Typ, der operator true und operator false implementiert, finden Sie in §11.4.2.

## Der NULL-Sammeloperator

Der ??-Operator wird als der NULL-Sammeloperator bezeichnet.

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

Bei einem NULL-Sammelausdruck der Form a ?? b muss a ein Typ, der NULL-Werte zulässt, oder ein Verweistyp sein. Wenn a nicht NULL ist, ist a das Ergebnis von a ?? b, andernfalls lautet das Ergebnis b. Bei der Operation wird b nur ausgewertet, wenn a NULL ist.

Der NULL-Sammeloperator ist rechtsorientiert, d. h., die Operationen werden von rechts nach links gruppiert. So wird ein Ausdruck der Form a ?? b ?? c beispielsweise als a ?? (b ?? c) ausgewertet. Allgemein bedeutet dies, dass ein Ausdruck der Form E1 ?? E2 ?? ... ?? EN den ersten Operanden zurückgibt, der nicht NULL ist, oder NULL, wenn alle Operanden NULL sind.

Der Typ des Ausdrucks a ?? b ist davon abhängig, welche impliziten Konvertierungen für die Operanden verfügbar sind. Der Typ von a ?? b ist A0, A oder B (in dieser Reihenfolge), wobei A der Typ von a (sofern a über einen Typ verfügt), B der Typ von b (sofern b über einen Typ verfügt) und A0 der zugrunde liegende Typ von A ist, sofern A ein Typ ist, der NULL-Werte zulässt, andernfalls A. a ?? b wird folgendermaßen verarbeitet:

* Wenn A vorhanden ist und kein Typ, der NULL-Werte zulässt, und kein Verweistyp ist, tritt ein Kompilierungsfehler auf.
* Wenn b ein dynamischer Ausdruck ist, ist der Ergebnistyp dynamic. Zur Laufzeit wird zuerst a ausgewertet. Wenn a nicht NULL ist, wird a in dynamic konvertiert und wird zum Ergebnis. Andernfalls wird b ausgewertet und als Ergebnis ausgegeben.
* Wenn andernfalls A vorhanden und ein Typ ist, der NULL-Werte zulässt, und es eine implizite Konvertierung von b in A0 gibt, lautet der Ergebnistyp A0. Zur Laufzeit wird zuerst a ausgewertet. Wenn a nicht NULL ist, wird a in den Typ A0 entwrappt und wird zum Ergebnis. Andernfalls wird b ausgewertet und in den Typ A0 konvertiert. Dies wird dann als Ergebnis ausgegeben.
* Wenn andernfalls A vorhanden ist und es eine implizite Konvertierung von b in A gibt, lautet der Ergebnistyp A. Zur Laufzeit wird zuerst a ausgewertet. Wenn a nicht NULL ist, wird a als Ergebnis ausgegeben. Andernfalls wird b ausgewertet und in den Typ A konvertiert. Dies wird dann als Ergebnis ausgegeben.
* Wenn andernfalls b einen Typ B besitzt und es eine implizite Konvertierung von a in B gibt, lautet der Ergebnistyp B. Zur Laufzeit wird zuerst a ausgewertet. Wenn a nicht NULL ist, wird a in Typ A0 entwrappt (sofern A vorhanden ist und NULL-Werte zulässt) und in Typ B konvertiert. Dies wird dann als Ergebnis ausgegeben. Andernfalls wird b ausgewertet und das Ergebnis.
* Andernfalls sind a und b nicht kompatibel, und es tritt ein Kompilierungsfehler auf.

## Bedingungsoperator

Der ?:-Operator ist der bedingte Operator. Zeitweise wird er auch als ternärer Operator bezeichnet.

conditional-expression:  
null-coalescing-expression  
null-coalescing-expression ? expression : expression

Ein Bedingungsausdruck der Form b ? x : y wertet zuerst die Bedingung b aus. Wenn b gleich true ist, wird x ausgewertet und als Ergebnis der Operation ausgegeben. Andernfalls wird y ausgewertet und als Ergebnis der Operation ausgegeben. In Bedingungsausdrücken wird immer nur x oder y ausgewertet.

Der bedingte Operator ist rechtsorientiert, d. h., die Operationen werden von rechts nach links gruppiert. So wird ein Ausdruck der Form a ? b : c ? d : e beispielsweise als a ? b : (c ? d : e) ausgewertet.

Der erste Operand des ?:-Operators muss ein Ausdruck sein, der implizit in bool konvertiert werden kann, oder ein Ausdruck eines Typs, der operator true implementiert. Ein Kompilierfehler tritt auf, wenn keine dieser Bedingungen erfüllt ist.

Der zweite und dritte Operand x und y des ?:-Operators bestimmen den Typ des Bedingungsausdrucks.

* Wenn x den Typ X und y den Typ Y aufweist, dann gilt Folgendes:
* Wenn eine implizite Konvertierung (§6.1) von X in Y vorhanden ist, jedoch nicht von Y in X, dann ist Y der Typ des Bedingungsausdrucks.
* Wenn eine implizite Konvertierung (§6.1) von Y in X vorhanden ist, jedoch nicht von X in Y, dann ist X der Typ des Bedingungsausdrucks.
* In allen anderen Fällen kann kein Ausdruckstyp bestimmt werden, und während der Kompilierung tritt ein Fehler auf.
* Wenn nur x oder y über einen Typ verfügt und sowohl x als auch y implizit in diesen Typ konvertiert werden können, so ist dies der Typ des Bedingungsausdrucks.
* In allen anderen Fällen kann kein Ausdruckstyp bestimmt werden, und während der Kompilierung tritt ein Fehler auf.

Die Laufzeitverarbeitung eines Bedingungsausdrucks der Form b ? x : y besteht aus den folgenden Schritten:

* Zuerst wird b ausgewertet und der bool-Wert von b bestimmt:
* Wenn eine implizite Konvertierung aus dem Typ von b in bool vorhanden ist, wird diese implizite Konvertierung ausgeführt, um einen bool-Wert zu ermitteln.
* Andernfalls wird zum Ermitteln eines bool-Werts der durch den Typ von b definierte operator true aufgerufen.
* Wenn der dabei ermittelte bool-Wert gleich true ist, wird x ausgewertet und in den Typ des Bedingungsausdrucks konvertiert, der als Ergebnis des Bedingungsausdrucks ausgegeben wird.
* In allen anderen Fällen wird y ausgewertet, in den Typ des Bedingungsausdrucks konvertiert und als Ergebnis des Bedingungsausdrucks ausgegeben.

## Konvertierungen anonymer Ausdrücke

Eine anonyme Funktion ist ein Ausdruck, der eine Inline-Methodendefinition darstellt. Eine anonyme Funktion muss weder einen Wert oder Typ enthalten noch über einen Wert oder Typ von sich selbst verfügen, sie muss aber in einen kompatiblen Delegattyp oder Ausdrucksbaumstrukturtyp konvertiert werden können. Die Auswertung einer anonymen Funktion ist vom Zieltyp der Konvertierung abhängig: Wenn dieser ein Delegattyp ist, wird die Konvertierung in einen Delegatwert ausgewertet, der auf die Methode verweist, die die anonyme Funktion definiert. Wenn der Zieltyp ein Ausdrucksbaumstrukturtyp ist, wird die Konvertierung in eine Ausdrucksbaumstruktur ausgewertet, die die Struktur der Methode als eine Objektstruktur darstellt.

Aus historischen Gründen stehen zwei syntaktische Versionen von anonymen Funktionen zur Verfügung: lambda-expressions und anonymous-method-expressions. lambda-expressions sind in fast allen Situationen genauer und aussagekräftiger als anonymous-method-expressions, die aus Kompatibilitätsgründen beibehalten werden.

lambda-expression:  
async*opt* anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
async*opt* delegate explicit-anonymous-function-signatureopt block

anonymous-function-signature:  
explicit-anonymous-function-signature   
implicit-anonymous-function-signature

explicit-anonymous-function-signature:  
( explicit-anonymous-function-parameter-list*opt* )

explicit-anonymous-function-parameter-list:  
explicit-anonymous-function-parameter  
explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

explicit-anonymous-function-parameter:  
anonymous-function-parameter-modifieropt type identifier

anonymous-function-parameter-modifier:   
ref  
out

implicit-anonymous-function-signature:  
( implicit-anonymous-function-parameter-list*opt* )  
implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:  
implicit-anonymous-function-parameter  
implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:  
identifier

anonymous-function-body:  
expression  
block

Der =>-Operator besitzt den gleichen Vorrang wie eine Zuweisung (=) und ist rechtsorientiert.

Eine anonyme Funktion mit dem async-Modifizierer ist eine Async-Funktion und folgt den in §10.14 beschriebenen Regeln.

Die Parameter einer anonymen Funktion der Form einer lambda-expression können explizit oder implizit typisiert werden. In einer explizit typisierten Parameterliste wird der Typ der einzelnen Parameter explizit angegeben. Bei einer implizit typisierten Parameterliste wird auf die Typen der Parameter aus dem Kontext geschlossen, in dem die anonyme Funktion auftritt. Insbesondere wenn die anonyme Funktion in einen kompatiblen Delegat- oder Ausdrucksbaumstrukturtyp konvertiert wird, gibt dieser Typ die Parametertypen an (§6.5).

In einer anonymen Funktion mit nur einem implizit typisierten Parameter in der Parameterliste keine Klammern angegeben werden. Das bedeutet, dass eine anonyme Funktion der Form

( param ) => expr

abgekürzt werden kann durch:

param => expr

Bei einer anonymen Funktion in Form einer anonymous-method-expression ist die Parameterliste optional. Sofern Parameter angegeben werden, müssen diese auch explizit typisiert werden. Andernfalls kann die anonyme Funktion mit einer beliebigen Parameterliste in einen Delegaten konvertiert werden, wenn diese keine out-Parameter enthält.

Ein block-Text einer anonymen Funktion ist erreichbar (§8.1), sofern die anonyme Funktion nicht in einer unerreichbaren Anweisung ausgeführt wird.

Im Folgenden finden Sie einige Beispiele für anonyme Funktionen:

x => x + 1 // Implicitly typed, expression body

x => { return x + 1; } // Implicitly typed, statement body

(int x) => x + 1 // Explicitly typed, expression body

(int x) => { return x + 1; } // Explicitly typed, statement body

(x, y) => x \* y // Multiple parameters

() => Console.WriteLine() // No parameters

async (t1,t2) => await t1 + await t2 // Async

delegate (int x) { return x + 1; } // Anonymous method expression

delegate { return 1 + 1; } // Parameter list omitted

Das Verhalten von lambda-expressions ist mit folgenden Ausnahme mit dem Verhalten von anonymous-method-expressions identisch:

* anonymous-method-expressions erlauben das vollständige Auslassen der Parameterliste und ermöglichen so das Konvertieren beliebiger Listen von Werteparametern in Delegattypen.
* lambda-expressions erlauben das Auslassen von und das Rückschließen auf Parametertypen, während anonymous-method-expressions die explizite Angabe von Parametertypen verlangen.
* Der Text einer lambda-expression kann ein Ausdruck oder ein Anweisungsblock sein, während der Text einer anonymous-method-expression ein Anweisungsblock sein muss.
* Nur lambda-expressions verfügen über Konvertierungen in kompatible Ausdrucksbaumstrukturtypen (§4.6).

### Signaturen anonymer Funktionen

Die optionale anonymous-function-signature einer anonymen Funktion definiert die Namen und optional auch die Typen der formalen Parameter für die anonyme Funktion. Der Gültigkeitsbereich der Parameter für die anonyme Funktion ist der anonymous-function-body. (§3.7) Der Text der anonymen Funktion bildet zusammen mit der Parameterliste (sofern angegeben) den Deklarationsabschnitt (§3.3). Daher tritt ein Kompilierungsfehler hinsichtlich des Namens eines Parameters der anonymen Funktion auf, wenn dieser mit dem Namen einer lokalen Variablen, einer lokalen Konstanten oder eines Parameters übereinstimmt, deren Gültigkeitsbereich die anonymous-method-expression oder die lambda-expression umfasst.

Wenn eine anonyme Funktion über eine explicit-anonymous-function-signature verfügt, ist die Menge der kompatiblen Delegat- und Ausdrucksbaumstrukturtypen auf die Typen mit denselben Parametertypen und Modifizierern in derselben Reihenfolge beschränkt. Im Unterschied zu Konvertierungen von Methodengruppen (§6.6) wird die Kontravarianz von anonymen Funktionsparametertypen nicht unterstützt. Wenn eine anonyme Funktion über keine anonymous-function-signature verfügt, ist die Menge der kompatiblen Delegat- und Ausdrucksbaumstrukturtypen auf die Typen beschränkt, die über keine out-Parameter verfügen.

Beachten Sie, dass eine anonymous-function-signature keine Attribute oder Parameterarrays enthalten kann. Dennoch kann eine anonymous-function-signature mit einem Delegattypen kompatibel sein, dessen Parameterliste ein Parameterarray enthält.

Beachten Sie des Weiteren, dass eine Konvertierung in einen Ausdrucksbaumstrukturtyp auch dann zu einem Kompilierungsfehler (§4.6) führen kann, wenn dieser Typ kompatibel ist.

### Anonyme Funktionstexte

Der Text (expression oder block) einer anonymen Funktion unterliegt den folgenden Regeln:

* Wenn die anonyme Funktion eine Signatur enthält, sind die in der Signatur angegebenen Parameter im Text verfügbar. Wenn die anonyme Funktion über keine Signatur verfügt, kann sie in einen Delegat- oder Ausdrucksbaumstrukturtypen konvertiert werden, der Parameter besitzt (§6.5), auf die Parameter kann im Text jedoch nicht zugegriffen werden.
* Mit Ausnahme der ref- bzw. out-Parameter, die in der Signatur (sofern vorhanden) der nächsten einschließenden anonymen Funktion angegeben werden, tritt ein Kompilierungsfehler auf, wenn im Text auf einen ref- oder out-Parameter zugegriffen wird.
* Wenn es sich beim Typ von this um einen Strukturtyp handelt, tritt ein Kompilierungsfehler auf, wenn im Text auf this zugegriffen wird. Dies ist der Fall bei einem expliziten Zugriff (wie in this.x) und bei einem impliziten Zugriff (wie in x, wobei x ein Instanzmember der Struktur ist). Diese Regel verhindert lediglich einen solchen Zugriff und wirkt sich nicht darauf aus, ob eine Membersuche einen Member der Struktur zum Ergebnis hat.
* Der Text kann auf die äußeren Variablen (§7.15.5) der anonymen Funktion zugreifen. Beim Zugriff auf eine äußere Variable wird auf die Instanz der Variablen verwiesen, die zum Zeitpunkt der Auswertung der lambda-expression oder der anonymous-method-expression aktiv ist (§7.15.6).
* Wenn der Text eine goto-Anweisung, break-Anweisung oder continue-Anweisung enthält, deren Ziel außerhalb des Texts oder im Text einer enthaltenen anonymen Funktion liegt, tritt ein Kompilierungsfehler auf.
* Eine return-Anweisung im Text gibt die Steuerung nicht vom einschließenden Funktionsmember, sondern von einem Aufruf der nächsten einschließenden anonymen Funktion zurück. Ein in einer return-Anweisung angegebener Ausdruck muss implizit in den Rückgabetyp des Delegat- oder Ausdrucksbaumstrukturtyps konvertierbar sein, in den die nächste einschließende lambda-expression oder anonymous-method-expression konvertiert wird (§6.5).

Eine Möglichkeit zum Ausführen des Blocks einer anonymen Funktion auf andere Weise als durch Auswerten und Aufrufen der lambda-expression oder anonymous-method-expression ist explizit nicht festgelegt. Insbesondere hat der Compiler die Möglichkeit, eine anonyme Funktion durch das synthetische Erstellen einer oder mehrerer benannter Methoden oder Typen zu implementieren. Die Namen solcher auf synthetische Weise erstellen Elemente müssen eine Form haben, die für die Verwendung durch den Compiler reserviert ist.

### Überladungsauflösung

Anonyme Funktionen in einer Argumentliste sind Teil von Typrückschluss und Überladungsauflösung. Die genauen Regeln finden Sie in §7.5.2 und §7.5.3.

Im folgenden Beispiel wird die Auswirkung von anonymen Funktionen auf die Überladungsauflösung veranschaulicht.

class ItemList<T>: List<T>  
{  
 public int Sum(Func<T,int> selector) {  
 int sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }

public double Sum(Func<T,double> selector) {  
 double sum = 0;  
 foreach (T item in this) sum += selector(item);  
 return sum;  
 }  
}

Die ItemList<T>-Klasse verfügt über zwei Sum-Methoden. Beide akzeptieren ein selector-Argument, das den zu summierenden Wert aus einem Listenelement extrahiert. Der extrahierte Wert kann ein int oder ein double sein, und die Summe ist entsprechend entweder ein int oder ein double.

Die Sum-Methoden könnten beispielsweise zum Berechnen Summen aus einer Liste von Einzelartikeln in einer Bestellung verwendet werden.

class Detail  
{  
 public int UnitCount;  
 public double UnitPrice;  
 ...  
}

void ComputeSums() {  
 ItemList<Detail> orderDetails = GetOrderDetails(...);  
 int totalUnits = orderDetails.Sum(d => d.UnitCount);  
 double orderTotal = orderDetails.Sum(d => d.UnitPrice \* d.UnitCount);  
 ...  
}

Beim ersten Aufruf von orderDetails.Sum sind beide Sum-Methoden anwendbar, da die anonyme Funktion d => d.UnitCount sowohl mit Func<Detail,int> als auch mit Func<Detail,double> kompatibel ist. Bei der Überladungsauflösung wird jedoch die erste Sum-Methode gewählt, da die Konvertierung in Func<Detail,int> besser ist als die Konvertierung in Func<Detail,double>.

Beim zweiten Aufruf von orderDetails.Sum ist nur die zweite Sum-Methode anwendbar, da die anonyme Funktion d => d.UnitPrice \* d.UnitCount einen Wert vom Typ double erzeugt. Daher wird bei der Überladungsauflösung die zweite Sum-Methode für diesen Aufruf ausgewählt.

### Anonyme Funktionen und dynamische Bindungen

Eine anonyme Funktion kann kein Empfänger, Argument oder Operand einer dynamisch gebundenen Operation sein.

### Äußere Variablen

Alle lokalen Variablen, Wertparameter und Parameterarrays, deren Gültigkeitsbereich die lambda-expression oder die anonymous-method-expression enthält, werden als äußere Variablen der anonymen Funktion bezeichnet. In einem Instanzfunktionsmember einer Klasse wird der this-Wert als Wertparameter angesehen, der eine äußere Variable für jede anonyme Funktion innerhalb des Funktionsmembers ist.

#### Erfasste äußere Variablen

Wenn durch eine anonyme Funktion auf eine äußere Variable verwiesen wird, bedeutet dies, dass die äußere Variable von der anonymen Funktion erfasst wurde. Im Allgemeinen ist die Lebensdauer einer lokalen Variablen auf die Ausführung des Blocks oder der Anweisung beschränkt, dem bzw. der sie zugeordnet ist (§5.1.7). Die Lebensdauer einer erfassten äußeren Variablen dauert jedoch mindestens bis zu dem Zeitpunkt an, zu dem der Delegat oder die Ausdrucksbaumstruktur, der bzw. die von der anonymen Funktion erstellt wird, für die Garbage Collection freigegeben wird.

In dem Beispiel

using System;

delegate int D();

class Test  
{  
 static D F() {  
 int x = 0;  
 D result = () => ++x;  
 return result;  
 }

static void Main() {  
 D d = F();  
 Console.WriteLine(d());  
 Console.WriteLine(d());  
 Console.WriteLine(d());  
 }  
}

wird die lokale Variable x von der anonymen Funktion erfasst, und die Lebensdauer von x wird mindestens bis zu dem Zeitpunkt verlängert, zu dem der von F zurückgegebene Delegat für die Garbage Collection freigegeben wird (dies ist erst beim Abschluss der Ausführung des Programms der Fall). Da bei jedem Aufruf der anonymen Funktion dieselbe Instanz von x verwendet wird, lautet die Ausgabe des Beispiels wie folgt:

1  
2  
3

Wenn eine lokale Variable oder ein Wertparameter von einer anonymen Funktion erfasst wird, wird die lokale Variable bzw. der Parameter nicht mehr als feste Variable (§18.3) angesehen, sondern als bewegliche Variable. Daher muss jeder unsafe-Code, der die Adresse einer erfassten äußeren Variablen annimmt, die Variable zunächst mithilfe der fixed-Anweisung festlegen.

Beachten Sie, dass eine erfasste lokale Variable, im Gegensatz zu einer nicht erfassten Variablen, gleichzeitig für mehrere Ausführungsthreads verfügbar gemacht werden kann.

#### Instanziierung von lokalen Variablen

Eine lokale Variable wird als instanziiert angesehen, wenn bei der Ausführung der Gültigkeitsbereich der Variablen erreicht wird. Beim Aufrufen der folgenden Methode wird beispielsweise die lokale Variable x instanziiert und dreimal initialisiert, einmal bei jedem Durchlauf der Schleife.

static void F() {  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 ...  
 }  
}

Wenn jedoch die Deklaration von x an eine Position außerhalb der Schleife verschoben wird, ist das Ergebnis eine einzige Instanziierung von x:

static void F() {  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 ...  
 }  
}

Wenn die Variable nicht erfasst wird, kann nicht genau bestimmt werden, wie oft eine lokale Variable instanziiert wird. Da die einzelnen Lebensdauern der Instanziierungen nicht zusammenhängen, kann von jeder Instanziierung einfach dieselbe Speicherposition verwendet werden. Wenn jedoch eine anonyme Funktion eine lokale Variable erfasst, werden die Auswirkungen der Instanziierung ersichtlich.

In dem Beispiel

using System;

delegate void D();

class Test  
{  
 static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 int x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
 }

static void Main() {  
 foreach (D d in F()) d();  
 }  
}

erzeugt die Ausgabe:

1  
3  
5

Wenn jedoch die Deklaration von x an eine Position außerhalb der Schleife verschoben wird:

static D[] F() {  
 D[] result = new D[3];  
 int x;  
 for (int i = 0; i < 3; i++) {  
 x = i \* 2 + 1;  
 result[i] = () => { Console.WriteLine(x); };  
 }  
 return result;  
}

wird die folgende Ausgabe erzeugt:

5  
5  
5

Wenn eine For-Schleife eine Iterationsvariable deklariert, wird diese Variable selbst als außerhalb der Schleife deklariert betrachtet. Daher wird, wenn das Beispiel für das Erfassen der Iterationsvariablen selbst geändert wird:

static D[] F() {  
 D[] result = new D[3];  
 for (int i = 0; i < 3; i++) {  
 result[i] = () => { Console.WriteLine(i); };  
 }  
 return result;  
}

nur eine Instanz der Iterationsvariablen erfasst, die folgende Ausgabe erzeugt:

3  
3  
3

Anonyme Funktionsdelegaten können einige erfasste Variablen gemeinsam verwenden, während sie gleichzeitig separate Instanzen von anderen verwenden. Wenn beispielsweise F wie folgt geändert wird:

static D[] F() {  
 D[] result = new D[3];  
 int x = 0;  
 for (int i = 0; i < 3; i++) {  
 int y = 0;  
 result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };  
 }  
 return result;  
}

erfassen die drei Delegaten dieselbe Instanz von x, jedoch separate Instanzen von y, und die Ausgabe lautet wie folgt:

1 1  
2 1  
3 1

Separate anonyme Funktionen können dieselbe Instanz einer äußeren Variablen erfassen. In dem Beispiel

using System;

delegate void Setter(int value);

delegate int Getter();

class Test  
{  
 static void Main() {  
 int x = 0;  
 Setter s = (int value) => { x = value; };  
 Getter g = () => { return x; };  
 s(5);  
 Console.WriteLine(g());  
 s(10);  
 Console.WriteLine(g());  
 }  
}

erfassen die beiden anonymen Funktionen dieselbe Instanz der lokalen Variablen x, und sie können daher über diese Variable miteinander "kommunizieren". Die Ausgabe des Beispiels lautet:

5  
10

### Auswertung anonymer Funktionsausdrücke

Eine anonyme Funktion F muss immer in einen Delegattyp D oder einen Ausdrucksbaumstrukturtyp E konvertiert werden. Dies kann direkt oder über die Ausführung eines Ausdrucks für die Delegaterstellung new D(F) erfolgen. Mit dieser Konvertierung wird das Ergebnis der anonymen Funktion bestimmt, wie in §6.5 beschrieben.

## Abfrageausdrücke

Abfrageausdrücke bieten eine spracheninterne Syntax für Abfragen, die denen relationaler und hierarchischer Abfragesprachen wie SQL und XQuery ähneln.

query-expression:  
from-clause query-body

from-clause:  
from typeopt identifier in expression

query-body:  
query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:  
query-body-clause  
query-body-clauses query-body-clause

query-body-clause:  
from-clause  
let-clause  
where-clause  
join-clause  
join-into-clause  
orderby-clause

let-clause:  
let identifier = expression

where-clause:  
where boolean-expression

join-clause:  
join typeopt identifier in expression on expression equals expression

join-into-clause:  
join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:  
orderby orderings

orderings:  
ordering  
orderings , ordering

ordering:  
expression ordering-directionopt

ordering-direction:  
ascending  
descending

select-or-group-clause:  
select-clause  
group-clause

select-clause:  
select expression

group-clause:  
group expression by expression

query-continuation:  
into identifier query-body

Ein Abfrageausdruck beginnt mit einer from-Klausel und endet entweder mit einer select- oder einer group-Klausel. Der from-Klausel können keine oder mehrere from, let, where, join- oder orderby-Klauseln folgen. Jede from-Klausel ist ein Generator, der eine Bereichsvariable einführt, die die Elemente einer Sequenz umfasst. let-Klauseln führen eine Bereichsvariable ein, die einen Wert darstellt, der mithilfe der vorhergehenden Bereichsvariablen berechnet wurde. Jede where-Klausel ist ein Filter, der Elemente aus dem Ergebnis ausschließt. join-Klauseln vergleichen die angegebenen Schlüssel der Quellsequenz mit den Schlüsseln einer anderen Sequenz und geben übereinstimmende Paare aus. Eine orderby-Klausel ordnet die Elemente entsprechend den angegebenen Kriterien neu. Und die finale select- oder group-Klausel gibt die Form des Ergebnisses entsprechend der Bereichsvariablen an. Abschließend können Sie mit einer into-Klausel Abfragen miteinander verbinden, indem die Ergebnisse einer Abfrage als Generator einer nachfolgenden Abfrage verwendet werden.

### Mehrdeutigkeiten in Abfrageausdrücken

Abfrageausdrücke enthalten eine Reihe von „kontextbezogenen Schlüsselwörtern“, d. h., Bezeichner mit einer speziellen Bedeutung in einem bestimmten Kontext. Dies gilt insbesondere für from, where, join, on, equals, into, let, orderby, ascending, descending, select, group und by. Um Mehrdeutigkeiten in Abfrageausdrücken durch das gleichzeitige Verwenden dieser Bezeichner als Schlüsselwörter oder einfache Namen zu vermeiden, gelten diese Bezeichner, wenn Sie in einem Abfrageausdruck auftreten, als Schlüsselwörter.

Aus diesem Grund ist ein Abfrageausdruck ein beliebiger Ausdruck, der mit "from identifier" gefolgt von einem beliebigen Token außer ";", "=" oder "," beginnt.

Um diese Wörter als Bezeichner in einem Abfrageausdruck zu verwenden, können Sie ihnen "@" (§2.4.2) voranstellen.

### Übersetzung von Abfrageausdrücken

Bei der Programmiersprache C# wird keine Ausführungssemantik für Abfrageausdrücke angegeben. Stattdessen werden Abfrageausdrücke in Methodenaufrufe übersetzt, die dem Muster eines Abfrageausdrucks entsprechen (§7.16.3). Die Abfrageausdrücke werden in Aufrufe der Methoden Where, Select, SelectMany, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, GroupBy und Cast übersetzt. Bei diesen Methoden wird davon ausgegangen, dass sie über bestimmte Signaturen und Ergebnistypen verfügen, die in §7.16.3 beschrieben werden. Bei diesen Methoden kann es sich um Instanzmethoden des Objekts handeln, das abgefragt wird, oder um Erweiterungsmethoden, die im Bezug auf dieses Objekt als extern gelten und die eigentliche Ausführung der Abfrage implementieren.

Die Übersetzung eines Abfrageausdrucks in einen Methodenaufruf ist eine syntaktische Zuordnung, die vor Typbindungen oder Überladungsauflösungen erfolgt. Dabei gilt diese Übersetzung als syntaktisch korrekt, es kann jedoch nicht garantiert werden, dass syntaktisch korrekter C#-Code generiert wird. Nach dem Übersetzen der Abfrageausdrücke werden die resultierenden Methodenaufrufe wie reguläre Methodenaufrufe verarbeitet. Dabei können wiederum Fehler entdeckt werden, wenn die Methoden z. B. nicht vorhanden sind, wenn Argumente über einen falschen Typ verfügen oder wenn die Methoden generisch sind, und daher der Typrückschluss fehlschlägt.

Die Verarbeitung eines Abfrageausdrucks erfolgt durch wiederholtes Anwenden der folgenden Übersetzungen, bis keine weitere Reduzierung möglich ist. Die Übersetzungen werden in der Reihenfolge ihrer Anwendung aufgelistet: Bei jedem Abschnitt wird davon ausgegangen, dass die Übersetzungen im vorherigen Abschnitt abgeschlossen wurden, da ein Abschnitt nach seinem Abschluss während der weiteren Verarbeitung desselben Abfrageausdrucks kein zweites Mal durchlaufen wird.

Eine Zuweisung zu Bereichsvariablen ist in Abfrageausdrücken nicht zulässig. Eine C#-Implementierung muss diese Einschränkung jedoch nicht immer erzwingen, da dies bei dem hier vorgestellten syntaktischen Übersetzungsschema möglicherweise nicht immer möglich ist.

Bei einigen Übersetzungen werden Bereichsvariablen mit transparenten Bezeichnern eingefügt, die durch \* gekennzeichnet sind. Weitere Informationen über die speziellen Eigenschaften von transparenten Bezeichnern finden Sie in §7.16.2.7.

#### Select- und groupby-Klauseln mit Fortsetzungen

Ein Abfrageausdruck mit einer Fortsetzung

from … into x …

wird übersetzt in:

from x in ( from … ) …

Bei den Übersetzungen in den folgenden Abschnitten wird davon ausgegangen, dass die Abfragen über keine into-Fortsetzungen verfügen.

In dem Beispiel

from c in customers  
group c by c.Country into g  
select new { Country = g.Key, CustCount = g.Count() }

wird übersetzt in:

from g in  
 from c in customers  
 group c by c.Country  
select new { Country = g.Key, CustCount = g.Count() }

Die endgültige Übersetzung hierfür lautet:

customers.  
GroupBy(c => c.Country).  
Select(g => new { Country = g.Key, CustCount = g.Count() })

#### Explizite Bereichsvariablentypen

Eine from-Klausel, die explizit einen Bereichsvariablentyp angibt

from T x in e

wird übersetzt in:

from x in ( e ) . Cast < T > ( )

Eine join-Klausel, die explizit einen Bereichsvariablentyp angibt

join T x in e on k1 equals k2

wird übersetzt in:

join x in ( e ) . Cast < T > ( ) on k1 equals k2

Bei den Übersetzungen in den folgenden Abschnitten wird davon ausgegangen, dass die Abfragen über keine expliziten Bereichsvariablentypen verfügen.

In dem Beispiel

from Customer c in customers  
where c.City == "London"  
select c

wird übersetzt in:

from c in customers.Cast<Customer>()  
where c.City == "London"  
select c

Die endgültige Übersetzung hierfür lautet:

customers.  
Cast<Customer>().  
Where(c => c.City == "London")

Explizite Bereichsvariablentypen sind hilfreich für das Abfragen von Auflistungen, die die nicht generische IEnumerable-Schnittstelle implementieren, jedoch nicht für die generische IEnumerable<T>-Schnittstelle. Im Beispiel oben würde dies zutreffen, wenn customers vom Typ ArrayList wäre.

#### Degenerieren von Abfrageausdrücken

Ein Abfrageausdruck der Form

from x in e select x

wird übersetzt in:

( e ) . Select ( x => x )

In dem Beispiel

from c in customers  
select c

wird übersetzt in:

customers.Select(c => c)

Ein degenerierter Abfrageausdruck wählt einfach die Elemente aus der Quelle aus. Bei einer späteren Phase der Übersetzung werden degenerierte Abfragen, die bei einem anderen Übersetzungsschritt eingeführt wurden, entfernt, indem sie durch ihre Quelle ersetzt werden. Sie sollten jedoch unbedingt sicherstellen, dass das Ergebnis eines Abfrageausdrucks nie das Quellobjekt selbst ist, da damit der Typ und die Identität der Quelle durch den Client der Abfrage festgestellt werden könnte. Daher werden bei diesem Schritt degenerierte Abfragen, die direkt in den Quellcode geschrieben wurden, geschützt, indem Select explizit in der Quelle aufgerufen wird. Anschließend müssen Implementierer von Select und anderen Abfrageoperatoren sicherstellen, dass diese Methoden nie das Quellobjekt selbst zurückgeben.

#### Die Klauseln from, let, where, join und orderby

Ein Abfrageausdruck mit einer zweiten from-Klausel gefolgt von einer select-Klausel

from x1 in e1  
from x2 in e2  
select v

wird übersetzt in:

( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )

Ein Abfrageausdruck mit einer zweiten from-Klausel, auf die keine select -Klausel folgt

from x1 in e1  
from x2 in e2  
…

wird übersetzt in:

from \* in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )  
…

Ein Abfrageausdruck mit einer let-Klausel

from x in e  
let y = f  
…

wird übersetzt in:

from \* in ( e ) . Select ( x => new { x , y = f } )  
…

Ein Abfrageausdruck mit einer where-Klausel

from x in e  
where f  
…

wird übersetzt in:

from x in ( e ) . Where ( x => f )  
…

Ein Abfrageausdruck mit einer join-Klausel ohne into gefolgt von einer select-Klausel

from x1 in e1  
join x2 in e2 on k1 equals k2  
select v

wird übersetzt in:

( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )

Ein Abfrageausdruck mit einer join-Klausel ohne into, auf die etwas anderes als eine select-Klausel folgt

from x1 in e1  
join x2 in e2 on k1 equals k2   
…

wird übersetzt in:

from \* in ( e1 ) . Join(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })  
…

Ein Abfrageausdruck mit einer join-Klausel mit into gefolgt von einer select-Klausel

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
select v

wird übersetzt in:

( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )

Ein Abfrageausdruck mit einer join-Klausel mit into, auf die etwas anderes als eine select-Klausel folgt

from x1 in e1  
join x2 in e2 on k1 equals k2 into g  
…

wird übersetzt in:

from \* in ( e1 ) . GroupJoin(  
 e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })  
…

Ein Abfrageausdruck mit einer orderby-Klausel

from x in e  
orderby k1 , k2 , … , kn  
…

wird übersetzt in:

from x in ( e ) .   
OrderBy ( x => k1 ) .   
ThenBy ( x => k2 ) .  
 … .   
ThenBy ( x => kn )  
…

Wenn eine Sortierklausel den Richtungsbezeichner descending angibt, wird stattdessen ein Aufruf von OrderByDescending oder ThenByDescending erzeugt.

Bei den folgenden Übersetzungen wird davon ausgegangen, dass keine let-, where-, join- oder orderby-Klauseln vorhanden sind und dass maximal eine einleitende from-Klausel in jedem Abfrageausdruck enthalten ist.

In dem Beispiel

from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }

wird übersetzt in:

customers.  
SelectMany(c => c.Orders,  
 (c,o) => new { c.Name, o.OrderID, o.Total }  
)

In dem Beispiel

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

wird übersetzt in:

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

Die endgültige Übersetzung hierfür lautet:

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })

Dabei ist x ein vom Compiler generierter Bezeichner, der ansonsten unsichtbar ist und auf den nicht zugegriffen werden kann.

In dem Beispiel

from o in orders  
let t = o.Details.Sum(d => d.UnitPrice \* d.Quantity)  
where t >= 1000  
select new { o.OrderID, Total = t }

wird übersetzt in:

from \* in orders.  
 Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) })  
where t >= 1000   
select new { o.OrderID, Total = t }

Die endgültige Übersetzung hierfür lautet:

orders.  
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice \* d.Quantity) }).  
Where(x => x.t >= 1000).  
Select(x => new { x.o.OrderID, Total = x.t })

Dabei ist x ein vom Compiler generierter Bezeichner, der ansonsten unsichtbar ist und auf den nicht zugegriffen werden kann.

In dem Beispiel

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
select new { c.Name, o.OrderDate, o.Total }

wird übersetzt in:

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c.Name, o.OrderDate, o.Total })

In dem Beispiel

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID into co  
let n = co.Count()  
where n >= 10  
select new { c.Name, OrderCount = n }

wird übersetzt in:

from \* in customers.  
 GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co })  
let n = co.Count()  
where n >= 10   
select new { c.Name, OrderCount = n }

Die endgültige Übersetzung hierfür lautet:

customers.  
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c, co }).  
Select(x => new { x, n = x.co.Count() }).  
Where(y => y.n >= 10).  
Select(y => new { y.x.c.Name, OrderCount = y.n)

Dabei sind x und y vom Compiler generierte Bezeichner, die ansonsten unsichtbar sind und auf die nicht zugegriffen werden kann.

In dem Beispiel

from o in orders  
orderby o.Customer.Name, o.Total descending  
select o

führt zu der endgültigen Übersetzung:

orders.  
OrderBy(o => o.Customer.Name).  
ThenByDescending(o => o.Total)

#### Select- Klauseln

Ein Abfrageausdruck der Form

from x in e select v

wird übersetzt in:

( e ) . Select ( x => v )

Dies gilt nicht, wenn v der Bezeichner x ist. Die Übersetzung lautet einfach:

( e )

Beispiel:

from c in customers.Where(c => c.City == “London”)  
select c

wird einfach übersetzt in:

customers.Where(c => c.City == “London”)

#### Groupby-Klauseln

Ein Abfrageausdruck der Form

from x in e group v by k

wird übersetzt in:

( e ) . GroupBy ( x => k , x => v )

Dies gilt nicht, wenn v der Bezeichner x ist. Die Übersetzung lautet:

( e ) . GroupBy ( x => k )

In dem Beispiel

from c in customers  
group c.Name by c.Country

wird übersetzt in:

customers.  
GroupBy(c => c.Country, c => c.Name)

#### Transparente Bezeichner

Bei einigen Übersetzungen werden Bereichsvariablen mit transparenten Bezeichnern eingefügt, die durch \* gekennzeichnet sind. Transparente Bezeichner sind keine eigentliche Funktion der Programmiersprache, sondern stellen lediglich einen Zwischenschritt bei der Übersetzung von Abfrageausdrücken dar.

Wenn eine Abfrageübersetzung einen transparenten Bezeichner einfügt, wird dieser bei den folgenden Übersetzungsschritten an anonyme Funktionen und anonyme Objektinitialisierer übergeben. In diesen Kontexten weisen transparente Bezeichner folgendes Verhalten auf:

* Wenn ein transparenter Bezeichner als Parameter in einer anonymen Funktion auftritt, befinden sich die Member des zugehörigen anonymen Typs automatisch im Gültigkeitsbereich im Text der anonymen Funktion.
* Wenn sich ein Member mit einem transparenten Bezeichner im Gültigkeitsbereich befindet, befinden sich auch die Member dieses Members im Gültigkeitsbereich.
* Wenn ein transparenter Bezeichner als Memberdeklarator in einem anonymen Objektinitialisierer auftritt, führt er einen Member mit einem transparenten Bezeichner ein.

Bei den bisher beschriebenen Übersetzungsschritten werden transparente Bezeichner immer zusammen mit anonymen Typen eingeführt, um mehrere Bereichsvariablen als Member eines einzelnen Objekts zu erfassen. In einer Implementierung von C# können auch andere Verfahren als anonyme Typen für das Gruppieren mehrerer Bereichsvariablen verwendet werden. Bei den folgenden Übersetzungsbeispielen, die veranschaulichen, wie transparente Bezeichner übersetzt werden, wird davon ausgegangen, dass anonyme Typen verwendet werden.

In dem Beispiel

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }

wird übersetzt in:

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }

Dies wird wiederum übersetzt in:

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(\* => o.Total).  
Select(\* => new { c.Name, o.Total })

Dies entspricht, wenn die transparenten Bezeichner entfernt werden, Folgendem:

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.Total })

Dabei ist x ein vom Compiler generierter Bezeichner, der ansonsten unsichtbar ist und auf den nicht zugegriffen werden kann.

In dem Beispiel

from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
join d in details on o.OrderID equals d.OrderID  
join p in products on d.ProductID equals p.ProductID  
select new { c.Name, o.OrderDate, p.ProductName }

wird übersetzt in:

from \* in customers.  
 Join(orders, c => c.CustomerID, o => o.CustomerID,   
 (c, o) => new { c, o })  
join d in details on o.OrderID equals d.OrderID  
join p in products on d.ProductID equals p.ProductID  
select new { c.Name, o.OrderDate, p.ProductName }

Dies wird wiederum reduziert in:

customers.  
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).  
Join(details, \* => o.OrderID, d => d.OrderID, (\*, d) => new { \*, d }).  
Join(products, \* => d.ProductID, p => p.ProductID, (\*, p) => new { \*, p }).  
Select(\* => new { c.Name, o.OrderDate, p.ProductName })

Die endgültige Übersetzung hierfür lautet:

customers.  
Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c, o }).  
Join(details, x => x.o.OrderID, d => d.OrderID,  
 (x, d) => new { x, d }).  
Join(products, y => y.d.ProductID, p => p.ProductID,  
 (y, p) => new { y, p }).  
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })

Dabei sind x, y und z vom Compiler generierte Bezeichner, die ansonsten unsichtbar sind und auf die nicht zugegriffen werden kann.

### Das Muster von Abfrageausdrücken

Das Muster von Abfrageausdrücken bildet ein Muster von Methoden, die Typen zur Unterstützung von Abfrageausdrücken implementieren können. Da Abfrageausdrücke mithilfe einer syntaktischen Zuordnung in Methodenaufrufe übersetzt werden, stehen für Typen flexible Möglichkeiten für das Implementieren des Musters für Abfrageausdrücke zur Verfügung. Die Methoden des Musters können z. B. als Instanzmethoden oder als Erweiterungsmethoden implementiert werden, da diese über eine identische Aufrufsyntax verfügen. Außerdem können die Methoden Delegaten und Ausdrucksbaumstrukturen anfordern, da anonyme Funktionen in diese konvertiert werden können.

Die empfohlene Form für einen generischen Typ C<T> , der das Muster für einen Abfrageausdruck unterstützt, wird unten dargestellt. Ein generischer Typ wird verwendet, um die korrekten Beziehungen zwischen Parameter- und Ergebnistypen darzustellen, es ist aber auch möglich, das Muster für nicht generische Typen zu implementieren.

delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C  
{  
 public C<T> Cast<T>();  
}

class C<T> : C  
{  
 public C<T> Where(Func<T,bool> predicate);

public C<U> Select<U>(Func<T,U> selector);

public C<V> SelectMany<U,V>(Func<T,C<U>> selector,  
 Func<T,U,V> resultSelector);

public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,  
 Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,  
 Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

public O<T> OrderBy<K>(Func<T,K> keySelector);

public O<T> OrderByDescending<K>(Func<T,K> keySelector);

public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,  
 Func<T,E> elementSelector);  
}

class O<T> : C<T>  
{  
 public O<T> ThenBy<K>(Func<T,K> keySelector);

public O<T> ThenByDescending<K>(Func<T,K> keySelector);  
}

class G<K,T> : C<T>  
{  
 public K Key { get; }  
}

Die oben gezeigten Methoden verwenden die generischen Delegattypen Func<T1, R> und Func<T1, T2, R>, es können aber auch andere Delegat- oder Ausdrucksbaumstrukturtypen verwendet werden, die dieselben Beziehungen zwischen Parameter- und Ergebnistypen aufweisen.

Beachten Sie die empfohlene Beziehung zwischen C<T> und O<T>, mit der sichergestellt wird, dass die ThenBy-Methode und die ThenByDescending-Methode nur im Ergebnis von OrderBy oder OrderByDescending verfügbar sind. Desweiteren sollten Sie die empfohlene Form für das Ergebnis von GroupBy beachten: eine Folge von Sequenzen, wobei jede innere Sequenz über eine zusätzliche Key-Eigenschaft verfügt.

Der System.Linq-Namespace bietet eine Implementierung des Musters für den Abfrageoperator für jeden beliebigen Typ, der die System.Collections.Generic.IEnumerable<T>-Schnittstelle implementiert.

## Zuweisungsoperatoren

Die Zuweisungsoperatoren weisen einer Variablen, einer Eigenschaft, einem Ereignis oder einem Indexelement einen neuen Wert zu.

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

Der linke Operand einer Zuweisung muss ein als Variable, Eigenschaftenzugriff, Indexerzugriff oder Ereigniszugriff klassifizierter Ausdruck sein.

Der =-Operator wird als einfacher Zuweisungsoperator bezeichnet. Er weist der durch den linken Operanden vorgegebenen Variablen, Eigenschaft oder dem Indexelement den Wert des rechten Operanden zu. Der linke Operand des einfachen Zuweisungsoperators darf kein Ereigniszugriff sein (außer wie in §10.8.1 beschrieben). Der einfache Zuweisungsoperator wird in §7.17.1 beschrieben.

Die Zuweisungsoperatoren mit Ausnahme des =-Operators werden Verbundzuweisungsoperatoren genannt. Diese Operatoren führen die angegebene Operation mit den beiden Operanden durch und weisen der durch den linken Operanden vorgegebenen Variablen, Eigenschaft oder dem vorgegebenen Indexelement das Ergebnis zu. Die Verbundzuweisungsoperatoren werden in §7.17.2 beschrieben.

Die Operatoren += und -= mit einem Ereigniszugriffsausdruck als linkem Operanden werden als Ereigniszuweisungsoperatoren bezeichnet. Alle anderen Zuweisungsoperatoren lassen keine Ereigniszugriffe als linken Operanden zu. Die Ereigniszuweisungsoperatoren werden in §7.17.3 beschrieben.

Zuweisungsoperatoren sind rechtsorientiert, d. h., die Operationen werden von rechts nach links gruppiert. So wird ein Ausdruck der Form a = b = c beispielsweise als a = (b = c) ausgewertet.

### Einfache Zuweisung

Der =-Operator wird als einfacher Zuweisungsoperator bezeichnet.

Wenn der linke Operand eines einfachen Zuweisungsoperators die Form E.P oder E[Ei] aufweist, wobei E den Kompilierungstyp dynamic aufweist, wird die Zuweisung dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Zuweisungsausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit basierend auf dem Laufzeittyp von E.

In einer einfachen Zuweisung muss der rechte Operand ein Ausdruck sein, der implizit in den Typ des linken Operanden konvertiert werden kann. Die Operation weist der durch den linken Operanden vorgegebenen Variablen, Eigenschaft oder Indexelement den Wert des rechten Operanden zu.

Das Ergebnis eines einfachen Zuweisungsausdrucks ist der dem linken Operanden zugewiesene Wert. Das Ergebnis hat denselben Typ wie der linke Operand und wird stets als Wert klassifiziert.

Wenn der linke Operand eine Eigenschaft oder ein Indexzugriff ist, muss die Eigenschaft oder der Indexer über einen set-Accessor verfügen. Ist dies nicht der Fall, tritt während der Bindung ein Fehler auf.

Die Laufzeitverarbeitung einer einfachen Zuweisung der Form x = y besteht aus den folgenden Schritten:

* Bei Klassifizierung von x als Variable:
* x wird zur Erzeugung der Variablen ausgewertet.
* y wird ausgewertet und ggf. mittels einer impliziten Konvertierung (§6.1) in den Typ x konvertiert.
* Wenn die durch x angegebene Variable ein Arrayelement eines reference-type ist, wird eine Laufzeitprüfung ausgeführt, um sicherzustellen, dass der für y errechnete Wert mit der Arrayinstanz kompatibel ist, deren Element x ist. Die Prüfung ist erfolgreich, wenn y gleich null ist, oder wenn eine implizite Verweiskonvertierung (§6.1.6 aus dem tatsächlichen Typ der Instanz, auf die y verweist, in den tatsächlichen Elementtyp der Arrayinstanz vorhanden ist, welche x enthält. Andernfalls wird eine System.ArrayTypeMismatchException ausgelöst.
* Der sich aus Auswertung und Konvertierung von y ergebende Wert wird an der Stelle gespeichert, die bei der Auswertung von x festgelegt wird.
* Bei Klassifizierung von x als Eigenschaft oder Indexzugriff:
* Der mit x verknüpfte Instanzausdruck (wenn x nicht static ist) sowie die mit x verknüpfte Argumentenliste (wenn x ein Indexzugriff ist) werden ausgewertet und die Ergebnisse in den darauffolgenden Accessoraufrufen vom Typ set verwendet.
* y wird ausgewertet und ggf. mittels einer impliziten Konvertierung (§6.1) in den Typ x konvertiert.
* Der set-Accessor von x wird mit dem Wert aufgerufen, der für y als das value-Argument errechnet wurde.

Durch die Arraykovarianzregeln (§12.5) wird ermöglicht, dass ein Wert eines Arraytyps A[] ein Verweis auf eine Instanz eines Arraytyps B[] sein kann, sofern eine implizite Verweiskonvertierung von B in A vorhanden ist. Aufgrund dieser Regeln ist bei der Zuweisung zu einem Arrayelement eines reference-type eine Laufzeitprüfung erforderlich, um sicherzustellen, dass der zugewiesene Wert mit der Arrayinstanz kompatibel ist. In dem Beispiel

string[] sa = new string[10];  
object[] oa = sa;

oa[0] = null; // Ok  
oa[1] = "Hello"; // Ok  
oa[2] = new ArrayList(); // ArrayTypeMismatchException

wird durch die letzte Zuweisung System.ArrayTypeMismatchException ausgelöst, da eine Instanz von ArrayList nicht in einem Element von string[] gespeichert werden kann.

Wenn eine in einem struct-type deklarierte Eigenschaft oder ein Indexer das Ziel einer Zuweisung ist, muss der Instanzausdruck, der mit der Eigenschaft oder dem Indexerzugriff verknüpft ist, als Variable klassifiziert werden. Wenn der Instanzausdruck als Wert klassifiziert ist, tritt während der Bindung ein Fehler auf. Dieselbe Regel gilt für Felder (siehe §7.6.4).

Bei den Deklarationen:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }

public int X {  
 get { return x; }  
 set { x = value; }  
 }

public int Y {  
 get { return y; }  
 set { y = value; }  
 }  
}

struct Rectangle  
{  
 Point a, b;

public Rectangle(Point a, Point b) {  
 this.a = a;  
 this.b = b;  
 }

public Point A {  
 get { return a; }  
 set { a = value; }  
 }

public Point B {  
 get { return b; }  
 set { b = value; }  
 }  
}

im Beispiel

Point p = new Point();  
p.X = 100;  
p.Y = 100;  
Rectangle r = new Rectangle();  
r.A = new Point(10, 10);  
r.B = p;

sind die Zuweisungen zu p.X, p.Y, r.A und r.B zulässig, da es sich bei p und r um Variablen handelt. In diesem Beispiel

Rectangle r = new Rectangle();  
r.A.X = 10;  
r.A.Y = 10;  
r.B.X = 100;  
r.B.Y = 100;

sind die Zuweisungen jedoch nicht gültig, da r.A und r.B keine Variablen sind.

### Verbundzuweisung

Wenn der linke Operand eines Verbundzuweisungsoperators die Form E.P oder E[Ei] aufweist, wobei E den Kompilierungstyp dynamic aufweist, wird die Zuweisung dynamisch gebunden (§7.2.2). In diesem Fall ist der Kompilierungstyp des Zuweisungsausdrucks dynamic, und die nachfolgend beschriebene Auflösung erfolgt zur Laufzeit basierend auf dem Laufzeittyp von E.

Eine Operation der Form x op= y wird durch Anwenden der Überladungsauflösung für binäre Operatoren (§7.3.4) genauso verarbeitet wie die Operation x op y. Es gilt Folgendes:

* Wenn der Rückgabetyp des ausgewählten Operators implizit in den Typ von x konvertiert werden kann, wird die Operation als x = x op y ausgewertet, mit der Ausnahme, dass x nur einmal ausgewertet wird.
* Andernfalls gilt: Wenn der ausgewählte Operator ein vordefinierter Operator ist, der Rückgabetyp des ausgewählten Operators explizit in den Typ von x und y implizit in den Typ von x konvertiert werden kann oder der Operator ein Schiebeoperator ist, wird die Operation als x = (T)(x op y) ausgewertet, wobei T der Typ von x ist, mit der Ausnahme, dass x nur einmal ausgewertet wird.
* In allen anderen Fällen ist der Verbundverweis ungültig, und es tritt ein Fehler während der Bindung auf.

Der Begriff "einmalige Auswertung" bedeutet, dass in der Auswertung von x op y die Ergebnisse der enthaltenen Ausdrücke von x temporär gespeichert und dann bei der Zuweisung zu x erneut verwendet werden. In der Zuweisung A()[B()] += C(), wobei A eine Methode ist, die int[] zurückgibt und B sowie C Methoden bezeichnen, die int zurückgeben, werden die Methoden nur einmal in der Reihenfolge A, B, C aufgerufen.

Wenn der linke Operand einer Verbundzuweisung ein Eigenschaftenzugriff oder ein Indexzugriff ist, muss die Eigenschaft oder der Indexer sowohl über einen get-Accessor als auch einen set-Accessor verfügen. Ist dies nicht der Fall, tritt während der Bindung ein Fehler auf.

Die zweite der oben stehenden Regeln lässt zu, dass x op= y in bestimmten Kontexten wie x = (T)(x op y) ausgewertet wird. Die Regel besagt, dass die vordefinierten Operatoren als Verbundoperatoren verwendet werden können, wenn der linke Operand vom Typ sbyte, byte, short, ushort oder char ist. Selbst wenn beide Argumente einen dieser Typen annehmen, ergeben die vordefinierten Operatoren ein Ergebnis vom Typ int (siehe Beschreibung in §7.3.6.2). Folglich wäre es ohne Typumwandlung nicht möglich, das Ergebnis dem linken Operanden zuzuweisen.

Die Regel für vordefinierte Operatoren wirkt sich intuitiv so aus, dass x op= y zulässig ist, wenn sowohl x op y als auch x = y zulässig sind. In dem Beispiel

byte b = 0;  
char ch = '\0';  
int i = 0;

b += 1; // Ok  
b += 1000; // Error, b = 1000 not permitted  
b += i; // Error, b = i not permitted  
b += (byte)i; // Ok

ch += 1; // Error, ch = 1 not permitted  
ch += (char)1; // Ok

besteht die Fehlerursache darin, dass die jeweils entsprechende einfache Zuweisung ebenfalls fehlerhaft wäre.

Das bedeutet auch, dass Verbundzuweisungsoperationen heraufgestufte Operationen unterstützen. In dem Beispiel

int? i = 0;  
i += 1; // Ok

wird der heraufgestufte Operator +(int?,int?) verwendet.

### Ereigniszuweisung

Wenn der linke Operand des Operators += oder -= als eine Ereigniszuweisung klassifiziert ist, wird der Ausdruck folgendermaßen ausgewertet:

* Falls vorhanden wird der Instanzausdruck des Ereigniszugriffs ausgewertet.
* Der rechte Operand des +=- oder -=Operators wird ausgewertet und ggf. mittels einer impliziten Konvertierung (§6.1) in den Typ des linken Operanden konvertiert.
* Ein Ereignisaccessor des Ereignisses wird mit der Argumentenliste aufgerufen, die den rechten Operanden nach der Auswertung enthält und ggf. auch die Konvertierung. Wenn der Operator += ist, wird der add-Accessor aufgerufen, und wenn der Operator -= ist, wird der remove-Accessor aufgerufen.

Ein Ereigniszuweisungsausdruck gibt keinen Wert zurück. Ein Ereigniszuweisungsausdruck ist somit nur im Kontext einer statement-expression (§8.6) gültig.

## Ausdruck

Ein expression ist entweder ein non-assignment-expression oder eine assignment.

expression:   
non-assignment-expression  
assignment

non-assignment-expression:  
conditional-expression  
lambda-expression  
query-expression

## Konstante Ausdrücke

Eine constant-expression ist ein Ausdruck, der während der Kompilierung vollständig ausgewertet werden kann.

constant-expression:  
expression

Ein konstanter Ausdruck muss das null-Literal oder ein Wert mit einem der folgenden Typen sein: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, object, string oder ein beliebiger Enumerationstyp. Nur die folgenden Konstrukte sind in konstanten Ausdrücken zulässig:

* Literale (einschließlich dem null-Literal).
* Verweise auf const-Member von Klassen- und Strukturtypen.
* Verweise auf Member mit einem Enumerationstyp.
* Verweise auf const-Parameter oder lokale Variablen
* Unterausdrücke in Klammern, die selbst konstante Ausdrücke sind.
* Typumwandlungsausdrücke, vorausgesetzt, der Zieltyp ist einer der oben aufgeführten Typen.
* checked- und unchecked -Ausdrücke
* Standardwertausdrücke
* Die vordefinierten unären Operatoren +, –, ! und ~.
* Die vordefinierten binären Operatoren +, –, \*, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, <= und >=, vorausgesetzt, dass alle Operanden von einem der oben aufgeführten Typen sind.
* Der bedingte Operator ?:.

Die folgenden Konvertierungen sind in konstanten Ausdrücken zulässig:

* Identitätskonvertierungen
* Numerische Konvertierungen
* Enumerationskonvertierungen
* Konvertierungen von konstanten Ausdrücken
* Konvertierungen von impliziten und expliziten Verweisen unter der Voraussetzung, dass die Quelle der Konvertierungen ein konstanter Ausdruck ist, der als null-Wert ausgewertet wird.

Andere Konvertierungen, einschließlich Boxing, Unboxing und Konvertierungen von impliziten Verweisen ungleich null, sind in konstanten Ausdrücken nicht zulässig. Beispiel:

class C {  
 const object i = 5; // error: boxing conversion not permitted  
 const object str = “hello”; // error: implicit reference conversion  
}

Die Initialisierung von i führt zu einem Fehler, weil eine Boxingkonvertierung erforderlich ist. Die Initialisierung von str führt zu einem Fehler, weil eine Konvertierung eines impliziten Verweises aus einem Wert ungleich Null erforderlich ist.

Wenn ein Ausdruck eine der oben aufgeführten Anforderungen erfüllt, wird der Ausdruck während der Kompilierung ausgewertet. Das trifft auch dann zu, wenn der Ausdruck ein untergeordneter Ausdruck zu einem längeren Ausdruck ist, der nicht konstante Konstrukte enthält.

Bei der Auswertung von konstanten Ausdrücken während der Kompilierung werden dieselben Regeln wie bei der Laufzeitauswertung von nicht konstanten Ausdrücken verwendet. Beide Verfahren unterschieden sich nur hinsichtlich der Fehlerart: An der Stelle, an der bei der Laufzeitauswertung eine Ausnahme auftreten würde, tritt bei der Auswertung während der Kompilierung ein Kompilierfehler auf.

Wenn es bei der Auswertung eines Ausdrucks während der Kompilierung zu Überläufen in arithmetischen Operationen vom Ganzzahltyp sowie Konvertierungen kommt, verursachen diese nur dann keinen Fehler während der Kompilierung (§7.19), wenn ein konstanter Ausdruck explizit in einen unchecked-Kontext eingefügt wird.

Konstante Ausdrücke treten in folgenden Kontexten auf: Dabei tritt in diesen Kontexten ein Kompilierungsfehler auf, wenn ein Ausdruck während der Kompilierung nicht vollständig ausgewertet werden kann.

* Konstantendeklarationen (§10.4).
* Enumerationsmemberdeklarationen (§14.3).
* Standardargumente für Listen formaler Parameter (§10.6.1)
* case-Bezeichnungen einer switch-Anweisung (§8.7.2).
* goto case-Anweisungen (§8.9.3).
* Dimensionslängen in einem Arrayerstellungsausdruck (§7.6.10.4), der einen Initialisierer enthält.
* Attribute (§17).

Eine implizite Konvertierung eines konstanten Ausdrucks (§6.1.9) ermöglicht die Umwandlung eines konstanten Ausdrucks vom Typ int in den Typ sbyte, byte, short, ushort, uint oder ulong, vorausgesetzt, der Wert des konstanten Ausdrucks befindet sich innerhalb des für den Zieltyp gültigen Bereichs.

## Boolesche Ausdrücke

Eine boolean-expression ist ein Ausdruck, der ein Ergebnis des Typs bool erzielt (entweder direkt oder durch Anwendung von operator true in bestimmten Kontexten, wie nachfolgend angegeben).

boolean-expression:  
expression

Der steuernde Bedingungsausdruck eines if-statement (§8.7.1), while-statement (§8.8.1), do-statement (§8.8.2) oder for-statement (§8.8.3) ist eine boolean-expression. Der steuernde Bedingungsausdruck des ?:-Operators (§7.14) folgt denselben Regeln wie eine boolean-expression, wird jedoch aus Gründen der Operatorrangfolge als conditional-or-expression klassifiziert.

Um einen Wert vom Typ bool zu erstellen, wie im Folgenden dargestellt, ist ein boolean-expression E erforderlich:

* Ist E implizit in bool konvertierbar, wird diese implizite Konvertierung zur Laufzeit angewendet.
* Andernfalls wird die Überladungsauflösung für unäre Operatoren (§7.3.3) angewendet, um eine eindeutige beste Implementierung des Operators true für E zu finden, die zur Laufzeit angewendet wird.
* Wird kein entsprechender Operator gefunden, tritt ein Bindungsfehler auf.

Der DBBool-Strukturtyp in §11.4.2 enthält ein Beispiel eines Typs, der operator true und operator false implementiert.

# Anweisungen

C# enthält eine Vielzahl von Anweisungen. Entwickler, die in C und in C++ programmiert haben, kennen die meisten dieser Anweisungen bereits.

statement:  
labeled-statement  
declaration-statement  
embedded-statement

embedded-statement:  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement   
yield-statement

Das nicht terminale embedded-statement wird für Anweisungen verwendet, die sich innerhalb von anderen Anweisungen befinden. Bei Verwendung eines embedded-statement anstelle eines statement ist die Verwendung von Deklarationsanweisungen und Anweisungen mit Bezeichnungen in diesen Kontexten ausgeschlossen. In dem Beispiel

void F(bool b) {  
 if (b)  
 int i = 44;  
}

führt zu einem Fehler bei der Kompilierung, da eine if-Anweisung für ihre if-Verzweigung keinstatement, sondern ein embedded-statement benötigt. Wäre dieser Code zugelassen, würde die Variable i zwar deklariert werden, könnte aber nicht verwendet werden. Beachten Sie jedoch, dass das Beispiel gültig ist, wenn Sie die Deklaration von i in einem Block platzieren.

## Endpunkte und Erreichbarkeit

Jede Anweisung hat einen Endpunkt. Bei dem Endpunkt einer Anweisung handelt es sich einfach ausgedrückt um die Position, die unmittelbar auf die Anweisung folgt. Die Ausführungsregeln für zusammengesetzte Anweisungen (Anweisungen, die eingebettete Anweisungen enthalten) geben an, welche Aktion ausgeführt wird, wenn die Steuerung den Endpunkt einer eingebetteten Anweisung erreicht. Wenn die Steuerung beispielsweise den Endpunkt einer Anweisung in einem Block erreicht, wird die Steuerung an die nächste Anweisung in dem Block übergeben.

Wenn die Möglichkeit besteht, dass die Anweisung von der Ausführung erreicht wird, wird die Anweisung als erreichbar bezeichnet. Wenn jedoch keine Möglichkeit für die Ausführung der Anweisung besteht, wird diese Anweisung als unerreichbar bezeichnet.

In dem Beispiel

void F() {  
 Console.WriteLine("reachable");  
 goto Label;  
 Console.WriteLine("unreachable");  
 Label:  
 Console.WriteLine("reachable");  
}

ist der zweite Aufruf von Console.WriteLine unerreichbar, weil keine Möglichkeit besteht, dass die Anweisung ausgeführt wird.

Wenn der Compiler feststellt, dass eine Anweisung unerreichbar ist, wird eine Warnung ausgegeben. Eine Anweisung ist ausdrücklich nicht fehlerhaft, wenn sie unerreichbar ist.

Um festzustellen, ob eine bestimmte Anweisung oder ein Endpunkt erreichbar ist, führt der Compiler eine Flussanalyse entsprechend der Erreichbarkeitsregeln aus, die für jede Anweisung definiert sind. Bei der Flussanalyse werden die Werte von konstanten Ausdrücken berücksichtigt (§7.19), die das Verhalten von Anweisungen steuern. Die möglichen Werte von nicht konstanten Ausdrücken bleiben jedoch unberücksichtigt. Für die Ablaufsteuerungsanalyse wird demnach davon ausgegangen, dass ein nicht konstanter Ausdruck eines bestimmten Typs alle möglichen Werte dieses Typs hat.

In dem Beispiel

void F() {  
 const int i = 1;  
 if (i == 2) Console.WriteLine("unreachable");  
}

handelt es sich bei dem booleschen Ausdruck der if-Anweisung um einen konstanten Ausdruck, da es sich bei beiden Operanden des ==-Operators um Konstanten handelt. Der konstante Ausdruck wird während der Kompilierung ausgewertet. Da dabei der Wert false erzeugt wird, gilt der Console.WriteLine-Aufruf als unerreichbar. Wenn i jedoch in eine lokale Variable geändert wird, also

void F() {  
 int i = 1;  
 if (i == 2) Console.WriteLine("reachable");  
}

wird der Console.WriteLine-Aufruf auch dann als erreichbar betrachtet, wenn er nicht wirklich ausgeführt wird.

Der block eines Funktionsmembers wird immer als erreichbar betrachtet. Da die Erreichbarkeitsregeln aller Anweisungen in einem Block nacheinander ausgewertet werden, kann die Erreichbarkeit jeder Anweisung bestimmt werden.

In dem Beispiel

void F(int x) {  
 Console.WriteLine("start");  
 if (x < 0) Console.WriteLine("negative");  
}

wird die Erreichbarkeit des zweiten Console.WriteLine-Aufrufs folgendermaßen bestimmt:

* Da der Block der Console.WriteLine-Methode erreichbar ist, ist auch die erste F-Ausdrucksanweisung erreichbar.
* Der Endpunkt der ersten Console.WriteLine-Ausdrucksanweisung ist erreichbar, da diese Anweisung erreichbar ist.
* Die if-Anweisung ist erreichbar, da der Endpunkt der ersten Console.WriteLine-Ausdrucksanweisung erreichbar ist.
* Da der boolesche Ausdruck der if-Anweisung nicht den konstanten Wert false hat, ist die zweite Console.WriteLine-Ausdrucksanweisung erreichbar.

Es gibt zwei Situationen, in denen ein Fehler während der Kompilierung auftritt, wenn der Endpunkt einer Anweisung erreichbar ist:

* Da die switch-Anweisung nicht zulässt, dass ein switch-Abschnitt mit dem nächsten switch-Abschnitt fortfährt, tritt ein Fehler während der Kompilierung auf, wenn der Endpunkt der Anweisungsliste eines switch-Abschnitts erreichbar ist. Wenn dieser Fehler auftritt, deutet dies in der Regel darauf hin, dass eine break-Anweisung fehlt.
* Wenn der Endpunkt des Blockes eines Funktionsmembers, der einen Wert berechnet, erreichbar ist, tritt ein Fehler während der Kompilierung auf. Wenn dieser Fehler auftritt, deutet dies in der Regel darauf hin, dass eine return-Anweisung fehlt.

## Blöcke

Ein Block ermöglicht das Schreiben von mehreren Anweisungen in Kontexten, in denen nur eine Anweisung zulässig ist.

block:  
{ statement-listopt }

Ein block besteht aus einer optionalen statement-list (§8.2.1), die in geschweiften Klammern eingeschlossen wird. Wenn die Anweisungsliste ausgelassen wird, handelt es sich um einen leeren Block.

Blöcke können Deklarationsanweisungen enthalten (§8.5). Der Gültigkeitsbereich einer in einem Block deklarierten lokalen Variablen oder Konstanten entspricht dem Block.

Innerhalb eines Blocks muss die Bedeutung eines Namens, der in einem Ausdruckskontext verwendet wird, immer gleich sein (§7.6.2.1).

Blöcke werden folgendermaßen ausgeführt:

* Wenn der Block leer ist, wird die Steuerung an den Endpunkt des Blockes übergeben.
* Wenn der Block nicht leer ist, wird die Steuerung an die Anweisungsliste übergeben. Wenn die Steuerung den Endpunkt der Anweisungsliste erreicht, wird sie an den Endpunkt des Blockes übergeben.

Die Anweisungsliste eines Blockes ist erreichbar, wenn der Block selbst erreichbar ist.

Der Endpunkt eines Blockes ist erreichbar, wenn der Block leer oder der Endpunkt der Anweisungsliste erreichbar ist.

Ein block, der mindestens eine yield-Anweisung (§8.14) enthält, wird als Iteratorblock bezeichnet. Iteratorblöcke werden zum Implementieren von Funktionsmembern als Iteratoren (§10.14) verwendet. Für Iteratorblöcke gelten einige zusätzliche Beschränkungen:

* Wenn eine return-Anweisung in einem Iteratorblock auftritt, führt dies zu einem Kompilierungsfehler (yield return-Anweisungen sind jedoch zulässig).
* Wenn ein Iteratorblock einen unsicheren Kontext enthält (§18.1), führt dies zu einem Kompilierungsfehler. Ein Iteratorblock definiert immer einen sicheren Kontext, auch wenn seine Deklaration in einem unsicheren Kontext geschachtelt ist.

### Anweisungslisten

Eine Anweisungsliste besteht aus einer oder mehreren nacheinander aufgeführten Anweisungen. Anweisungslisten kommen in blocks (§8.2) und in switch-blocks (§8.7.2) vor.

statement-list:  
statement  
statement-list statement

Anweisungslisten werden ausgeführt, indem die Steuerung an die erste Anweisung übergeben wird. Wenn die Steuerung den Endpunkt einer Anweisung erreicht, wird sie an die nächste Anweisung übergeben. Wenn die Steuerung den Endpunkt der letzten Anweisung erreicht, wird sie an den Endpunkt der Anweisungsliste übergeben.

Eine Anweisung in einer Anweisungsliste ist erreichbar, wenn eine der folgenden Bedingungen erfüllt ist:

* Es handelt sich bei der Anweisung um die erste Anweisung, und die Anweisungsliste selbst ist erreichbar.
* Der Endpunkt der vorangehenden Anweisung ist erreichbar.
* Es handelt sich bei der Anweisung um eine Anweisung mit Bezeichnung, und auf die Bezeichnung wird durch eine erreichbare goto-Anweisung verwiesen.

Der Endpunkt einer Anweisungsliste ist erreichbar, wenn der Endpunkt der letzten Anweisung in der Liste erreichbar ist.

## Die leere Anweisung

Bei einem empty-statement wird nichts ausgeführt.

empty-statement:  
;

Leere Anweisungen werden verwendet, wenn in einem Kontext, in dem eine Anweisung erforderlich ist, keine Operationen ausgeführt werden sollen.

Bei der Ausführung einer leeren Anweisung wird einfach die Steuerung an den Endpunkt der Anweisung übergeben. Somit ist der Endpunkt einer leeren Anweisung erreichbar, wenn die leere Anweisung erreichbar ist.

Eine leere Anweisung kann beim Schreiben einer while-Anweisung ohne Inhalt verwendet werden:

bool ProcessMessage() {...}

void ProcessMessages() {  
 while (ProcessMessage())  
 ;  
}

Leere Anweisungen können auch verwendet werden, um eine Bezeichnung direkt vor der schließenden geschweiften Klammer (}) eines Blocks zu deklarieren:

void F() {  
 ...

if (done) goto exit;  
 ...

exit: ;  
}

## Anweisung mit Bezeichnungen

Ein labeled-statement ermöglicht das Voranstellen einer Bezeichnung vor eine Anweisung. Anweisungen mit Bezeichnungen sind in Blöcken zulässig, können jedoch nicht als eingebettete Anweisungen verwendet werden.

labeled-statement:  
identifier : statement

Eine Anweisung mit Bezeichnung deklariert eine Bezeichnung mit dem Namen, der vom identifier angegeben wird. Der Gültigkeitsbereich einer Bezeichnung ist der gesamte Block, in dem die Bezeichnung deklariert wird. Dazu zählen auch alle geschachtelten Blöcke. Die Gültigkeitsbereiche von zwei Bezeichnungen mit demselben Namen dürfen sich nicht überschneiden, da andernfalls ein Fehler während der Kompilierung verursacht wird.

Auf eine Bezeichnung kann von einer goto-Anweisung (§8.9.3) innerhalb des Gültigkeitsbereichs der Bezeichnung verwiesen werden. Dies bedeutet, dass goto-Anweisungen die Steuerung unter keinen Umständen in die Blöcke hinein, sondern nur innerhalb von Blöcken und aus den Blöcken heraus übergeben können.

Bezeichnungen verfügen über eigene Deklarationsabschnitte und verursachen keine Konflikte mit anderen Bezeichnern. In dem Beispiel

int F(int x) {  
 if (x >= 0) goto x;  
 x = -x;  
 x: return x;  
}

ist gültig und verwendet den Namen x sowohl als Parameter als auch als Bezeichnung.

Die Ausführung einer Anweisung mit Bezeichnung entspricht der Ausführung der Anweisung, die auf die Bezeichnung folgt.

Abgesehen davon, dass die Erreichbarkeit aufgrund der normalen Ablaufsteuerung gegeben ist, sind Anweisungen mit Bezeichnungen erreichbar, wenn auf die Bezeichnung durch eine erreichbare goto-Anweisung verwiesen wird. (Ausnahme: Wenn sich eine goto-Anweisung innerhalb von try mit enthaltenem finally-Block befindet und sich die Anweisung mit Bezeichnung außerhalb von try befindet und der Endpunkt des finally-Blocks nicht erreichbar ist, dann ist die Anweisung mit Bezeichnung von der goto-Anweisung aus nicht erreichbar.)

## Deklarationsanweisungen

Mit einem declaration-statement werden lokale Variablen oder Konstanten deklariert. Deklarationsanweisungen sind in Blöcken zulässig, können jedoch nicht als eingebettete Anweisungen verwendet werden.

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

### Deklaration von lokalen Variablen

Eine local-variable-declaration deklariert eine oder mehrere lokale Variablen.

local-variable-declaration:  
local-variable-type local-variable-declarators

local-variable-type:  
type  
var

local-variable-declarators:  
local-variable-declarator  
local-variable-declarators , local-variable-declarator

local-variable-declarator:  
identifier  
identifier = local-variable-initializer

local-variable-initializer:  
expression  
array-initializer

Der local-variable-type einer local-variable-declaration gibt den Typ der durch die Deklaration eingeführten Variablen entweder direkt an, oder er gibt durch den Bezeichners var an, dass anhand des Initialisierers auf den Typ geschlossen werden soll. Auf den Typ folgt eine Liste von local-variable-declarators, die jeweils eine neue Variable einführen. Ein local-variable-declarator besteht aus einem identifier, der die Variable benennt, optional gefolgt von einem "="-Token und einem local-variable-initializer, die den Anfangswert der Variablen angibt.

Im Kontext einer lokalen Variablendeklaration fungiert der Bezeichner var als Kontextschlüsselwort (§2.4.3). Wenn der local-variable-type als var angegeben wird und sich kein Typ mit dem Namen var im Gültigkeitsbereich befindet, handelt es sich bei der Deklaration um eine iimplizit typisierte lokale Variablendeklaration, auf deren Typ aus dem Typ des zugehörigen Initialisierersausdrucks geschlossen wird. Für implizit typisierte lokale Variablendeklarationen gelten die folgenden Einschränkungen:

* Die local-variable-declaration darf nicht mehrere local-variable-declarators enthalten.
* Der local-variable-declarator muss einen local-variable-initializer enthalten.
* Der local-variable-initializer muss eine expression sein.
* Die expression für den Initialisierer muss über einen Kompilierungstyp verfügen.
* Die expression für den Initialisierer darf nicht auf die deklarierte Variable selbst verweisen.

Im Folgenden finden Sie einige Beispiel für falsche implizit typisierte Variablendeklarationen:

var x; // Error, no initializer to infer type from  
var y = {1, 2, 3}; // Error, array initializer not permitted  
var z = null; // Error, null does not have a type  
var u = x => x + 1; // Error, anonymous functions do not have a type  
var v = v++; // Error, initializer cannot refer to variable itself

Der Wert einer lokalen Variablen wird in einem Ausdruck ermittelt, indem ein simple-name (§7.6.2) verwendet und der Wert einer lokalen Variablen unter Verwendung eines assignment (§7.17) geändert wird. Eine lokale Variable muss an allen Positionen definitiv zugewiesen werden (§5.3), an der ein Wert ermittelt wird.

Der Gültigkeitsbereich einer durch eine local-variable-declaration deklarierten lokalen Variablen ist der Block, in dem die Deklaration erfolgt. Das Verweisen auf eine lokale Variable an einer Textposition vor dem local-variable-declarator der lokalen Variablen ist nicht zulässig und erzeugt einen Fehler. Innerhalb des Gültigkeitsbereichs einer lokalen Variablen darf keine weitere lokale Variable oder Konstante mit demselben Namen deklariert werden, da ansonsten ein Kompilierungsfehler auftritt.

Eine Deklaration von lokalen Variablen, die mehrere Variablen deklariert, ist äquivalent zu mehreren Deklarationen einzelner Variablen mit demselben Typ. Außerdem entspricht eine Variableninitialisierung in einer Deklaration von lokalen Variablen einer Zuweisungsanweisung, die unmittelbar nach der Deklaration eingefügt wird.

In dem Beispiel

void F() {  
 int x = 1, y, z = x \* 2;  
}

stimmt mit dem folgenden Beispiel überein

void F() {  
 int x; x = 1;  
 int y;  
 int z; z = x \* 2;  
}

Bei einer implizit typisierten lokalen Variablendeklaration wird davon ausgegangen, dass der Typ der deklarierten lokalen Variablen mit dem Typ des Ausdrucks übereinstimmt, mit dem die Variable initialisiert wird. Beispiel:

var i = 5;  
var s = "Hello";  
var d = 1.0;  
var numbers = new int[] {1, 2, 3};  
var orders = new Dictionary<int,Order>();

Die oben stehenden implizit typisierten lokalen Variablendeklarationen sind identisch mit den folgenden explizit typisierten Deklarationen:

int i = 5;  
string s = "Hello";  
double d = 1.0;  
int[] numbers = new int[] {1, 2, 3};  
Dictionary<int,Order> orders = new Dictionary<int,Order>();

### Deklaration von lokalen Konstanten

Eine local-constant-declaration deklariert eine oder mehrere lokale Konstanten.

local-constant-declaration:  
const type constant-declarators

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

Der type einer local-constant-declaration gibt den Typ der Konstanten an, die durch die Deklaration eingeführt wurden. Auf den Typ folgt eine Liste von constant-declarators, die jeweils eine neue Konstante einführen. Ein constant-declarator besteht aus einem identifier, der die Konstante benennt, und einem "="-Token, auf das eine constant-expression (§7.19) folgt, die den Wert der Konstanten angibt.

Der type und der constant-expression einer Deklaration von lokalen Konstanten muss mit den Regeln der Konstantenmemberdeklaration (§10.4) übereinstimmen.

Der Wert einer lokalen Konstanten wird in einem Ausdruck mithilfe eines simple-name (§7.6.2) ermittelt.

Der Gültigkeitsbereich einer lokalen Konstanten entspricht dem Block, in dem die Deklaration erfolgt. Wenn in einer Textposition vor dem constant-declarator auf eine lokale Konstante verwiesen wird, tritt ein Fehler auf. Innerhalb des Gültigkeitsbereichs einer lokalen Konstanten darf keine weitere lokale Variable oder Konstante mit demselben Namen deklariert werden, da ansonsten ein Kompilierungsfehler auftritt.

Eine Deklaration von lokalen Konstanten, die mehrere Konstanten deklariert, ist äquivalent mit mehreren Deklarationen einzelner Konstanten mit demselben Typ.

## Ausdrucksanweisungen

Ein expression-statement wertet einen bestimmten Ausdruck aus. Der von dem Ausdruck berechnete Wert wird ggf. verworfen.

expression-statement:  
statement-expression ;

statement-expression:  
invocation-expression  
object-creation-expression  
assignment  
post-increment-expression  
post-decrement-expression  
pre-increment-expression  
pre-decrement-expression  
await-expression

Nicht alle Ausdrücke sind zulässige Anweisungen. Insbesondere Ausdrücke wie x + y und x == 1, die lediglich einen Wert berechnen (der verworfen wird), sind als Anweisungen nicht zulässig.

Die Ausführung einer expression-statement wertet den enthaltenen expression aus und übergibt anschließend die Steuerung an den Endpunkt der expression-statement. Der Endpunkt einer expression-statement ist erreichbar, wenn diese expression-statement erreichbar ist.

## Auswahlanweisungen

Auswahlanweisungen wählen anhand des Ausdruckswertes eine Anweisung aus mehreren möglichen Anweisungen zur Ausführung aus.

selection-statement:  
if-statement  
switch-statement

### Die if-Anweisung

Die if-Anweisung wählt eine Anweisung für die Ausführung anhand des Werts eines booleschen Ausdrucks aus.

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

Ein else-Teil wird mit der if-Anweisung verknüpft, die auf lexikalischer Ebene direkt vorangeht und von der Syntax zugelassen wird. Daher ist eine if-Anweisung der Form

if (x) if (y) F(); else G();

äquivalent zu

if (x) {  
 if (y) {  
 F();  
 }  
 else {  
 G();  
 }  
}

Eine if-Anweisung wird folgendermaßen ausgeführt:

* Die boolean-expression (§7.20) wird ausgewertet.
* Wenn der boolesche Ausdruck true ergibt, wird die Steuerung an die erste eingebettete Anweisung übergeben. Wenn die Steuerung den Endpunkt dieser Anweisung erreicht, wird sie an den Endpunkt der if-Anweisung übergeben.
* Wenn der boolesche Ausdruck false ergibt und ein else-Teil vorhanden ist, wird die Steuerung an die zweite eingebettete Anweisung übergeben. Wenn die Steuerung den Endpunkt dieser Anweisung erreicht, wird sie an den Endpunkt der if-Anweisung übergeben.
* Wenn der boolesche Ausdruck false ergibt und kein else-Teil vorhanden ist, wird die Steuerung an den Endpunkt der if-Anweisung übergeben.

Die erste eingebettete Anweisung einer if-Anweisung ist erreichbar, wenn die if-Anweisung erreichbar ist und der boolesche Ausdruck nicht den konstanten Wert false hat.

Die zweite eingebettete Anweisung einer if-Anweisung ist ggf. erreichbar, wenn die if-Anweisung erreichbar ist und der boolesche Ausdruck nicht den konstanten Wert true hat.

Der Endpunkt einer if-Anweisung ist erreichbar, wenn der Endpunkt der letzten eingebetteten Anweisung erreichbar ist. Außerdem ist der Endpunkt einer if-Anweisung ohne else-Teil erreichbar, wenn die if-Anweisung erreichbar ist und der boolesche Ausdruck nicht den konstanten Wert true hat.

### Die switch-Anweisung

Die switch-Anweisung wählt für die Ausführung eine Anweisungsliste mit einer verknüpften switch-Bezeichnung aus, die mit dem Wert des switch-Ausdrucks übereinstimmt.

switch-statement:  
switch ( expression ) switch-block

switch-block:  
{ switch-sectionsopt }

switch-sections:  
switch-section  
switch-sections switch-section

switch-section:  
switch-labels statement-list

switch-labels:  
switch-label  
switch-labels switch-label

switch-label:  
case constant-expression :  
default :

Ein switch-statement besteht aus dem Schlüsselwort switch, gefolgt von einem in Klammern gesetzten Ausdruck (der als switch-Ausdruck bezeichnet wird), der wiederum von einem switch-block gefolgt wird. Der switch-block besteht aus keiner oder mehreren switch-sections, die in Klammern eingeschlossen sind. Jede switch-section besteht aus einem oder mehreren switch-labels gefolgt von einer statement-list (§8.2.1).

Der vorherrschende Typ einer switch-Anweisung wird durch den switch-Ausdruck festgelegt.

* Wenn es sich beim Typ des switch-Ausdrucks um sbyte, byte, short, ushort, int, uint, long, ulong, bool, char, string oder einen enum-type handelt oder wenn er der Typ ist, der NULL-Werte zulässt und mit einem dieser Typen übereinstimmt, gilt er als der vorherrschende Datentyp der switch-Anweisung.
* Andernfalls muss genau eine benutzerdefinierte implizite Konvertierung (§6.4) vom Typ des switch-Ausdrucks in einen der folgenden möglichen vorherrschenden Datentypen definiert sein: sbyte, byte, short, ushort, int, uint, long, ulong, char, string oder ein Typ, der NULL-Werte zulässt und mit einem dieser Typen übereinstimmt.
* Wenn entweder keine oder mehrere solcher impliziten Konvertierungen vorhanden sind, tritt andernfalls während der Kompilierung ein Fehler auf.

Der konstante Ausdruck jeder case-Bezeichnung muss auf einen Wert hindeuten, der implizit in den vorherrschenden Datentyp der switch-Anweisung konvertiert werden kann (§6.1). Wenn mindestens zwei case-Bezeichnungen in derselben switch-Anweisung denselben konstanten Wert angeben, tritt während der Kompilierung ein Fehler auf.

Eine switch-Anweisung darf höchstens eine default-Bezeichnung enthalten.

Eine switch-Anweisung wird folgendermaßen ausgeführt:

* Der switch-Ausdruck wird ausgewertet und in den vorherrschenden Datentyp konvertiert.
* Wenn eine der in der case-Bezeichnung angegebenen Konstanten in derselben switch-Anweisung mit dem Wert des switch-Ausdrucks übereinstimmt, wird die Steuerung an die Anweisungsliste übergeben, die auf die case-Bezeichnung folgt.
* Wenn keine der in den case-Bezeichnungen angegebenen Konstanten in derselben switch-Anweisung mit dem Wert des switch-Ausdrucks übereinstimmt und eine default-Bezeichnung vorhanden ist, wird die Steuerung an die Anweisungsliste übergeben, die auf die default-Bezeichnung folgt.
* Wenn keine der in case-Bezeichnungen angegebenen Konstanten in derselben switch-Anweisung mit dem Wert des switch-Ausdrucks übereinstimmt und keine default-Bezeichnung vorhanden ist, wird die Steuerung an den Endpunkt der switch-Anweisung übergeben.

Wenn der Endpunkt der Anweisungsliste eines switch-Abschnitts erreichbar ist, tritt während der Kompilierung ein Fehler auf. Dies wird als „No-Fall-Through“-Regel bezeichnet. In dem Beispiel

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
default:  
 CaseOthers();  
 break;  
}

ist gültig, weil kein switch-Abschnitt einen erreichbaren Endpunkt enthält. Anders als in C und C++ darf die Ausführung eines switch-Abschnitts nicht mit dem nächsten switch-Abschnitt fortfahren, sodass das Beispiel

switch (i) {  
case 0:  
 CaseZero();  
case 1:  
 CaseZeroOrOne();  
default:  
 CaseAny();  
}

führt zu einem Kompilierungsfehler. Wenn die Ausführung eines switch-Abschnitts von der Ausführung eines anderen switch-Abschnitts gefolgt werden soll, muss eine explizite goto case- oder goto default-Anweisung verwendet werden:

switch (i) {  
case 0:  
 CaseZero();  
 goto case 1;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
default:  
 CaseAny();  
 break;  
}

In einer switch-section sind mehrere Bezeichnungen zulässig. In dem Beispiel

switch (i) {  
case 0:  
 CaseZero();  
 break;  
case 1:  
 CaseOne();  
 break;  
case 2:  
default:  
 CaseTwo();  
 break;  
}

ist gültig. Es verstößt nicht gegen die "No-Fall-Through"-Regel, da die Bezeichnungen case 2: und default: Teil derselben switch-section sind.

Die "No-Fall-Through"-Regel verhindert eine bestimmte Klasse von Fehlern, die in C und C++ auftreten, wenn break-Anweisungen versehentlich ausgelassen werden. Aufgrund dieser Regel können die switch-Abschnitte einer switch-Anweisung außerdem beliebig neu angeordnet werden, ohne dass sich dies auf das Verhalten der Anweisung auswirkt. So können die Abschnitte der oben genannten switch-Anweisung beispielsweise umgekehrt werden, ohne dass sich dies auf das Verhalten der Anweisung auswirkt:

switch (i) {  
default:  
 CaseAny();  
 break;  
case 1:  
 CaseZeroOrOne();  
 goto default;  
case 0:  
 CaseZero();  
 goto case 1;  
}

Die Anweisungsliste eines switch-Abschnitts endet zwar in der Regel mit einer break-, goto- case- oder goto default-Anweisung, aber prinzipiell sind alle Konstrukte zulässig, die den Endpunkt der Anweisung unerreichbar machen. Beispielsweise erreicht eine while-Anweisung, die von dem booleschen Ausdruck true gesteuert wird, nie ihren Endpunkt. Eine throw- oder return-Anweisung übergibt immer die Steuerung und erreicht ebenfalls nie ihren Endpunkt. Folglich ist das folgende Beispiel gültig:

switch (i) {  
case 0:  
 while (true) F();  
case 1:  
 throw new ArgumentException();  
case 2:  
 return;  
}

Der vorherrschende Datentyp einer switch-Anweisung kann der Typ string sein. Beispiel:

void DoCommand(string command) {  
 switch (command.ToLower()) {  
 case "run":  
 DoRun();  
 break;  
 case "save":  
 DoSave();  
 break;  
 case "quit":  
 DoQuit();  
 break;  
 default:  
 InvalidCommand(command);  
 break;  
 }  
}

Ähnlich wie bei den Gleichheitsoperatoren für Zeichenfolgen (§7.10.7) muss bei der switch-Anweisung die Groß-/Kleinschreibung beachtet werden. Außerdem führt diese Anweisung einen bestimmten switch-Abschnitt nur aus, wenn die switch-Ausdruckszeichenfolge mit einer case-Bezeichnungskonstante übereinstimmt.

Wenn der vorherrschende Datentyp einer switch-Anweisung string lautet, ist der Wert null als case-Bezeichnungskonstante zulässig.

Die statement-lists eines switch-block können Deklarationsanweisungen enthalten (§8.5). Der Gültigkeitsbereich einer in einem switch-Block deklarierten lokalen Variablen oder Konstanten entspricht dem switch-Block.

Innerhalb eines switch-Blocks muss die Bedeutung eines Namens, der in einem Ausdruckskontext verwendet wird, immer gleich sein (§7.6.2.1).

Die Anweisungsliste eines bestimmten switch-Abschnitts ist erreichbar, wenn die switch-Anweisung erreichbar ist und mindestens eine der folgenden Bedingungen erfüllt wird:

* Der switch-Ausdruck ist kein konstanter Wert.
* Der switch-Ausdruck ist ein konstanter Wert, der mit einer case-Bezeichnung im switch-Abschnitt übereinstimmt.
* Der switch-Ausdruck ist ein konstanter Wert, der mit keiner case-Bezeichnung übereinstimmt, und der switch-Abschnitt enthält die default-Bezeichnung.
* Auf eine switch-Bezeichnung des switch-Abschnitts wird von einer erreichbaren goto case-Anweisung oder goto default-Anweisung verwiesen.

Der Endpunkt einer switch-Anweisung ist erreichbar, wenn mindestens eine der folgenden Bedingungen erfüllt wird:

* Die switch-Anweisung enthält eine erreichbare break-Anweisung, die die switch-Anweisung beendet.
* Die switch-Anweisung ist erreichbar, der swich-Ausdruck ist kein konstanter Wert, und es ist keine default-Bezeichnung vorhanden.
* Die switch-Anweisung ist erreichbar, der switch-Ausdruck ist ein konstanter Wert, der mit keiner case-Bezeichnung übereinstimmt, und es ist keine default-Bezeichnung vorhanden.

## Iterationsanweisungen

Iterationsanweisungen führen eine eingebettete Anweisung wiederholt aus.

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

### Die while-Anweisung

Die while-Anweisung führt bedingt eine eingebettete Anweisung entweder mehrere Male oder gar nicht aus.

while-statement:  
while ( boolean-expression ) embedded-statement

Eine while-Anweisung wird folgendermaßen ausgeführt:

* Die boolean-expression (§7.20) wird ausgewertet.
* Wenn der boolesche Ausdruck true ergibt, wird die Steuerung an die eingebettete Anweisung übergeben. Wenn die Steuerung den Endpunkt der eingebetteten Anweisung erreicht (möglichst durch Ausführen einer continue-Anweisung), wird sie an den Anfang der while-Anweisung übergeben.
* Wenn der boolesche Ausdruck false ergibt, wird die Steuerung an den Endpunkt der while-Anweisung übergeben.

Innerhalb der eingebetteten Anweisung einer while-Anweisung kann die Steuerung mithilfe einer break-Anweisung (§8.9.1) an den Endpunkt der while-Anweisung übergeben werden (wodurch die Iteration der eingebetteten Anweisung beendet wird). Durch eine continue-Anweisung (§8.9.2) kann die Steuerung entsprechend an den Endpunkt der eingebetteten Anweisung übergeben werden (wodurch eine weitere Iteration der while-Anweisung ausgeführt wird).

Die eingebettete Anweisung einer while-Anweisung ist erreichbar, wenn die while-Anweisung erreichbar ist und der boolesche Ausdruck nicht den konstanten Wert false hat.

Der Endpunkt einer while-Anweisung ist erreichbar, wenn mindestens eine der folgenden Bedingungen erfüllt wird:

* Die while-Anweisung enthält eine erreichbare break-Anweisung, die die while-Anweisung beendet.
* Die while-Anweisung ist erreichbar, und der boolesche Ausdruck hat nicht den konstanten Wert true.

### Die do-Anweisung

Die do-Anweisung führt bedingt eine eingebettete Anweisung mehrere Male oder gar nicht aus.

do-statement:  
do embedded-statement while ( boolean-expression ) ;

Eine do-Anweisung wird folgendermaßen ausgeführt:

* Die Steuerung wird an die eingebettete Anweisung übergeben.
* Wenn die Steuerung den Endpunkt der eingebetteten Anweisung erreicht (möglicherweise durch Ausführen einer continue-Anweisung), wird die boolean-expression (§7.20) ausgewertet. Wenn der boolesche Ausdruck true ergibt, wird die Steuerung an den Anfang der do-Anweisung übergeben. Andernfalls wird die Steuerung an den Endpunkt der do-Anweisung übergeben.

Innerhalb der eingebetteten Anweisung einer do-Anweisung kann die Steuerung mithilfe einer break-Anweisung (§8.9.1) an den Endpunkt der do-Anweisung übergeben werden (wodurch die Iteration der eingebetteten Anweisung beendet wird). Durch eine continue-Anweisung (§8.9.2) kann die Steuerung entsprechend an den Endpunkt der eingebetteten Anweisung übergeben werden.

Die eingebettete Anweisung einer do-Anweisung ist erreichbar, wenn die do-Anweisung erreichbar ist.

Der Endpunkt einer do-Anweisung ist erreichbar, wenn mindestens eine der folgenden Bedingungen erfüllt wird:

* Die do-Anweisung enthält eine erreichbare break-Anweisung, die die do-Anweisung beendet.
* Der Endpunkt der eingebetteten Anweisung ist erreichbar, und der boolesche Ausdruck hat nicht den konstanten Wert true.

### Die for-Anweisung

Die for-Anweisung wertet eine Reihe von Initialisierungsausdrücken aus. Wenn dabei eine Bedingung erfüllt wird, führt sie eine eingebettete Anweisung wiederholt aus und wertet eine Reihe von Iterationsausdrücken aus.

for-statement:  
for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:  
local-variable-declaration  
statement-expression-list

for-condition:  
boolean-expression

for-iterator:  
statement-expression-list

statement-expression-list:  
statement-expression  
statement-expression-list , statement-expression

Sofern vorhanden, besteht der for-initializer entweder aus einer local-variable-declaration (§8.5.1) oder aus einer durch Trennzeichen getrennten Liste mit statement-expressions (§8.6). Der Gültigkeitsbereich einer lokalen Variablen, die von einem for-initializer deklariert wird, beginnt beim local-variable-declarator der Variablen und reicht bis zum Ende der eingebetteten Anweisung. Der Gültigkeitsbereich umfasst die for-condition und den for-iterator.

Bei der for-condition muss es sich (sofern vorhanden) um eine boolean-expression (§7.20) handeln.

Sofern vorhanden, besteht der for-iterator aus einer durch Trennzeichen getrennten Liste mit statement-expressions (§8.6).

Eine for-Anweisung wird folgendermaßen ausgeführt:

* Wenn ein for-initializer vorhanden ist, werden die Variableninitialisierungen bzw. Anweisungsausdrücke in der Reihenfolge ausgeführt, in der sie geschrieben wurden. Dieser Schritt wird nur einmal ausgeführt.
* Wenn eine for-condition vorhanden ist, wird diese ausgewertet.
* Wenn keine for-condition vorhanden ist oder das Ergebnis der Auswertung true ist, wird die Steuerung an die eingebettete Anweisung übergeben. Wenn die Steuerung den Endpunkt der eingebetteten Anweisung erreicht (möglichst durch Ausführen einer continue-Anweisung), werden die Ausdrücke des for-iterator ggf. nacheinander ausgewertet, und anschließend wird durch Auswertung der for-condition aus dem oben genannten Schritt eine weitere Iteration ausgeführt.
* Wenn die for-condition vorhanden und das Ergebnis der Auswertung false ist, wird die Steuerung an den Endpunkt der for-Anweisung übergeben.

Innerhalb der eingebetteten Anweisung einer for-Anweisung kann die Steuerung mithilfe einer break-Anweisung (§8.9.1) an den Endpunkt der for-Anweisung übergeben werden (wodurch die Iteration der eingebetteten Anweisung beendet wird). Durch eine continue-Anweisung (§8.9.2) kann die Steuerung entsprechend an den Endpunkt der eingebetteten Anweisung übergeben werden (wodurch der for-iterator und eine weitere Iteration der for-Anweisung beginnend mit der for-condition ausgeführt wird).

Die eingebettete Anweisung einer for-Anweisung ist erreichbar, wenn eine der folgenden Bedingungen erfüllt wird:

* Die for-Anweisung ist erreichbar, und es ist keine for-condition vorhanden.
* Die for-Anweisung ist erreichbar, und es ist eine for-condition vorhanden, die nicht den konstanten Wert false hat.

Der Endpunkt einer for-Anweisung ist erreichbar, wenn mindestens eine der folgenden Bedingungen erfüllt wird:

* Die for-Anweisung enthält eine erreichbare break-Anweisung, die die for-Anweisung beendet.
* Die for-Anweisung ist erreichbar, und es ist eine for-condition vorhanden, die nicht den konstanten Wert true hat.

### Die foreach-Anweisung

Die foreach-Anweisung zählt die Elemente einer Auflistung auf, indem für jedes Element der Auflistung eine eingebettete Anweisung ausgeführt wird.

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

Der type und identifier einer foreach-Anweisung deklarieren die Iterationsvariable der Anweisung. Wenn als local-variable-type der var-Bezeichner angegeben wird und sich kein Typ mit dem Namen var im Gültigkeitsbereich befindet, gilt die Iterationsvariable als eine implizit typisierte Iterationsvariable, deren Typ als Elementtyp der foreach-Anweisung übernommen wird, wie unten dargestellt. Die Iterationsvariable entspricht einer schreibgeschützten lokalen Variablen mit einem Gültigkeitsbereich, der sich über die eingebettete Anweisung erstreckt. Während der Ausführung einer foreach-Anweisung stellt die Iterationsvariable das Auflistungselement dar, für das die Iteration gegenwärtig ausgeführt wird. Wenn die eingebettete Anweisung versucht, eine Änderung der Iterationsvariablen vorzunehmen (durch Zuweisung bzw. die Operatoren ++ und ‑‑) oder die Iterationsvariable als ref- oder out-Parameter weiterzugeben, tritt während der Kompilierung ein Fehler auf.

Aus Platzgründen verweisen IEnumerable, IEnumerator, IEnumerable<T> und IEnumerator<T> im Folgenden auf die entsprechenden Typen in den Namespaces System.Collections und System.Collections.Generic.

Bei der Verarbeitung einer foreach-Anweisung während der Kompilierung wird zunächst der Auflistungstyp, der Enumeratortyp und der Elementtyp des Ausdrucks bestimmt. Dieser bestimmt die weitere Verarbeitung wie folgt:

* Wenn der Typ X der expression ein Arraytyp ist, ist eine implizite Verweiskonvertierung von X in die IEnumerable-Schnittstelle vorhanden (da System.Array diese Schnittstelle implementiert). Der Auflistungstyp ist die IEnumerable-Schnittstelle, der Enumeratortyp ist die IEnumerator-Schnittstelle, und der Elementtyp ist der Elementtyp des Arraytyps X.
* Wenn der Typ X der expression dynamic ist, ist eine implizite Verweiskonvertierung von expression in die IEnumerable-Schnittstelle vorhanden (§6.1.8). Der Auflistungstyp ist die IEnumerable-Schnittstelle, und der Enumeratortyp ist die IEnumerator-Schnittstelle. Wenn der var-Bezeichner als local-variable-type angegeben wird, ist der Elementtyp dynamic, andernfalls ist er object.
* Andernfalls wird bestimmt, ob der Typ X über eine passende GetEnumerator-Methode verfügt:
* Es wird eine Membersuche für den Typ X mit dem Bezeichner GetEnumerator und ohne Typargumente durchgeführt. Wenn die Membersuche kein Ergebnis, ein mehrdeutiges Ergebnis oder ein Ergebnis liefert, das keine Methodengruppe ist, wird eine Überprüfung auf eine aufzählbare Schnittstelle durchgeführt, wie unten beschrieben. Es wird empfohlen, eine Warnung auszugeben, falls bei der Membersuche entweder keine Methodengruppe oder gar kein Ergebnis gefunden wird.
* Es wird eine Überladungsauflösung mit der Ergebnismethodengruppe und einer leeren Argumentliste durchgeführt. Wenn die Überladungsauflösung keine anwendbaren Methoden, ein mehrdeutiges Ergebnis oder eine optimale Methode liefert, die jedoch entweder statisch oder nicht öffentlich ist, wird eine Überprüfung auf eine aufzählbare Schnittstelle durchgeführt, wie unten beschrieben. Es wird empfohlen, eine Warnung auszugeben, falls bei der Überladungsauflösung entweder eine eindeutige öffentliche Instanzmethode oder keine anwendbare Methode gefunden wird.
* Wenn der Rückgabetyp E der GetEnumerator-Methode kein Klassen-, Struktur- oder Schnittstellentyp ist, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.
* Die Membersuche für E wird mit dem Bezeichner Current und ohne Typargumente durchgeführt. Wenn die Membersuche kein Ergebnis liefert, einen Fehler generiert oder ein Ergebnis liefert, das keine öffentliche Instanzeigenschaft ist, die das Lesen zulässt, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.
* Die Membersuche für E wird mit dem Bezeichner MoveNext und ohne Typargumente durchgeführt. Wenn die Membersuche kein Ergebnis liefert, einen Fehler generiert oder ein Ergebnis liefert, das keine Methodengruppe ist, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.
* Es wird eine Überladungsauflösung für die Methodengruppe mit einer leeren Argumentliste durchgeführt. Wenn die Überladungsauflösung keine anwendbaren Methoden, ein mehrdeutiges Ergebnis oder eine optimale Methode liefert, die jedoch entweder statisch oder nicht öffentlich ist oder deren Rückgabetyp nicht bool ist, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.
* Der Auflistungstyp ist X, der Enumeratortyp is E, und der Elementtyp ist der Typ der Current-Eigenschaft.
* Wenn diese Punkte nicht zutreffen, wird eine Überprüfung auf eine aufzählbare Schnittstelle durchgeführt:
* Wenn für alle Typen Ti, für die eine implizite Konvertierung von X in IEnumerable<Ti> vorhanden ist, ein eindeutiger Typ T existiert, sodass T nicht dynamic ist, und für alle anderen Ti eine implizite Konvertierung von IEnumerable<T> in IEnumerable<Ti> vorhanden ist, dann entspricht der Auflistungstyp der Schnittstelle IEnumerable<T>, der Enumeratortyp der Schnittstelle IEnumerator<T> und der Elementtyp ist T.
* Sind mehrere solcher Typen T vorhanden, so wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.
* Wenn andernfalls eine implizite Konvertierung von X in die System.Collections.IEnumerable-Schnittstelle vorhanden ist, so ist der Auflistungstyp diese Schnittstelle, der Enumeratortyp die System.Collections.IEnumerator-Schnittstelle und der Elementtyp der Typ object.
* Andernfalls wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt.

Bei erfolgreicher Ausführung liefern die oben stehenden Schritte einen eindeutigen Auflistungstyp C, einen eindeutigen Enumeratortyp E und einen eindeutigen Elementtyp T. Eine foreach-Anweisung der Form

foreach (V v in x) embedded-statement

wird erweitert in:

{  
 E e = ((C)(x)).GetEnumerator();  
 try {  
 while (e.MoveNext()) {  
 V v = (V)(T)e.Current;  
 embedded-statement  
 }  
 }  
 finally {  
 … // Dispose e  
 }  
}

Die Variable e ist für den Ausdruck x, die eingebettete Anweisung und anderen Quellcode des Programms weder sichtbar noch zugreifbar. Die Variable v ist in der eingebetteten Anweisung schreibgeschützt. Wenn keine explizite Konvertierung (§6.2) von T (dem Elementtyp) in V (den local-variable-type in der foreach-Anweisung) vorhanden ist, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt. Wenn der Wert von x gleich null ist, wird zur Laufzeit System.NullReferenceException ausgelöst.

Bei einer Implementierung kann eine foreach-Anweisung auch auf andere Weise implementiert werden, z. B. aus Leistungsgründen, sofern das Verhalten mit der oben stehenden Erweiterung übereinstimmt.

Die Platzierung von v innerhalb der While-Schleife ist wichtig dafür, wie sie von einer anonymen Funktion erfasst wird, die im embedded-statement auftritt.

Beispiel:

int[] values = { 7, 9, 13 };  
Action f = null;

foreach (var value in values)  
{  
 if (f == null) f = () => Console.WriteLine("First value: " + value);  
}

f();

Wenn v außerhalb der While-Schleife deklariert wäre, würde sie für alle Iterationen freigegeben, und der zugehörige Wert nach der For-Schleife wäre der Anfangswert 13, der beim Aufrufen von f ausgegeben würde. Da stattdessen jede Iteration ihre eigene Variable v besitzt, enthält die von f in der ersten Iteration erfasste Variable weiterhin den Wert 7, der ausgegeben wird. (Hinwei: In früheren Versionen von C# wurde v außerhalb der While-Schleife deklariert.)

Der Text des finally-Blocks wird entsprechend den folgenden Schritten erstellt:

* Wenn keine implizite Konvertierung von E in die the System.IDisposable-Schnittstelle stattfindet:
* Wenn E ein Werttyp ist, der keine NULL-Werte zulässt, wird die finally-Klausel erweitert auf eine semantische Entsprechung von:

finally {  
 ((System.IDisposable)e).Dispose();  
}

* Andernfalls wird die finally-Klausel erweitert auf eine semantische Entsprechung von:

finally {  
 if (e != null) ((System.IDisposable)e).Dispose();  
}

Wobei jedoch, wenn E ein Werttyp oder ein in einen Werttyp instanziierter Typparameter ist, bei der Umwandlung von e in System.IDisposable kein Boxing auftritt.

* Wenn andernfalls E ein versiegelter Typ ist, wird die finally-Klausel auf einen leeren Block erweitert:

finally {  
}

* Trifft dies nicht zu, wird die finally-Klausel wie folgt erweitert:

finally {  
 System.IDisposable d = e as System.IDisposable;  
 if (d != null) d.Dispose();  
}

Die lokale Variable d ist für Benutzercode weder sichtbar noch zugreifbar. Sie führt auch zu keinen Konflikten mit anderen Variablen, deren Gültigkeitsbereich den finally-Block einschließt.

Die Elemente eines Arrays werden von foreach in der folgenden Reihenfolge durchlaufen: In eindimensionalen Arrays werden Elemente in aufsteigender Indexreihenfolge durchlaufen, beginnend mit Index 0 und abschließend mit Index Length – 1. Die Elemente mehrdimensionaler Arrays werden so durchlaufen, dass die Indizes in folgender Reihenfolge erhöht werden: zuerst die Indizes der am weitesten rechts stehenden Dimension, dann die der nächsten links davon liegenden Dimension usw.

Im folgenden Beispiel wird jeder Wert in einem zweidimensionalen Array in der Reihenfolge der Elemente ausgegeben:

using System;

class Test  
{  
 static void Main() {  
 double[,] values = {  
 {1.2, 2.3, 3.4, 4.5},  
 {5.6, 6.7, 7.8, 8.9}  
 };

foreach (double elementValue in values)  
 Console.Write("{0} ", elementValue);

Console.WriteLine();  
 }  
}

Die Ausgabe lautet:

1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9

In dem Beispiel

int[] numbers = { 1, 3, 5, 7, 9 };  
foreach (var n in numbers) Console.WriteLine(n);

wird für n auf den Typ int geschlossen, dem Elementtyp von numbers.

## Sprunganweisungen

Sprunganweisungen übertragen die Steuerung unabhängig von den Bedingungen.

jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement

Die Position, an die eine Sprunganweisung die Steuerung übergibt, wird als Ziel der Sprunganweisung bezeichnet.

Wenn eine Sprunganweisung innerhalb eines Blocks auftritt und sich das Ziel der Sprunganweisung außerhalb dieses Blocks befindet, beendet die Sprunganweisung den Block. Sprunganweisungen können die Steuerung zwar aus einem Block heraus, jedoch nie in einen Block hinein übergeben.

Die Ausführung einer Sprunganweisung ist wegen der dazwischen liegenden try-Anweisungen kompliziert. Wenn keine solche try-Anweisungen vorhanden sind, übergibt eine Sprunganweisung die Steuerung unabhängig von den Bedingungen an ihr Ziel. Wenn try-Anweisungen dazwischen liegen, ist die Ausführung komplex. Beendet die Sprunganweisung einen oder mehrere try-Blöcke mit verknüpften finally-Blöcken, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle dazwischen liegenden try-Anweisungen ausgeführt wurden.

In dem Beispiel

using System;

class Test  
{  
 static void Main() {  
 while (true) {  
 try {  
 try {  
 Console.WriteLine("Before break");  
 break;  
 }  
 finally {  
 Console.WriteLine("Innermost finally block");  
 }  
 }  
 finally {  
 Console.WriteLine("Outermost finally block");  
 }  
 }  
 Console.WriteLine("After break");  
 }  
}

werden die finally-Blöcke, die mit zwei try-Anweisungen verknüpft sind, vor der Übergabe der Steuerung an das Ziel der Sprunganweisung ausgeführt.

Die Ausgabe lautet:

Before break  
Innermost finally block  
Outermost finally block  
After break

### Die break-Anweisung

Die break-Anweisung beendet die nächste einschließende switch-, while-, do-, for- oder foreach-Anweisung.

break-statement:  
break ;

Das Ziel einer break-Anweisung ist der Endpunkt der nächsten einschließenden switch-, while-, do-, for- oder foreach-Anweisung. Wenn eine break-Anweisung nicht von einer switch-, while-, do-, for- oder foreach-Anweisung eingeschlossen ist, tritt während der Kompilierung ein Fehler auf.

Wenn mehrere switch-, while-, do-, for- oder foreach-Anweisungen ineinander geschachtelt sind, trifft die break-Anweisung nur für die innerste Anweisung zu. Um die Steuerung über mehrere geschachtelte Ebenen hinweg übertragen zu können, muss eine goto-Anweisung verwendet werden (§8.9.3).

Eine break-Anweisung kann keine finally-Blöcke beenden (§8.10). Wenn eine break-Anweisung innerhalb eines finally-Blocks auftritt, muss sich das Ziel der break-Anweisung innerhalb desselben finally-Blocks befinden. Andernfalls tritt während der Kompilierung ein Fehler auf.

Eine break-Anweisung wird folgendermaßen ausgeführt:

* Beendet die break-Anweisung einen oder mehrere try-Blöcke mit verknüpften finally-Blöcken, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle dazwischen liegenden try-Anweisungen ausgeführt wurden.
* Die Steuerung wird an das Ziel der break-Anweisung übergeben.

Da eine break-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer break-Anweisung nie erreichbar.

### Die continue-Anweisung

Die continue-Anweisung startet eine neue Iteration der nächsten einschließenden while-, do-, for- oder foreach-Anweisung.

continue-statement:  
continue ;

Das Ziel einer continue-Anweisung ist der Endpunkt der eingebetteten Anweisung der nächsten einschließenden while-, do-, for- oder foreach-Anweisung. Wenn eine continue-Anweisung nicht von einer while-, do-, for- oder foreach-Anweisung eingeschlossen ist, tritt während der Kompilierung ein Fehler auf.

Wenn mehrere while-, do-, for- oder foreach-Anweisungen ineinander geschachtelt sind, trifft die continue-Anweisung nur für die innerste Anweisung zu. Um die Steuerung über mehrere geschachtelte Ebenen hinweg übertragen zu können, muss eine goto-Anweisung verwendet werden (§8.9.3).

Eine continue-Anweisung kann keine finally-Blöcke beenden (§8.10). Wenn eine continue-Anweisung innerhalb eines finally-Blocks auftritt, muss sich das Ziel der continue-Anweisung innerhalb desselben finally-Blocks befinden. Andernfalls tritt während der Kompilierung ein Fehler auf.

Eine continue-Anweisung wird folgendermaßen ausgeführt:

* Beendet die continue-Anweisung einen oder mehrere try-Blöcke mit verknüpften finally-Blöcken, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle dazwischen liegenden try-Anweisungen ausgeführt wurden.
* Die Steuerung wird an das Ziel der continue-Anweisung übergeben.

Da eine continue-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer continue-Anweisung nie erreichbar.

### Die goto-Anweisung

Die goto-Anweisung übergibt die Steuerung an eine Anweisung mit einer Bezeichnung.

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

Das Ziel einer goto identifier-Anweisung ist die Anweisung mit der angegebenen Bezeichnung. Wenn eine Bezeichnung mit dem angegebenen Namen in dem aktuellen Funktionsmember nicht vorhanden ist oder die goto-Anweisung sich nicht im Gültigkeitsbereich der Bezeichnung befindet, tritt während der Kompilierung ein Fehler auf. Diese Regel ist die Voraussetzung dafür, dass die Steuerung mithilfe einer goto-Anweisung aus einem geschachtelten Gültigkeitsbereich heraus, aber nicht in einen geschachtelten Gültigkeitsbereich hinein übergeben werden kann. In dem Beispiel

using System;

class Test  
{  
 static void Main(string[] args) {  
 string[,] table = {  
 {"Red", "Blue", "Green"},  
 {"Monday", "Wednesday", "Friday"}  
 };

foreach (string str in args) {  
 int row, colm;  
 for (row = 0; row <= 1; ++row)  
 for (colm = 0; colm <= 2; ++colm)  
 if (str == table[row,colm])  
 goto done;

Console.WriteLine("{0} not found", str);  
 continue;  
 done:  
 Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);  
 }  
 }  
}

wird die Steuerung mithilfe einer goto-Anweisung aus einem geschachtelten Gültigkeitsbereich heraus übergeben.

Das Ziel einer goto case-Anweisung ist die Anweisungsliste in der unmittelbar einschließenden switch-Anweisung (§8.7.2), die eine case-Bezeichnung mit dem angegebenen konstanten Wert enthält. Wenn die goto case-Anweisung nicht von einer switch-Anweisung eingeschlossen ist, die constant-expression nicht implizit in den vorherrschenden Datentyp der nächsten einschließenden switch-Anweisung konvertiert werden kann (§6.1) oder die nächste einschließende switch-Anweisung keine case-Bezeichnung mit dem angegebenen konstanten Wert enthält, tritt bei der Kompilierung ein Fehler auf.

Das Ziel einer goto default-Anweisung ist die Anweisungsliste in der unmittelbar einschließenden switch-Anweisung (§8.7.2), die eine default-Bezeichnung enthält. Wenn die goto default-Anweisung nicht von einer switch-Anweisung eingeschlossen wird oder die nächste einschließende switch-Anweisung keine default-Bezeichnung enthält, tritt während der Kompilierung ein Fehler auf.

Eine goto-Anweisung kann keine finally-Blöcke beenden (§8.10). Wenn eine goto-Anweisung innerhalb eines finally-Blocks auftritt, muss sich das Ziel der goto-Anweisung innerhalb desselben finally-Blocks befinden. Andernfalls tritt während der Kompilierung ein Fehler auf.

Eine goto-Anweisung wird folgendermaßen ausgeführt:

* Beendet die goto-Anweisung einen oder mehrere try-Blöcke mit verknüpften finally-Blöcken, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle dazwischen liegenden try-Anweisungen ausgeführt wurden.
* Die Steuerung wird an das Ziel der goto-Anweisung übergeben.

Da eine goto-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer goto-Anweisung nie erreichbar.

### Die return-Anweisung

Die return-Anweisung gibt die Steuerung an den aktuellen Aufrufer der Funktion zurück, in der die return-Anweisung auftritt.

return-statement:  
return expressionopt ;

Eine return-Anweisung ohne Ausdruck kann nur in einem Funktionsmember verwendet werden, der keinen Wert berechnet. Dazu zählen Methoden mit dem Ergebnistyp (§10.6.10) void, der set-Accessor einer Eigenschaft oder eines Indexers, der add- und remove-Accessor eines Ereignisses, Instanzkonstruktoren, statische Konstruktoren oder Destruktoren.

Eine return -Anweisung mit einem Ausdruck kann nur in einem Funktionsmember verwendet werden, der einen Wert berechnet. Dazu zählen Methoden mit einem nicht leeren Ergebnistyp, der get-Accessor einer Eigenschaft oder eines Indexers oder benutzerdefinierte Operatoren. Es muss eine implizite Konvertierung (§6.1) aus dem Typ des Ausdrucks in den Rückgabetyp des enthaltenden Funktionsmembers vorhanden sein.

Rückgabeanweisungen können auch im Text von anonymen Funktionsausdrücken (§7.15) verwendet werden und helfen dabei zu bestimmen, welche Konvertierungen für diese Funktionen vorhanden sind.

In einem finally-Block (§8.10) darf keine return-Anweisung auftreten, andernfalls wird ein Kompilierungsfehler erzeugt.

Eine return-Anweisung wird folgendermaßen ausgeführt:

* Wenn die return-Anweisung einen Ausdruck angibt, wird zunächst der Ausdruck ausgewertet und der sich daraus ergebende Wert dann in einem impliziten Konvertierungsverfahren in den Rückgabetyp der enthaltenden Funktion konvertiert. Das Ergebnis der Konvertierung erhält den durch die Funktion erzeugten Ergebniswert.
* Wenn die return-Anweisung in einen oder mehrere try- oder catch-Blöcke mit verknüpften finally-Blöcken eingeschlossen ist, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle einschließenden try-Anweisungen ausgeführt wurden.
* Wenn die enthaltende Funktion keine Async-Funktion ist, wird die Steuerung an den Aufrufer der enthaltenden Funktion zusammen mit dem Ergebniswert (soweit vorhanden) zurückgegeben.
* Wenn die enthaltende Funktion eine Async-Funktion ist, wird die Steuerung an den aktuellen Aufrufer zurückgegeben, und der Ergebniswert (soweit vorhanden) wird wie in §10.14.1 erläutert in der Rückgabeaufgabe aufgezeichnet.

Da eine return-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer return-Anweisung nie erreichbar.

### Die throw-Anweisung

Die throw-Anweisung löst eine Ausnahme aus.

throw-statement:  
throw expressionopt ;

Eine throw-Anweisung mit einem Ausdruck löst den Wert aus, der beim Auswerten des Ausdrucks ermittelt wird. Der Ausdruck muss auf einen Wert des Klassentyps System.Exception, eines Klassentyps, der von System.Exception abgeleitet ist, oder eines Typparametertyps deuten, dessen effektive Basisklasse System.Exception (oder eine Unterklasse dieser) ist. Wenn das Ergebnis der Auswertung des Ausdrucks null ist, wird stattdessen System.NullReferenceException ausgelöst.

Eine throw-Anweisung ohne Ausdruck kann nur in einem catch-Block verwendet werden. In diesem Fall wird erneut die Ausnahme ausgelöst, die gegenwärtig vom catch-Block behandelt wird.

Da eine throw-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer throw-Anweisung nie erreichbar.

Beim Auslösen einer Ausnahme wird die Steuerung an die erste catch-Klausel in einer einschließenden try-Anweisung übergeben, die die Ausnahme behandeln kann. Der ausgehend vom Auslösen der Ausnahme bis hin zum Übertragen der Steuerung an einen geeigneten Ausnahmehandler ausgeführte Prozess wird als Verbreitung von Ausnahmen bezeichnet. Bei der Verbreitung einer Ausnahme werden die folgenden Schritte so oft ausgewertet, bis eine catch-Klausel gefunden wird, die mit der Ausnahme übereinstimmt. In dieser Beschreibung ist der Meldepunkt anfänglich die Position, an der die Ausnahme ausgelöst wird.

* In dem aktuellen Funktionsmember werden alle try-Anweisungen überprüft, die den Meldepunkt einschließen. Für jede Anweisung S werden, angefangen bei der innersten try-Anweisung bis zur äußersten try-Anweisung, nacheinander folgende Schritte ausgewertet:
* Wenn der try-Block von S den Meldepunkt einschließt und S mindestens eine catch-Klausel enthält, wird in den catch-Klauseln in der Reihenfolge ihres Auftretens nach einem geeigneten Handler für die Ausnahme gesucht. Die erste catch-Klausel, die den Ausnahmetyp oder einen Basistyp des Ausnahmetyps angibt, wird als Entsprechung betrachtet. Eine allgemeine catch-Klausel (§8.10) wird als Entsprechung für alle Ausnahmetypen betrachtet. Wenn eine entsprechende catch -Klausel gefunden wird, wird die Verbreitung der Ausnahme durch Übergabe der Steuerung an den Block dieser catch-Klausel abgeschlossen.
* Wenn jedoch der try-Block oder ein catch-Block von S den Meldepunkt einschließt und S einen finally-Block enthält, wird die Steuerung an den finally-Block übergeben. Wenn der finally-Block eine weitere Ausnahme auslöst, wird die Verarbeitung der aktuellen Ausnahme beendet. Wenn die Steuerung hingegen den Endpunkt des finally-Blocks erreicht, wird mit der Verarbeitung der aktuellen Ausnahme fortgefahren.
* Wenn bei dem aktuellen Funktionsaufruf kein Ausnahmehandler gefunden wurde, wird der Funktionsaufruf beendet, und Folgendes geschieht:
* Wenn die aktuelle Funktion nicht "async" ist, werden die oben genannten Schritte für den Aufrufer der Funktion mit einem Meldepunkt wiederholt, der mit der Anweisung übereinstimmt, aus der der Funktionsmember aufgerufen wurde.
* Wenn die aktuelle Funktion "async" ist und eine Aufgabe zurückgibt, wird die Ausnahme in der Rückgabeaufgabe aufgezeichnet, die wie in §10.14.1 erläutert in einen fehlerhaften oder abgebrochenen Zustand versetzt wird.
* Wenn die aktuelle Funktion "async" ist und "void" zurückgibt, wird der Synchronisierungskontext des aktuellen Threads wie in §10.14.2 erläutert benachrichtigt.
* Wenn die Ausnahmeverarbeitung alle Funktionsmemberaufrufe im aktuellen Thread beendet und somit darauf hinweist, dass der Thread keinen Handler für die Ausnahme enthält, dann wird der Thread selbst beendet. Die Auswirkung dieses Abbruchs hängt von der Implementierung ab.

## Die try-Anweisung

Die try-Anweisung stellt einen Mechanismus zum Auffangen von Ausnahmen zur Verfügung, die während der Ausführung eines Blocks auftreten. Außerdem bietet die try-Anweisung die Möglichkeit, einen Codeblock anzugeben, der immer ausgeführt wird, wenn die Steuerung die try-Anweisung verlässt.

try-statement:  
try block catch-clauses  
try block finally-clause  
try block catch-clauses finally-clause

catch-clauses:  
specific-catch-clauses general-catch-clauseopt  
specific-catch-clausesopt general-catch-clause

specific-catch-clauses:  
specific-catch-clause  
specific-catch-clauses specific-catch-clause

specific-catch-clause:  
catch ( class-type identifieropt ) block

general-catch-clause:  
catch block

finally-clause:  
finally block

Es gibt drei mögliche Formen von try-Anweisungen:

* Ein try-Block gefolgt von einem oder mehreren catch-Blöcken.
* Ein try-Block gefolgt von einem finally-Block.
* Ein try-Block gefolgt von einem oder mehreren catch-Blöcken, denen wiederum ein finally-Block folgt.

Wenn eine catch-Klausel einen class-type angibt, muss der Typ System.Exception sein, bzw. ein Typ, der von System.Exception abgeleitet ist, oder ein Typparametertyp, dessen effektive Basisklasse System.Exception (oder eine Unterklasse dieser) ist.

Wenn eine catch-Klausel sowohl einen class-type als auch einen identifier angibt, wird eine Ausnahmevariable des angegebenen Namens und Typs deklariert. Die Ausnahmevariable entspricht einer schreibgeschützten lokalen Variablen mit einem Gültigkeitsbereich, der sich über den catch-Block erstreckt. Während der Ausführung des catch-Blocks stellt die Ausnahmevariable die gegenwärtig behandelte Ausnahme dar. Für die endgültige Zuweisungsüberprüfung wird die Ausnahmevariable in ihrem gesamten Gültigkeitsbereich als endgültig zugewiesen betrachtet.

Auf das Ausnahmeobjekt in dem catch-Block kann nur zugegriffen werden, wenn eine catch-Klausel einen Ausnahmevariablennamen enthält.

Eine catch-Klausel, die weder einen Ausnahmetyp noch einen Ausnahmevariablennamen angibt, wird als allgemeine catch-Klausel bezeichnet. Eine try-Anweisung kann nur eine allgemeine catch-Klausel enthalten, und es muss sich dabei ggf. um die letzte catch-Klausel handeln.

Einige Programmiersprachen unterstützen möglicherweise Ausnahmen, die nicht als von System.Exception abgeleitete Objekte dargestellt werden können. Derartige Ausnahmen könnten jedoch niemals mittels C#-Code generiert werden. Zum Auffangen solcher Ausnahmen kann eine allgemeine catch-Klausel verwendet werden. Folglich unterscheidet sich eine allgemeine catch-Klausel semantisch von einer Klausel, mit der der Typ System.Exception angegeben wird, da die catch-Klausel auch Ausnahmen anderer Sprachen auffangen kann.

Um einen Handler für eine Ausnahme zu suchen, werden catch-Klauseln in lexikalischer Reihenfolge überprüft. Wenn eine catch-Klausel einen Typ angibt, der dem in einer vorherigen catch-Klausel für dieselbe try-Anweisung angegebenen Typ entspricht oder von diesem abgeleitet wurde, tritt ein Kompilierungsfehler auf. Ohne diese Einschränkung wäre es möglich, unerreichbare catch-Klauseln zu schreiben.

Innerhalb eines catch-Blocks kann eine throw-Anweisung (§8.9.5) ohne Ausdruck zum erneuten Auslösen der Ausnahme verwendet werden, die vom catch-Block aufgefangen wurde. Zuweisungen zu einer Ausnahmenvariablen bewirken keine Änderung der erneut ausgelösten Ausnahme.

In dem Beispiel

using System;

class Test  
{  
 static void F() {  
 try {  
 G();  
 }  
 catch (Exception e) {  
 Console.WriteLine("Exception in F: " + e.Message);  
 e = new Exception("F");  
 throw; // re-throw  
 }  
 }

static void G() {  
 throw new Exception("G");  
 }

static void Main() {  
 try {  
 F();  
 }  
 catch (Exception e) {  
 Console.WriteLine("Exception in Main: " + e.Message);  
 }  
 }  
}

fängt die Methode F eine Ausnahme auf, schreibt einige Diagnosedaten in die Konsole, ändert die Ausnahmevariable und löst die Ausnahme erneut aus. Da es sich bei der erneut ausgelösten Ausnahme um die ursprüngliche Ausnahme handelt, lautet die Ausgabe wie folgt:

Exception in F: G  
Exception in Main: G

Wenn der erste catch-Block anstelle der aktuellen Ausnahme e ausgelöst hätte, würde die Ausgabe wie folgt lauten:

Exception in F: G  
Exception in Main: F

break-, continue- oder goto-Anweisungen können die Steuerung nicht aus einem finally-Block heraus übergeben, da ansonsten ein Kompilierungsfehler verursacht würde. Wenn eine break-, continue- oder goto-Anweisung innerhalb eines finally-Blocks auftritt, muss sich das Ziel der Anweisung innerhalb desselben finally-Blocks befinden. Andernfalls tritt während der Kompilierung ein Fehler auf.

In einem finally-Block darf keine return-Anweisung auftreten, andernfalls wird ein Kompilierungsfehler erzeugt.

Eine try-Anweisung wird folgendermaßen ausgeführt:

* Die Steuerung wird an den try-Block übergeben.
* Wenn die Steuerung den Endpunkt des try-Blocks erreicht:
* Wenn die try-Anweisung über einen finally-Block verfügt, wird der finally-Block ausgeführt.
* Die Steuerung wird an den Endpunkt der try-Anweisung übergeben.
* Wenn eine Ausnahme während der Ausführung des try-Blocks an die try-Anweisung weitergegeben wird:
* Die catch-Klauseln werden ggf. in der Reihenfolge ihres Auftretens überprüft, um einen geeigneten Handler für die Ausnahme zu finden. Die erste catch-Klausel, die den Ausnahmetyp oder einen Basistyp des Ausnahmetyps angibt, wird als Entsprechung betrachtet. Eine allgemeine catch-Klausel wird als Entsprechung für alle Ausnahmetypen betrachtet. Wenn eine entsprechende catch-Klausel gefunden wird:
* Wenn die entsprechende catch-Klausel eine Ausnahmevariable deklariert, wird das Ausnahmeobjekt der Ausnahmevariablen zugewiesen.
* Die Steuerung wird an den übereinstimmenden catch-Block übergeben.
* Wenn die Steuerung den Endpunkt des catch-Blocks erreicht:
* Wenn die try-Anweisung über einen finally-Block verfügt, wird der finally-Block ausgeführt.
* Die Steuerung wird an den Endpunkt der try-Anweisung übergeben.
* Wenn eine Ausnahme während der Ausführung des catch-Blocks an die try-Anweisung weitergegeben wird:
* Wenn die try-Anweisung über einen finally-Block verfügt, wird der finally-Block ausgeführt.
* Die Ausnahme wird an die nächste einschließende try-Anweisung weitergegeben.
* Wenn die try-Anweisung nicht über eine catch-Klausel verfügt oder keine catch-Klausel der Ausnahme entspricht:
* Wenn die try-Anweisung über einen finally-Block verfügt, wird der finally-Block ausgeführt.
* Die Ausnahme wird an die nächste einschließende try-Anweisung weitergegeben.

Die Anweisungen eines finally-Blocks werden immer ausgeführt, wenn die Steuerung eine try-Anweisung verlässt. Dies trifft zu, wenn die Steuerung aufgrund einer normalen Ausführung, aufgrund der Ausführung einer break-Anweisung, einer continue-Anweisung, einer goto-Anweisung oder einer return-Anweisung oder aufgrund der Weitergabe einer Ausnahme aus der try-Anweisung heraus übergeben wird.

Wenn eine Ausnahme während der Ausführung eines finally-Blocks ausgelöst und nicht innerhalb desselben finally-Blocks abgefangen wird, wird die Ausnahme an die nächste einschließende try-Anweisung weitergegeben. Wenn zu demselben Zeitpunkt die Weitergabe einer weiteren Ausnahme erfolgt, geht diese Ausnahme verloren. Weitere Informationen zum Weitergeben von Ausnahmen finden Sie in der Beschreibung der throw-Anweisung (§8.9.5).

Der try-Block einer try-Anweisung ist erreichbar, wenn die try-Anweisung erreichbar ist.

Ein catch-Block einer try-Anweisung ist erreichbar, wenn die try-Anweisung erreichbar ist.

Der finally-Block einer try-Anweisung ist erreichbar, wenn die try-Anweisung erreichbar ist.

Der Endpunkt einer try-Anweisung ist erreichbar, wenn beide der folgenden Bedingungen erfüllt werden:

* Der Endpunkt des try-Blocks oder der Endpunkt mindestens eines catch-Blocks ist erreichbar.
* Der Endpunkt des finally-Blocks ist erreichbar, wenn ein finally-Block vorhanden ist.

## Die checked-Anweisung und die unchecked-Anweisung

Die checked-Anweisung und die unchecked-Anweisung werden verwendet, um den Kontext der Überlaufprüfung für arithmetische Ganzzahloperationen und Konvertierungen zu steuern.

checked-statement:  
checked block

unchecked-statement:  
unchecked block

Während die checked-Anweisung bewirkt, dass alle Ausdrücke im block in einem geprüften Kontext ausgewertet werden, führt die unchecked-Anweisung dazu, dass alle Ausdrücke im block in einem nicht geprüften Kontext ausgewertet werden.

Die checked-Anweisung und die unchecked-Anweisung stimmen mit dem checked-Operator und dem unchecked-Operator überein (§7.6.12), außer dass sie in Blöcken und nicht in Ausdrücken angewendet werden.

## Die lock-Anweisung

Die lock-Anweisung ermittelt die Sperre für gegenseitigen Ausschluss eines bestimmten Objekts, führt eine Anweisung aus und hebt anschließend die Sperre auf.

lock-statement:  
lock ( expression ) embedded-statement

Der Ausdruck einer lock-Anweisung muss auf einen Wert von einem Typ deuten, von dem bekannt ist, dass er ein reference-type ist. Da eine implizite Boxingkonvertierung (§6.1.7) für den Ausdruck einer lock-Anweisung nie ausgeführt wird, darf der Ausdruck nicht auf den Wert eines value-type deuten, da andernfalls ein Kompilierungsfehler ausgelöst würde.

Eine lock-Anweisung der Form

lock (x) ...

(wobei x ein Ausdruck eines reference-type ist) stimmt überein mit

bool \_\_lockWasTaken = false;  
try {  
 System.Threading.Monitor.Enter(x, ref \_\_lockWasTaken);  
 ...  
}  
finally {  
 if (\_\_lockWasTaken) System.Threading.Monitor.Exit(x);  
}

mit der Ausnahme, dass x nur einmal ausgewertet wird.

Während eine Sperre für gegenseitigen Ausschluss besteht, kann der im selben Ausführungsthread ausgeführte Code ebenfalls gesperrt werden oder die Sperre aufheben. In anderen Threads ausgeführter Code kann so lange nicht mit dieser Sperre belegt werden, bis sie wieder freigegeben ist.

Ein Sperren von System.Type-Objekten zum Synchronisieren des Zugriffs auf statische Daten wird nicht empfohlen. Anderer Code können denselben Typ sperren, was zu einem Deadlock führt. Es ist daher empfehlenswert, stattdessen den Zugriff auf statische Daten durch Sperren eines privaten statischen Objekts zu synchronisieren. Beispiel:

class Cache  
{  
 private static readonly object synchronizationObject = new object();

public static void Add(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }

public static void Remove(object x) {  
 lock (Cache.synchronizationObject) {  
 ...  
 }  
 }  
}

## Die using-Anweisung

Die using-Anweisung ermittelt eine oder mehrere Ressourcen, führt eine Anweisung aus und verwirft die Ressource anschließend.

using-statement:  
using ( resource-acquisition ) embedded-statement

resource-acquisition:  
local-variable-declaration  
expression

Eine Ressource ist eine Klasse oder eine Struktur zur Implementierung von System.IDisposable, die eine einzige parameterlose Methode mit der Bezeichnung Dispose enthält. Code, der eine Ressource verwendet, kann durch Aufrufen von Dispose anzeigen, dass die Ressource nicht mehr benötigt wird. Wenn Dispose nicht aufgerufen wird, wird die Ressource aufgrund von Garbage Collection schließlich automatisch verworfen.

Wenn resource-acquisition die Form local-variable-declaration hat, dann muss der Typ der local-variable-declaration entweder dynamic oder ein Typ sein, der implizit in System.IDisposable konvertiert werden kann. Wenn resource-acquisition die Form expression hat, dann muss dieser Ausdruck implizit in System.IDisposable konvertiert werden können.

Lokale Variablen, die in einer resource-acquisition deklariert werden, sind schreibgeschützt und müssen einen Initialisierer enthalten. Wenn die eingebettete Anweisung versucht, eine Änderung der lokalen Variablen (durch Zuweisung bzw. die Operatoren ++ und ‑‑, deren Adresse abzurufen oder die Weitergabe als ref-Parameter oder als out-Parameter vorzunehmen, tritt während der Kompilierung ein Fehler auf.

Eine using-Anweisung wird in drei Teile übersetzt: Akquirieren, Verwenden und Verwerfen. Die Verwendung der Ressource wird implizit in eine try-Anweisung eingeschlossen, die eine finally-Klausel enthält. Diese finally-Klausel verwirft die Ressource. Wenn eine null-Ressource erforderlich ist, dann wird Dispose nicht aufgerufen, und es wird keine Ausnahme ausgelöst. Wenn die Ressource vom Typ dynamic ist, wird sie während des Akquirierens über eine implizite dynamische Konvertierung (§6.1.8) dynamisch in IDisposable konvertiert, damit vor der Verwendung und dem Verwerfen sichergestellt wird, dass die Konvertierung erfolgreich ist.

Eine using-Anweisung der Form

using (ResourceType resource = expression) statement

entspricht einer der drei folgenden Erweiterungen: Wenn ResourceType ein Werttyp ist, der keine NULL-Werte zulässt, ist die Erweiterung

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 ((IDisposable)resource).Dispose();  
 }  
}

Wenn andernfalls ResourceType ein Werttyp, der NULL-Werte zulässt, oder ein anderer Verweistyp als dynamic ist, lautet die Erweiterung

{  
 ResourceType resource = expression;  
 try {  
 statement;  
 }  
 finally {  
 if (resource != null) ((IDisposable)resource).Dispose();  
 }  
}

Wenn ResourceType andernfalls dynamic ist, lautet die Erweiterung

{  
 ResourceType resource = expression;  
 IDisposable d = (IDisposable)resource;  
 try {  
 statement;  
 }  
 finally {  
 if (d != null) d.Dispose();  
 }  
}

In jeder Erweiterung ist die resource-Variable in der eingebetteten Anweisung schreibgeschützt, und auf die d-Variable kann in der eingebetteten Anweisung nicht zugegriffen werden, und sie wird auch nicht für die Anweisung angezeigt.

Bei einer Implementierung kann eine using-Anweisung auch auf andere Weise implementiert werden, z. B. aus Leistungsgründen, sofern das Verhalten mit der oben stehenden Erweiterung übereinstimmt.

Eine using-Anweisung der Form

using (expression) statement

verfügt über dieselben drei möglichen Erweiterungen. In diesem Fall ist ResourceType implizit der Kompilierungstyp der expression, sofern vorhanden. Andernfalls wird die IDisposable-Schnittstelle als ResourceType verwendet. Auf die resource-Variable kann in der eingebetteten Anweisung nicht zugegriffen werden, und sie wird auch nicht für die Anweisung angezeigt.

Wenn eine resource-acquisition das Format einer local-variable-declaration annimmt, können mehrere Ressourcen eines bestimmten Typs abgerufen werden. Eine using-Anweisung der Form

using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement

entspricht genau einer Abfolge geschachtelter using-Anweisungen:

using (ResourceType r1 = e1)  
 using (ResourceType r2 = e2)  
 ...  
 using (ResourceType rN = eN)  
 statement

Im folgenden Beispiel wird eine Datei mit dem Namen log.txt erstellt, in die zwei Textzeilen geschrieben werden. Anschließend wird im Beispiel genau diese Datei zum Lesen geöffnet, und die darin enthaltenen Textzeilen werden in die Konsole kopiert.

using System;  
using System.IO;

class Test  
{  
 static void Main() {  
 using (TextWriter w = File.CreateText("log.txt")) {  
 w.WriteLine("This is line one");  
 w.WriteLine("This is line two");  
 }

using (TextReader r = File.OpenText("log.txt")) {  
 string s;  
 while ((s = r.ReadLine()) != null) {  
 Console.WriteLine(s);  
 }

}  
 }  
}

Da die IDisposable-Schnittstelle durch die Klassen TextWriter und TextReader implementiert wird, kann im Beispiel mithilfe von using-Anweisungen sichergestellt werden, dass die zugrunde liegende Datei nach den Schreib- oder Leseoperationen ordnungsgemäß geschlossen wird.

## Die yield-Anweisung

Die yield-Anweisung wird in einem Iteratorblock (§8.2) zur Ausgabe eines Werts an das Enumeratorobjekt (§10.14.4) bzw. das aufzählbare Objekt (§10.14.5) eines Iterators oder zum Angeben des Endes der Iteration verwendet.

yield-statement:  
yield return expression ;  
yield break ;

yield ist kein reserviertes Wort, und verfügt auch nur dann über eine spezielle Bedeutung, wenn es direkt vor dem Schlüsselwort return oder break steht. In anderen Kontexten kann yield als Bezeichner verwendet werden.

Es gibt einige Einschränkungen für die Stellen, an denen eine yield-Anweisung stehen kann, die im Folgenden beschrieben werden.

* Wenn eine yield-Anweisung (beider Formen) außerhalb von method-body, operator-body oder accessor-body vorkommt, tritt ein Kompilierungsfehler auf.
* Wenn eine yield-Anweisung (beider Formen) in einer anonymen Funktion steht, tritt ein Kompilierungsfehler auf.
* Wenn eine yield-Anweisung (beider Formen) in der finally-Klausel einer try-Anweisung steht, tritt ein Kompilierungsfehler auf.
* Wenn eine yield return-Anweisung in einer try-Anweisung mit beliebigen catch-Klauseln steht, tritt ein Kompilierungsfehler auf.

Im folgenden Beispiel werden einige gültige und ungültige Verwendungen von yield-Anweisungen veranschaulicht.

delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {  
 try {  
 yield return 1; // Ok  
 yield break; // Ok  
 }  
 finally {  
 yield return 2; // Error, yield in finally  
 yield break; // Error, yield in finally  
 }

try {  
 yield return 3; // Error, yield return in try...catch  
 yield break; // Ok  
 }  
 catch {  
 yield return 4; // Error, yield return in try...catch  
 yield break; // Ok  
 }

D d = delegate {   
 yield return 5; // Error, yield in an anonymous function  
 };   
}

int MyMethod() {  
 yield return 1; // Error, wrong return type for an iterator block  
}

Eine implizite Konvertierung (§6.1) muss für den Typ des Ausdrucks in der yield return-Anweisung in den Ausgabetyp (§10.14.3) des Iterators vorhanden sein.

Eine yield return-Anweisung wird folgendermaßen ausgeführt:

* Der in der Anweisung angegebene Ausdruck wird ausgewertet, implizit in den Ausgabetyp umgewandelt und der Current-Eigenschaft des Enumeratorobjekts zugewiesen.
* Die Ausführung des Iteratorblocks wird unterbrochen. Wenn sich die yield return-Anweisung innerhalb eines oder mehrerer try-Blöcke befindet, werden die zugehörigen finally-Blöcke zu diesem Zeitpunkt nicht ausgeführt.
* Die MoveNext-Methode des Enumeratorobjekts gibt true an seinen Aufrufer zurück, um anzugeben, dass das Enumeratorobjekt erfolgreich auf das nächste Element erhöht wurde.

Der nächste Aufruf der MoveNext-Methode des Enumeratorobjekts setzt die Ausführung des Iteratorblocks an der Stelle der letzten Unterbrechung fort.

Eine yield break-Anweisung wird folgendermaßen ausgeführt:

* Wenn die yield break-Anweisung in einen oder mehrere try-Blöcke mit verknüpften finally-Blöcken eingeschlossen ist, wird die Steuerung zunächst an den finally-Block der innersten try-Anweisung übergeben. Wenn die Steuerung den Endpunkt eines finally-Blocks erreicht, wird sie an den finally-Block der nächsten einschließenden try-Anweisung übergeben. Dieser Prozess wird so lange wiederholt, bis von den finally-Blöcken alle einschließenden try-Anweisungen ausgeführt wurden.
* Die Steuerung wird wieder an den Aufrufer des Iteratorblocks übertragen. Dies ist entweder die MoveNext-Methode oder die Dispose-Methode des Enumeratorobjekts.

Da eine yield break-Anweisung die Steuerung unabhängig von den Bedingungen übergibt, ist der Endpunkt einer yield break-Anweisung nie erreichbar.

# Namespaces

C#-Programme werden mithilfe von Namespaces organisiert. Namespaces werden sowohl als „internes“ Organisationssystem für ein Programm als auch als „externes“ Organisationssystem verwendet. Das externe System stellt eine Möglichkeit der Präsentation von Programmelementen dar, die für andere Programme offen gelegt werden.

Die Verwendung von Namespaces wird durch die angegebenen using-Direktiven (§9.4) erleichtert.

## Kompilationseinheiten

Eine compilation-unit definiert die allgemeine Struktur einer Quelldatei. Eine Kompilationseinheit besteht aus keiner oder mehreren using-directives, gefolgt von keinem oder mehreren global-attributes, an die sich wiederum keine oder mehrere namespace-member-declarations anschließen.

compilation-unit:  
extern-alias-directivesopt using-directivesopt global-attributesopt  
 namespace-member-declarationsopt

Ein C#-Programm besteht aus einer oder mehreren Kompilationseinheiten, die jeweils in einer separaten Quelldatei enthalten sind. Wenn ein C#-Programm kompiliert wird, werden alle Kompilationseinheiten zusammen verarbeitet. Daher können Kompilationseinheiten voneinander abhängen (möglichst in zirkulärer Form).

Die using-directives einer Kompilationseinheit wirken sich zwar auf die global-attributes und namespace-member-declarations dieser Kompilationseinheit aus, haben jedoch keinen Einfluss auf andere Kompilationseinheiten.

Die global-attributes (§17) einer Kompilationseinheit ermöglichen es, Attribute für die Zielassembly und das Zielmodul festzulegen. Assemblies und Module werden als physische Container für Typen verwendet. Eine Assembly kann aus mehreren physikalisch getrennten Modulen bestehen.

Die namespace-member-declarations jeder Kompilationseinheit eines Programms tragen Member zu einem einzelnen Deklarationsabschnitt bei, der als globaler Namespace bezeichnet wird. Beispiel:

Datei A.cs:

class A {}

Datei B.cs:

class B {}

Die beiden Kompilationseinheiten tragen zu demselben einzelnen globalen Namespace bei. In diesem Fall werden dazu zwei Klassen mit den vollqualifizierten Namen A und B deklariert. Da die beiden Kompilationseinheiten zu demselben Deklarationsabschnitt beitragen, würde ein Fehler auftreten, wenn jede von ihnen eine Deklaration eines Members mit demselben Namen enthalten würde.

## Namespacedeklarationen

Eine namespace-declaration besteht aus dem Schlüsselwort namespace, gefolgt von einem Namespacenamen und einem Namespacetext, an die sich wiederum optional ein Semikolon anschließt.

namespace-declaration:  
namespace qualified-identifier namespace-body ;opt

qualified-identifier:  
identifier  
qualified-identifier . identifier

namespace-body:  
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

Eine namespace-declaration kann als Deklaration der höchsten Ebene in einer compilation-unit oder als Memberdeklaration innerhalb einer anderen namespace-declaration auftreten. Wenn eine namespace-declaration als Deklaration der höchsten Ebene in einer compilation-unit auftritt, wird der Namespace ein Member des globalen Namespaces. Wenn eine namespace-declaration innerhalb einer anderen namespace-declaration auftritt, wird der innere Namespace ein Member des äußeren Namespaces. In beiden Fällen muss der Name des Namespaces innerhalb des enthaltenden Namespaces eindeutig sein.

Da Namespaces implizit public sind, darf die Deklaration eines Namespaces keinen Zugriffsmodifizierer enthalten.

Innerhalb eines namespace-bodys importieren die optionalen using-directives die Namen anderer Namespaces und Typen, sodass auf diese direkt und nicht über gekennzeichnete Namen verwiesen werden kann. Die optionalen namespace-member-declarations tragen Member zum Deklarationsabschnitt des Namespaces bei. Alle using-directives müssen vor den Memberdeklarationen auftreten.

Beim qualified-identifier einer namespace-declaration kann es sich um einen einzelnen bzw. mehrere, durch "."-Token voneinander getrennte Bezeichner handeln. Wenn die zweite Form verwendet wird, kann durch ein Programm ein geschachtelter Namespace ohne die lexikalische Schachtelung von mehreren Namespacedeklarationen definiert werden. Beispiel:

namespace N1.N2  
{  
 class A {}

class B {}  
}

ist mit folgender Anweisung semantisch äquivalent

namespace N1  
{  
 namespace N2  
 {  
 class A {}

class B {}  
 }  
}

Namespaces haben ein offenes Ende. Zwei Namespacedeklarationen mit demselben vollqualifizierten Namen tragen zu demselben Deklarationsabschnitt bei (§3.3). In dem Beispiel

namespace N1.N2  
{  
 class A {}  
}

namespace N1.N2  
{  
 class B {}  
}

tragen die beiden oben stehenden Namespacedeklarationen zu demselben Deklarationsabschnitt bei. In diesem Fall werden dazu zwei Klassen mit denselben vollqualifizierten Namen N1.N2.A und N1.N2.B deklariert. Da die beiden Deklarationen zu demselben Deklarationsabschnitt beitragen, würde ein Fehler auftreten, wenn jede von ihnen eine Deklaration eines Members mit demselben Namen enthalten würde.

## extern-Aliase

Eine extern-alias-directive führt einen Bezeichner ein, der als Alias für einen Namespace fungiert. Die Spezifikation für den mit einem Alias versehenen Namespace ist für den Quellcode des Programms extern und gilt auch für in diesem Namespace geschachtelte Namespaces.

extern-alias-directives:  
extern-alias-directive  
extern-alias-directives extern-alias-directive

extern-alias-directive:  
extern alias identifier ;

Der Gültigkeitsbereich einer extern-alias-directive erstreckt sich über die using-directives, global-attributes und namespace-member-declarations der unmittelbar enthaltenden Kompilationseinheit bzw. des unmittelbar enthaltenden Namespacetexts.

Innerhalb von Kompilierungseinheiten oder Namespacetexten, die eine extern-alias-directive enthalten, kann der über die extern-alias-directive eingeführte Bezeichner zum Verweisen auf den mit einem Alias versehenen Namespace verwendet werden. Wenn der identifier das Wort global ist, tritt ein Fehler auf.

Eine extern-alias-directive stellt zwar einen Alias innerhalb einer bestimmten Kompilationseinheit oder innerhalb eines bestimmten Namespacetexts zur Verfügung, trägt aber zum zugrunde liegenden Deklarationsabschnitt keine neuen Member bei. Mit anderen Worten: Eine extern-alias-directive ist nicht transitiv. Stattdessen wirkt sie sich nur auf die Kompilationseinheit bzw. den Namespacetext aus, in der bzw. dem sie auftritt.

Das folgende Programm deklariert und verwendet die zwei extern-Aliase X und Y, die jeweils den Stamm einer separaten Namespacehierarchie darstellen:

extern alias X;  
extern alias Y;

class Test  
{  
 X::N.A a;  
 X::N.B b1;  
 Y::N.B b2;  
 Y::N.C c;  
}

Im Programm wird das Vorhandensein der extern-Aliase X und Y deklariert. Die tatsächliche Definition der Aliase erfolgt jedoch außerhalb des Programms. Auf die gleichnamigen N.B-Klassen kann nun mit X.N.B und Y.N.B oder unter Verwendung des Namespacealias-Qualifizierers mit X::N.B und Y::N.B verwiesen werden. Wenn ein Programm einen extern-Alias deklariert, für den keine externe Definition angegeben ist, tritt ein Fehler auf.

## Using-Direktiven

Using-Direktiven erleichtern die Verwendung von Namespaces und Typen, die in anderen Namespaces definiert sind. Using-Direktiven wirken sich auf den Auflösungsprozess von namespace-or-type-names (§3.8) sowie simple-names (§7.6.2) aus. Im Gegensatz zu Deklarationen tragen using-Direktiven jedoch keine neuen Member zu den zugrunde liegenden Deklarationsabschnitten der Kompilationseinheiten oder Namespaces bei, innerhalb derer sie verwendet werden.

using-directives:  
using-directive  
using-directives using-directive

using-directive:  
using-alias-directive  
using-namespace-directive

Eine using-alias-directive (§9.4.1) führt einen Alias für einen Namespace oder Typ ein.

Eine using-namespace-directive (§9.4.2) importiert die Typmember eines Namespaces.

Der Gültigkeitsbereich einer using-directive erstreckt sich über die namespace-member-declarations der unmittelbar enthaltenden Kompilationseinheit oder des unmittelbar enthaltenden Namespacetexts. Der Gültigkeitsbereich einer using-directive enthält ausdrücklich nicht deren gleichrangige using-directives. Daher beeinflussen sich gleichrangige using-directives nicht gegenseitig, und es spielt keine Rolle, in welcher Reihenfolge sie geschrieben werden.

### Using-Alias-Direktiven

Eine using-alias-directive führt einen Bezeichner ein, der als Alias für einen Namespace oder Typ innerhalb der unmittelbar einschließenden Kompilationseinheit oder des unmittelbar einschließenden Namespacetexts verwendet werden kann.

using-alias-directive:  
using identifier = namespace-or-type-name ;

Innerhalb von Memberdeklarationen in einer Kompilationseinheit oder einem Namespacetext, der eine using-alias-directive enthält, kann der über die using-alias-directive eingeführte Bezeichner zum Verweisen auf den angegebenen Namespace oder Typ verwendet werden. Beispiel:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A;

class B: A {}  
}

Im oben genannten Beispiel ist A zwischen den Memberdeklarationen im N3-Namespace ein Alias für N1.N2.A. Somit wird die Klasse N3.B von der Klasse N1.N2.A abgeleitet. Dasselbe Ergebnis wird erzielt, wenn ein Alias R für N1.N2 erstellt und auf R.A verwiesen wird:

namespace N3  
{  
 using R = N1.N2;

class B: R.A {}  
}

Der identifier einer using-alias-directive muss innerhalb des Deklarationsabschnitts der Kompilationseinheit oder des Namespaces, der die using-alias-directive unmittelbar enthält, eindeutig sein. Beispiel:

namespace N3  
{  
 class A {}  
}

namespace N3  
{  
 using A = N1.N2.A; // Error, A already exists  
}

Da N3 im oben genannten Beispiel bereits einen Member A enthält, gilt die Verwendung dieses Bezeichners durch eine using-alias-directive als Kompilierungsfehler. Ebenso entsteht ein Kompilierungsfehler, wenn zwei oder mehrere using-alias-directives in derselben Kompilationseinheit oder in demselben Namespacetext Aliase mit demselben Namen deklarieren.

Eine using-alias-directive zwar einen Alias innerhalb einer bestimmten Kompilationseinheit oder innerhalb eines bestimmten Namespacetexts zur Verfügung, trägt aber zum zugrunde liegenden Deklarationsabschnitt keine neuen Member bei. Das bedeutet, dass eine using-alias-directive nicht transitiv ist und nur die Kompilationseinheit bzw. den Namespacetext betrifft, in der bzw. dem sie auftritt. In dem Beispiel

namespace N3  
{  
 using R = N1.N2;  
}

namespace N3  
{  
 class B: R.A {} // Error, R unknown  
}

erstreckt sich der Gültigkeitsbereich der using-alias-directive, die R einführt, nur auf Memberdeklarationen in dem Namespacetext, in dem dieser enthalten ist. Folglich ist R in der zweiten Namespacedeklaration unbekannt. Wenn die using-alias-directive jedoch in der enthaltenden Kompilationseinheit abgelegt wird, steht der Alias in beiden Namespacedeklarationen zur Verfügung:

using R = N1.N2;

namespace N3  
{  
 class B: R.A {}  
}

namespace N3  
{  
 class C: R.A {}  
}

Ähnlich wie reguläre Member sind auch Namen, die von einer using-alias-directive eingeführt wurden, von Membern mit ähnlichen Namen in geschachtelten Gültigkeitsbereichen ausgeblendet. In dem Beispiel

using R = N1.N2;

namespace N3  
{  
 class R {}

class B: R.A {} // Error, R has no member A  
}

verursacht der Verweis auf R.A in der Deklaration von B einen Kompilierungsfehler, da R nicht auf N1.N2, sondern auf N3.R verweist.

Die Reihenfolge, in der using-alias-directives geschrieben werden, hat keine Bedeutung. Die Auflösung des namespace-or-type-name, auf den von einer using-alias-directive verwiesen wird, wird weder von der using-alias-directive selbst noch von einer anderen using-directive in der unmittelbar enthaltenden Kompilationseinheit oder dem unmittelbar enthaltenden Namespacetext beeinflusst. Das bedeutet, dass der namespace-or-type-name einer using-alias-directive so aufgelöst wird, als ob die unmittelbar enthaltende Kompilationseinheit bzw. der unmittelbar enthaltende Namespacetext über keine using-directives verfügen würde. Eine using-alias-directive kann jedoch durch extern-alias-directives in der unmittelbar enthaltenden Kompilationseinheit oder dem Namespacetext beeinflusst werden. In dem Beispiel

namespace N1.N2 {}

namespace N3  
{  
 extern alias E;

using R1 = E.N; // OK

using R2 = N1; // OK

using R3 = N1.N2; // OK

using R4 = R2.N2; // Error, R2 unknown  
}

resultiert die letzte using-alias-directive in einem Kompilierungsfehler, da sie nicht von der ersten using-alias-directive beeinflusst wird. Die erste using-alias-directive resultiert nicht in einem Fehler, da der Gültigkeitsbereich des extern-Alias E die using-alias-directive einschließt.

Eine using-alias-directive kann einen Alias für jeden Namespace oder Typ erstellen. Dazu zählen auch der Namespace, in dem sie auftritt, sowie alle Namespaces und Typen, die in diesem Namespace geschachtelt sind.

Das Zugreifen auf einen Namespace oder Typ über einen Alias führt zu genau demselben Ergebnis wie das Zugreifen auf den Namespace oder Typ über dessen deklarierten Namen. Wenn wie im folgendem Beispiel

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using R1 = N1;  
 using R2 = N1.N2;

class B  
 {  
 N1.N2.A a; // refers to N1.N2.A  
 R1.N2.A b; // refers to N1.N2.A  
 R2.A c; // refers to N1.N2.A  
 }  
}

die Namen N1.N2.A, R1.N2.A und R2.A äquivalent sind und alle auf die Klasse verweisen, deren vollqualifizierter Name N1.N2.A lautet.

Using-Aliase können einen geschlossen konstruierten Typ benennen, aber keine Deklaration eines ungebundenen generischen Typs ohne Angabe von Typargumenten. Beispiel:

namespace N1  
{  
 class A<T>  
 {  
 class B {}  
 }  
}

namespace N2  
{  
 using W = N1.A; // Error, cannot name unbound generic type

using X = N1.A.B; // Error, cannot name unbound generic type

using Y = N1.A<int>; // Ok, can name closed constructed type

using Z<T> = N1.A<T>; // Error, using alias cannot have type parameters  
}

### Using-Namespace-Direktiven

Eine using-namespace-directive importiert die in einem Namespace enthaltenen Typen in die unmittelbar einschließende Kompilationseinheit oder den unmittelbar einschließenden Namespacetext, indem der Bezeichner jedes Typs für die Verwendung ohne Qualifikation freigegeben wird.

using-namespace-directive:  
using namespace-name ;

Innerhalb von Memberdeklarationen in einer Kompilationseinheit oder einem Namespacetext, die/der eine using-namespace-directive enthält, kann direkt auf die Typen verwiesen werden, die im angegebenen Namespace enthalten sind. Beispiel:

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1.N2;

class B: A {}  
}

Im oben genannten Beispiel stehen die Typmember von N1.N2 innerhalb der Memberdeklarationen in dem Namespace N3 direkt zur Verfügung, sodass die Klasse N3.B von der Klasse N1.N2.A abgeleitet wird.

Eine using-namespace-directive importiert die Typen, die in einem bestimmten Namespace enthalten sind, jedoch ausdrücklich keine geschachtelten Namespaces. In dem Beispiel

namespace N1.N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

class B: N2.A {} // Error, N2 unknown  
}

importiert die using-namespace-directive die Typen, die in N1 enthalten sind. Ausdrücklich nicht importiert werden Namespaces, die in N1 geschachtelt sind. Folglich tritt bei dem Verweis auf N2.A in der Deklaration von B ein Kompilierungsfehler auf, weil sich keine Member mit der Bezeichnung N2 im Gültigkeitsbereich befinden.

Im Gegensatz zu einer using-alias-directive kann eine using-namespace-directive Typen importieren, deren Bezeichner bereits in der einschließenden Kompilationseinheit oder dem einschließenden Namespacetext definiert sind. Tatsächlich sind Namen, die von einer using-namespace-directive importiert wurden, durch Member mit ähnlichen Bezeichnungen in der einschließenden Kompilationseinheit bzw. in dem einschließenden Namespacetext ausgeblendet. Beispiel:

namespace N1.N2  
{  
 class A {}

class B {}  
}

namespace N3  
{  
 using N1.N2;

class A {}  
}

Hier bezieht sich A innerhalb der Memberdeklarationen im N3-Namespace auf N3.A und nicht auf N1.N2.A.

Wenn mindestens ein Namespace, der von using-namespace-directives in dieselbe Kompilationseinheit oder denselben Namespacetext importiert wurde, Typen mit demselben Namen enthält, werden Verweise auf diese Namen als mehrdeutig betrachtet. In dem Beispiel

namespace N1  
{  
 class A {}  
}

namespace N2  
{  
 class A {}  
}

namespace N3  
{  
 using N1;

using N2;

class B: A {} // Error, A is ambiguous  
}

enthalten sowohl N1 als auch N2 einen Member A, und da N3 beide importiert, verursacht ein Verweis auf A in N3 einen Kompilierungsfehler. In dieser Situation kann der Konflikt entweder über die Qualifikation von Verweisen auf A oder durch Einführen einer using-alias-directive aufgelöst werden, die einen bestimmten A auswählt. Beispiel:

namespace N3  
{  
 using N1;

using N2;

using A = N1.A;

class B: A {} // A means N1.A  
}

Ähnlich wie die using-alias-directive trägt auch die using-namespace-directive keine neuen Member zu dem zugrunde liegenden Deklarationsabschnitt der Kompilationseinheit oder des Namespacetexts bei, sondern beeinflusst nur die Kompilationseinheit oder den Namespacetext, in dem sie auftritt.

Der namespace-name, auf den von einer using-namespace-directive verwiesen wird, wird so aufgelöst wie ein namespace-or-type-name, auf den von einer using-alias-directive verwiesen wird. Dies bedeutet, dass using-namespace-directives innerhalb derselben Kompilationseinheit bzw. desselben Namespacetexts sich nicht gegeneinander beeinflussen und in einer beliebigen Reihenfolge geschrieben werden können.

## Namespacemember

Bei einer namespace-member-declaration handelt es sich entweder um eine namespace-declaration (§9.2) oder eine type-declaration (§9.6).

namespace-member-declarations:  
namespace-member-declaration  
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:  
namespace-declaration  
type-declaration

Kompilationseinheiten oder Namespacetexte können namespace-member-declarations enthalten, und solche Deklarationen tragen neue Member zu dem zugrunde liegenden Deklarationsabschnitt der enthaltenden Kompilationseinheit des enthaltenden Namespacetexts bei.

## Typdeklarationen

Bei einer type-declaration handelt es sich um eine class-declaration (§10.1), eine struct-declaration (§11.1), eine interface-declaration (§13.1), eine enum-declaration (§14.1) oder eine delegate-declaration (§15.1).

type-declaration:  
class-declaration  
struct-declaration  
interface-declaration  
enum-declaration  
delegate-declaration

Eine type-declaration kann als Deklaration der höchsten Ebene in einer Kompilationseinheit oder als Memberdeklaration innerhalb eines Namespaces, einer Klasse oder einer Struktur auftreten.

Wenn eine Typdeklaration für einen Typ T als Deklaration der obersten Ebene in einer Kompilierung auftritt, ist der vollqualifizierte Name des neu deklarierten Typs einfach T. Tritt eine Typdeklaration für einen Typ T innerhalb eines Namespace, einer Klasse oder einer Struktur auf, ist der vollqualifizierte Name des neu deklarierten Typs N.T, wobei N der vollqualifizierte Name des enthaltenden Namespace, der enthaltenden Klasse oder der enthaltenden Struktur ist.

Ein Typ, der innerhalb einer Klasse oder einer Struktur deklariert wird, wird als geschachtelter Typ bezeichnet (§10.3.8).

Die zulässigen Zugriffsmodifizierer und der Standardzugriff für eine Typdeklaration hängen von dem Kontext ab, in dem die Deklaration stattfindet (§3.5.1):

* Typen, die in Kompilationseinheiten oder Namespaces deklariert werden, können über public-Zugriff oder internal-Zugriff verfügen. Standardmäßig wird der internal-Zugriff verwendet.
* Typen, die in Klassen deklariert werden, können über public-, protected internal-, protected-, internal- oder private-Zugriff verfügen. Standardmäßig wird der private-Zugriff verwendet.
* Typen, die in Strukturen deklariert werden, können über public-, internal- oder private-Zugriff verfügen. Standardmäßig wird der private-Zugriff verwendet.

## Namespacealias-Qualifizierer

Mithilfe des namespace alias qualifiers :: kann sichergestellt werden, dass das Einführen neuer Typen und Member Typnamensuchen nicht beeinträchtigt. Der Namespacealias-Qualifizierer tritt immer zwischen zwei Bezeichnern auf, die linker und rechter Bezeichner genannt werden. Im Unterschied zum normalen Qualifizierer . wird der linke Bezeichner des Qualifizierers :: nur als extern-Alias oder using-Alias gesucht.

Ein qualified-alias-member wird wie folgt definiert:

qualified-alias-member:  
identifier :: identifier type-argument-listopt

Ein qualified-alias-member kann als namespace-or-type-name (§3.8) oder als linker Operand in einem member-access (§7.6.4) verwendet werden.

Ein qualified-alias-member weist eine der folgenden beiden Formen auf:

* N::I<A1, ..., AK>, wobei N und I Bezeichner darstellt und <A1, ..., AK> eine Typargumentliste ist. (K ist immer mindestens 1.)
* N::I,, wobei N und I Bezeichner darstellen. (In diesem Fall wird K als 0 (null) angesehen.)

Mit dieser Notation wird die Bedeutung eines qualified-alias-member wie folgt bestimmt:

* Wenn N der Bezeichner global ist, wird der globale Namespace nach I durchsucht:
* Wenn der globale Namespace einen Namespace mit dem Namen I enthält und K gleich 0 (null) ist, verweist der qualified-alias-member auf diesen Namespace.
* Wenn der globale Namespace hingegen einen nicht generischen Typ mit dem Namen I enthält und K gleich 0 (null) ist, verweist der qualified-alias-member auf diesen Typ.
* Wenn der globale Namespace hingegen einen Typ mit dem Namen I enthält und über K Typparameter verfügt, verweist der qualified-alias-member auf diesen Typ, der mit den angegebenen Typargumenten erstellt wurde.
* Andernfalls ist der qualified-alias-member nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Wenn jedoch mit der Namespacedeklaration (§9.2) direkt begonnen wird, die den qualified-alias-member enthält (falls vorhanden), anschließend die einzelnen einschließenden Namespacedeklarationen (falls vorhanden) aufgeführt werden und schließlich die Kompilationseinheit angegeben wird, die den qualified-alias-member enthält, werden die folgenden Schritte ausgewertet, bis eine Entität gefunden wird:
* Wenn die Namespacedeklaration oder Kompilationseinheit eine using-alias-directive enthält, die N einem Typ zuordnet, ist der qualified-alias-member nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Wenn hingegen die Namespacedeklaration oder Kompilationseinheit eine extern-alias-directive oder using-alias-directive enthält, die N einem Namespace zuordnet, dann gilt Folgendes:
* Wenn der N zugeordnete Namespace einen Namespace mit dem Namen I enthält und K gleich 0 (null) ist, verweist der qualified-alias-member auf diesen Namespace.
* Wenn der N zugeordnete Namespace hingegen einen nicht generischen Typ mit dem Namen I enthält und K gleich 0 (null) ist, verweist der qualified-alias-member auf diesen Typ.
* Wenn der N zugeordnete Namespace jedoch einen Typ mit dem Namen I enthält, der über K Typparameter verfügt, verweist der qualified-alias-member auf diesen Typ, der mit den angegebenen Typargumenten erstellt wurde.
* Andernfalls ist der qualified-alias-member nicht definiert, und es tritt ein Kompilierungsfehler auf.
* Andernfalls ist der qualified-alias-member nicht definiert, und es tritt ein Kompilierungsfehler auf.

Beachten Sie, dass durch die Verwendung des Namespacealias-Qualifizierers mit einem Alias, der auf einen Typ verweist, ein Kompilierungsfehler verursacht wird. Beachten Sie darüber hinaus, dass wenn der BezeichnerN global ist, eine Suche im globalen Namespace ausgeführt wird, selbst wenn dort ein using-Alias vorhanden ist, durch den global einem Typ oder einem Namespace zugeordnet wird.

### Eindeutigkeit von Aliasen

Jede Kompilationseinheit und jeder Namespacetext verfügt über einen separaten Deklarationsabschnitt für extern-Aliase und using-Aliase. Somit muss der Name eines extern-Alias bzw. using-Alias zwar innerhalb der Menge der extern-Aliase bzw. using-Aliase, die in der unmittelbar enthaltenden Kompilationseinheit bzw. im unmittelbar enthaltenden Namespacetext deklariert sind, eindeutig sein, ein Alias kann jedoch denselben Namen wie ein Typ oder Namespace aufweisen, sofern er ausschließlich mit dem Qualifizierer :: verwendet wird.

In dem Beispiel

namespace N  
{  
 public class A {}

public class B {}  
}

namespace N  
{  
 using A = System.IO;

class X  
 {  
 A.Stream s1; // Error, A is ambiguous

A::Stream s2; // Ok  
 }  
}

hat der Name A im zweiten Namespacetext zwei mögliche Bedeutungen, da sich sowohl die Klasse A als auch der using-Alias A im Gültigkeitsbereich befinden. Daher ist die Verwendung von A im qualifizierten Namen A.Stream mehrdeutig, und es tritt ein Kompilierungsfehler auf. Bei der Verwendung von A mit dem Qualifizierer :: tritt dagegen kein Fehler auf, da der Name A lediglich als Namespacealias gesucht wird.

# Klassen

Eine Klasse ist eine Datenstruktur, die Datenmember (Konstanten und Felder), Funktionsmember (Methoden, Eigenschaften, Indexer, Ereignisse, Operatoren, Instanzkonstruktoren, statische Konstruktoren und Destruktoren) sowie geschachtelte Typen enthalten kann. Klassentypen unterstützen die Vererbung, ein Mechanismus, bei dem eine abgeleitete Klasse eine Basisklasse erweitern und spezialisieren kann.

## Klassendeklarationen

Eine class-declaration ist eine type-declaration (§9.6), die eine neue Klasse deklariert.

class-declaration:  
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
 class-baseopt type-parameter-constraints-clausesopt class-body ;opt

Eine class-declaration besteht aus einer optionalen Gruppe von attributes (§17) gefolgt von einer optionalen Gruppe von class-modifiers (§10.1.1), einem optionalen partial-Modifizierer, dem Schlüsselwort class und einem identifier, der die Klasse benennt. Darauf folgt eine optionale type-parameter-list (§10.1.3), eine optionale class-base-Spezifikation (§), eine optionale Gruppe von type-parameter-constraints-clauses (§10.1.5) und ein class-body (§10.1.6), an den sich optional ein Semikolon anschließt.

Eine Klassendeklaration kann nur dann type-parameter-constraints-clauses bereitstellen, wenn sie auch eine type-parameter-list bereitstellt.

Eine Klassendeklaration, die eine type-parameter-list bereitstellt, ist eine generische Klassendeklaration. Außerdem sind alle in einer generischen Klassendeklaration oder einer generischen Strukturdeklaration geschachtelten Klassen selbst generische Klassendeklarationen, da Typparameter für den enthaltenden Typ bereitgestellt werden müssen, um einen konstruierten Typ zu erstellen.

### Klassenmodifizierer

Eine class-declaration kann optional eine Reihe von Klassenmodifizierern enthalten:

class-modifiers:  
class-modifier  
class-modifiers class-modifier

class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed  
static

Wenn derselbe Modifizierer mehrmals in einer Klassendeklaration aufgeführt ist, wird ein Kompilierungsfehler verursacht.

Der Modifizierer new ist nur bei geschachtelten Klassen zulässig. Er gibt an, dass die Klasse einen geerbten Member mit demselben Namen verdeckt (siehe §10.3.4). Die Verwendung des Modifizierers new in einer Klassendeklaration, die nicht geschachtelt ist, verursacht einen Fehler während der Kompilierung.

Der public-, protected-, internal- und private-Modifizierer steuern den Zugriff auf die Klasse. Abhängig vom Kontext, in dem die Klassendeklaration steht, sind möglicherweise nicht alle Modifizierer zulässig (§3.5.1).

Die Modifizierer abstract, sealed und static werden in den folgenden Abschnitten beschrieben.

#### Abstrakte Klassen

Anhand des abstract-Modifizierers wird angezeigt, dass eine Klasse nicht vollständig ist und nur als Basisklasse vorgesehen ist. Eine abstrakte Klasse unterscheidet sich wie folgt von einer nicht abstrakten Klasse:

* Eine abstrakte Klasse kann nicht direkt instanziiert werden, und in abstrakten Klassen darf der new-Operator nicht verwendet werden, da daraus ein Kompilierungsfehler resultiert. Es können zwar Variablen und Werte vorhanden sein, deren Kompilierungstypen abstrakt sind, diese Variablen und Werte sind jedoch notwendigerweise entweder null oder enthalten Verweise auf Instanzen von nicht abstrakten Klassen, die von den abstrakten Typen abgeleitet wurden.
* Eine abstrakte Klasse darf (muss aber nicht) abstrakte Member enthalten.
* Eine abstrakte Klasse darf nicht versiegelt sein.

Wenn eine nicht abstrakte Klasse von einer abstrakten Klasse abgeleitet wurde, muss die nicht abstrakte Klasse die aktuellen Implementierungen aller geerbten abstrakten Member enthalten. Dadurch werden die vorhandenen abstrakten Member überschrieben. In dem Beispiel

abstract class A  
{  
 public abstract void F();  
}

abstract class B: A  
{  
 public void G() {}  
}

class C: B  
{  
 public override void F() {  
 // actual implementation of F  
 }  
}

führt die abstrakte Klasse A eine abstrakte Methode F ein. Klasse B führt eine zusätzliche Methode G ein, da aber dadurch keine Implementierung von F erfolgt, muss B ebenfalls als abstrakt deklariert werden. Klasse C überschreibt F und sorgt somit für die tatsächliche Implementierung. Da sich in C keine abstrakten Member befinden, ist es zulässig (aber nicht erforderlich), dass C nicht abstrakt ist.

#### Versiegelte Klassen

Der sealed-Modifizierer wird verwendet, um die Ableitung von einer Klasse zu verhindern. Wenn eine versiegelte Klasse als Basisklasse einer anderen Klasse angegeben wird, tritt ein Kompilierungsfehler auf.

Eine versiegelte Klasse kann nicht gleichzeitig eine abstrakte Klasse sein.

Der sealed-Modifizierer wird in erster Linie zum Verhindern unbeabsichtigter Ableitungen verwendet, ermöglicht jedoch auch bestimmte Laufzeitoptimierungen. So können beispielsweise virtuelle Funktionsmemberaufrufe in versiegelten Klasseninstanzen in nicht virtuelle Aufrufe umgewandelt werden, da eine versiegelte Klasse nie abgeleitete Klassen enthält.

#### Statische Klassen

Mit dem static-Modifizierer wird die deklarierte Klasse als eine statische Klasse gekennzeichnet. Eine statische Klasse kann nicht instanziiert oder als Typ verwendet werden. Außerdem darf sie lediglich statische Member enthalten. Nur statische Klassen können Deklarationen von Erweiterungsmethoden (§10.6.9) enthalten.

Die Deklaration einer statischen Klasse unterliegt den folgenden Einschränkungen:

* Eine statische Klasse darf keinen sealed-Modifizierer und keinen abstract-Modifizierer enthalten. Beachten Sie jedoch Folgendes: Da eine statische Klasse nicht instanziiert und nicht abgeleitet werden kann, entspricht ihr Verhalten dem einer versiegelten und einer abstrakten Klasse.
* Eine statische Klasse darf keine class-base-Spezifikation (§10.1.4) enthalten, und sie darf nicht explizit eine Basisklasse oder eine Liste von implementierten Schnittstellen angeben. Eine statische Klasse erbt implizit vom Typ object.
* Eine statische Klasse kann nur statische Member (§10.3.7) enthalten. Beachten Sie, dass Konstanten und geschachtelte Typen als statische Member klassifiziert werden.
* Eine statische Klasse kann keine Member besitzen, für die als Zugriff protected oder protected internal deklariert ist.

Wenn eine der oben stehenden Einschränkungen verletzt wird, tritt ein Kompilierungsfehler auf.

Eine statische Klasse verfügt über keine Instanzkonstruktoren. Es ist nicht möglich, Instanzkonstruktoren in einer statischen Klasse zu deklarieren, und für eine statische Klasse wird kein Standardinstanzkonstruktor (§10.11.4) bereitgestellt.

Die Member einer statischen Klasse sind nicht automatisch statisch, und die Memberdeklarationen müssen explizit einen static-Modifizierer enthalten (mit Ausnahme von Konstanten und geschachtelten Typen). Wenn eine Klasse in einer äußeren statischen Klasse geschachtelt ist, stellt die geschachtelte Klasse keine statische Klasse dar, es sei denn, sie enthält explizit einen static-Modifizierer.

##### Verweisen auf statische Klassentypen

Ein namespace-or-type-name (§3.8) kann unter den folgenden Bedingungen auf eine statische Klasse verweisen:

* Der namespace-or-type-name ist das T in einem namespace-or-type-name der Form T.I oder
* Der namespace-or-type-name ist das T in einer typeof-expression (§7.5.11) der Form typeof(T).

Eine primary-expression (§7.5) kann unter der folgenden Bedingung auf eine statische Klasse verweisen:

* Die primary-expression ist E in einem member-access (§7.5.4) der Form E.I.

In jedem anderen Kontext tritt ein Kompilierungsfehler auf, wenn auf eine statische Klasse verwiesen wird. Beispielsweise liegt ein Fehler vor, wenn eine statische Klasse als Basisklasse, als konstituierender Typ (§10.3.8) eines Members, als generisches Typargument oder als Typparametereinschränkung verwendet wird. Ebenso kann eine statische Klasse nicht in einem Arraytyp, einem Zeigertyp, einem new-Ausdruck, einem Umwandlungsausdruck, einem is-Ausdruck, einem as-Ausdruck, einem sizeof-Ausdruck oder einem Standardwertausdruck verwendet werden.

### Partial-Modifizierer

Mit dem partial-Modifizierer wird angegeben, dass diese class-declaration eine partielle Typdeklaration ist. Mehrere partielle Typdeklarationen mit demselben Namen in einem einschließenden Namespace oder einer einschließenden Typdeklaration bilden zusammen eine Typdeklaration. Dabei gelten die in §10.2 beschriebenen Regeln.

Das Verteilen der Deklaration einer Klasse auf verschiedene Abschnitte des Programmtexts kann hilfreich sein, wenn diese Abschnitt in unterschiedlichen Kontexten erzeugt oder verwaltet werden. Beispielsweise könnte ein Teil einer Klassendeklaration vom Computer generiert werden, während der andere Teil manuell geschrieben wird. Die Trennung dieser beiden Abschnitte im Text verhindert, dass Aktualisierungen eines Abschnitts zu Konflikten mit dem anderen Abschnitt führen.

### Typparameter

Ein Typparameter ist ein einfacher Bezeichner für einen Platzhalter für ein Typargument, das zum Erstellen eines konstruierten Typs angegeben wird. Ein Typparameter ist ein formaler Platzhalter für einen Typ, der zu einem späteren Zeitpunkt bereitgestellt wird. Im Gegensatz dazu ist ein Typargument (§4.4.1) der tatsächliche Typ, der den Typparameter ersetzt, wenn ein konstruierter Typ erstellt wird.

type-parameter-list:  
< type-parameters >

type-parameters:  
attributesopt type-parameter  
type-parameters , attributesopt type-parameter

type-parameter:  
identifier

Jeder Typparameter in einer Klassendeklaration definiert einen Namen im Deklarationsabschnitt (§3.3) der Klasse. Daher kann er nicht denselben Namen wie ein anderer Typparameter oder ein in der Klasse deklarierter Member haben. Ein Typparameter kann nicht denselben Namen wie der Typ selbst haben.

### Basisspezifikation für Klassen

Eine Klassendeklaration kann eine class-base-Spezifikation enthalten, die die direkte Basisklasse der Klasse sowie die Schnittstellen (§13) definiert, die von der Klasse direkt implementiert werden.

class-base:  
: class-type  
: interface-type-list  
: class-type , interface-type-list

interface-type-list:  
interface-type  
interface-type-list , interface-type

Die in einer Klassendeklaration angegebene Basisklasse kann ein konstruierter Klassentyp (§4.4) sein. Eine Basisklasse kann nicht selbst ein Typparameter sein, sie kann aber die Typparameter im Bereich enthalten.

class Extend<V>: V {} // Error, type parameter used as base class

#### Basisklassen

Wenn ein cclass-type in der class-base enthalten ist, gibt dieser die direkte Basisklasse der deklarierten Klasse an. Wenn eine Klassendeklaration keine class-base hat oder die class-base nur Schnittstellentypen enthält, wird davon ausgegangen, dass object die direkte Basisklasse ist. Eine Klasse erbt Member aus deren direkter Basisklasse (siehe §10.3.3).

In dem Beispiel

class A {}

class B: A {}

wird die Klasse A als die direkte Basisklasse von B eingestuft und B wird von A abgeleitet. Da A nicht explizit eine direkte Basisklasse angibt, ist die direkte Basisklasse implizit object.

Wenn bei einem konstruierten Klassentyp in der generischen Klassendeklaration eine Basisklasse angegeben ist, wird die Basisklasse des konstruierten Typs durch Ersetzen jedes type-parameter in der Basisklassendeklaration durch das entsprechende type-argument des konstruierten Typs abgerufen. Bei den generischen Klassendeklarationen

class B<U,V> {...}

class G<T>: B<string,T[]> {...}

ist B<string,int[]> die Basisklasse des konstruierten Typs G<int>.

Die direkte Basisklasse eines Klassentyps muss mindestens genauso verfügbar sein wie der Klassentyp selbst (§3.5.2). Wenn eine public-Klasse beispielsweise von einer private-Klasse oder einer internal-Klasse abgeleitet wird, resultiert daraus ein Kompilierungsfehler.

Für die direkte Basisklasse eines Klassentyps darf keiner der folgenden Typen verwendet werden: System.Array, System.Delegate, System.MulticastDelegate, System.Enum oder System.ValueType. Außerdem kann eine generische Klassendeklaration System.Attribute nicht als direkte oder indirekte Basisklasse verwenden.

Bei der Bestimmung der Bedeutung der direkten Basisklassenspezifikation A einer Klasse B wird vorübergehend davon ausgegangen, dass die direkte Basisklasse von B object ist. Erwartungsgemäß wird dadurch sichergestellt, dass die Bedeutung einer Basisklassenspezifikation nicht rekursiv von sich selbst abhängig sein kann. Beispiel:

class A<T> {

public class B{}

}

class C : A<C.B> {}

führt zu einem Fehler, da in der Basisklassenspezifikation A<C.B> davon ausgegangen wird, dass die direkte Basisklasse von C object ist und daher (gemäß den Regeln von §3.8) nicht davon ausgegangen wird, dass C einen Member B aufweist.

Als Basisklassen eines Klassentyps gelten die direkte Basisklasse und deren Basisklassen. Das bedeutet, dass die Reihe der Basisklassen der transitive Schluss der direkten Basisklassenbeziehung ist. Im oben genannten Beispiel sind A und object die Basisklassen von B. In dem Beispiel

class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}

die Basisklassen von D<int> sind C<int[]>, B<IComparable<int[]>>, A und object.

Außer der Klasse object hat jeder Klassentyp genau eine direkte Basisklasse. Die object-Klasse verfügt nicht über eine direkte Basisklasse und ist die oberste Basisklasse von allen anderen Klassen.

Wenn eine Klasse B von einer Klasse A abgeleitet wird, tritt ein Kompilierungsfehler auf, wenn A von B abhängig ist. Eine Klasse ist direkt abhängig von ihrer direkten Basisklasse (sofern vorhanden), und sie ist direkt abhängig von der Klasse, in der sie unmittelbar geschachtelt ist (sofern vorhanden). Aufgrund dieser Definition ist die vollständige Reihe von Klassen, von der eine Klasse abhängt, der reflexive und transitive Schluss der Beziehung der direkten Abhängigkeit.

In dem Beispiel

class A: A {}

wird ein Fehler verursacht, da die Klasse von sich selbst abhängig ist. Auch das Beispiel

class A: B {}

class B: C {}

class C: A {}

ist fehlerhaft, da die Klassen zirkulär von sich selbst abhängig sind. Das Beispiel

class A: B.C {}

class B: A  
{  
 public class C {}  
}

führt zu einem Kompilierungsfehler, da A von B.C (der direkten Basisklasse) und B.C wiederum von der unmittelbar einschließenden Klasse B abhängig ist, die zirkulär von A abhängt.

Eine Klasse hängt nicht von den Klassen ab, die in ihr geschachtelt sind. In dem Beispiel

class A  
{  
 class B: A {}  
}

hängt B zwar von A ab (da A sowohl die direkte Basisklasse als auch die unmittelbare einschließende Klasse ist), aber A hängt nicht von B ab (da B weder eine Basisklasse noch eine einschließende Klasse von A ist). Aus diesem Grund ist das Beispiel gültig.

Das Ableiten von einer sealed-Klasse ist nicht möglich. In dem Beispiel

sealed class A {}

class B: A {} // Error, cannot derive from a sealed class

tritt bei Klasse B ein Fehler auf, da sie versucht, von der sealed-Klasse A abzuleiten.

#### Schnittstellenimplementierungen

Eine class-base-Spezifikation kann eine Liste von Schnittstellentypen enthalten. Wenn dies der Fall ist, implementiert die Klasse die angegebenen Schnittstellentypen direkt. Schnittstellenimplementierungen werden in §13.4 ausführlicher beschrieben.

### Typparametereinschränkungen

Für generische Typ- und Methodendeklarationen können Typparametereinschränkungen optional mit einer type-parameter-constraints-clause angegeben werden.

type-parameter-constraints-clauses:  
type-parameter-constraints-clause  
type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:  
where type-parameter : type-parameter-constraints

type-parameter-constraints:  
primary-constraint  
secondary-constraints  
constructor-constraint  
primary-constraint , secondary-constraints  
primary-constraint , constructor-constraint  
secondary-constraints , constructor-constraint  
primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:  
class-type  
class  
struct

secondary-constraints:  
interface-type  
type-parameter  
secondary-constraints , interface-type  
secondary-constraints , type-parameter

constructor-constraint:  
new ( )

Jede type-parameter-constraints-clause besteht aus dem Token where, gefolgt vom Namen eines Typparameters, einem Doppelpunkt und der Liste der Einschränkungen für diesen Typparameter. Für jeden Typparameter kann nur eine where-Klausel angegeben werden, und die where-Klauseln können in beliebiger Reihenfolge aufgeführt werden. Wie das get- und set-Token in Eigenschaftenaccessoren ist das where-Token kein Schlüsselwort.

Die Liste der Einschränkungen in einer where-Klausel kann aus den folgenden Komponenten in der angegebenen Reihenfolge bestehen: eine einzelne primäre Einschränkung, eine oder mehrere sekundäre Einschränkungen und die Konstruktoreinschränkung, new().

Eine primäre Einschränkung kann ein Klassentyp, die Verweistypeinschränkung class oder die Werttypeinschränkung struct sein. Eine sekundäre Einschränkung kann ein type-parameter oder ein interface-type sein.

Die Verweistypeinschränkung gibt an, dass ein für den Typparameter verwendetes Typargument ein Verweistyp sein muss. Diese Einschränkung wird von allen Klassentypen, Schnittstellentypen, Delegattypen, Arraytypen und allen Typparametern erfüllt, von denen bekannt ist, dass sie (entsprechend der folgenden Definition) einen Verweistyp darstellen.

Die Werttypeinschränkung gibt an, dass ein für den Typparameter verwendetes Typargument ein Werttyp sein muss, der keine NULL-Werte zulässt. Diese Einschränkung wird von allen Strukturtypen, Enumerationstypen und Typparametern mit Werttypeinschränkung erfüllt, die keine NULL-Werte zulassen. Beachten Sie, dass Typen, die NULL-Werte zulassen (§4.1.10), die Werttypeinschränkung nicht erfüllen, obwohl sie zu den Werttypen zählen. Ein Typparameter mit Werttypeinschränkung kann zudem eine constructor-constraint besitzen.

Zeigertypen dürfen unter keinen Umständen Typargumente sein und erfüllen weder die Verweis- noch die Werttypeinschränkung.

Wenn eine Einschränkung ein Klassentyp, Schnittstellentyp oder Typparameter ist, gibt dieser Typ einen minimalen „Basistyp“ an, der von allen für diesen Typparameter verwendeten Typargumenten unterstützt werden muss. Bei Verwendung eines konstruierten Typs oder einer generischen Methode wird das Typargument zur Kompilierzeit auf die Einschränkungen für den Typparameter hin überprüft. Das angegebene Typargument muss die im Abschnitt 4.4.4 beschriebenen Bedingungen erfüllen.

Eine class-type-Einschränkung muss den folgenden Regeln entsprechen:

* Der Typ muss ein Klassentyp sein.
* Der Typ darf nicht sealed sein.
* Der Typ darf keiner der folgenden Typen sein: System.Array, System.Delegate, System.Enum oder System.ValueType.
* Der Typ darf nicht object sein. Da alle Typen von object abgeleitet sind, hätte es keine Auswirkungen, diese Einschränkung zuzulassen.
* Für einen bestimmten Typparameter darf höchstens eine Einschränkung ein Klassentyp sein.

Ein als interface-type-Einschränkung angegebener Typ muss den folgenden Regeln entsprechen:

* Der Typ muss ein Schnittstellentyp sein.
* Kein Typ darf mehrfach in einer bestimmten where-Klausel angegeben werden.

In jedem Fall kann sich die Einschränkung auf jeden der Typparameter der zugeordneten Typen- oder Methodendeklaration als Teil eines konstruierten Typs beziehen sowie auf den Typ, der deklariert wird.

Jeder Klassen- oder Schnittstellentyp, der als Typparametereinschränkung angegeben wird, muss mindestens so verfügbar (§3.5.4) wie der deklarierte generische Typ oder die deklarierte generische Methode sein.

Ein als type-parameter-Einschränkung angegebener Typ muss den folgenden Regeln entsprechen:

* Der Typ muss ein Typparameter sein.
* Kein Typ darf mehrfach in einer bestimmten where-Klausel angegeben werden.

Außerdem dürfen im Abhängigkeitsdiagramm der Typparameter keine Ringabhängigkeiten auftreten, wobei Abhängigkeit eine transitive Beziehung ist, die wie folgt definiert ist:

* Wenn ein Typparameter T als Einschränkung für den Typparameter S verwendet wird, dann ist S abhängig von T.
* Wenn ein Typparameter S von einem Typparameter T und T von einem Typparameter U abhängig ist, dann ist S abhängig von U.

Bei einer solchen Beziehung führt es zu einem Kompilierungsfehler, wenn ein Typparameter (direkt oder indirekt) von sich selbst abhängig ist.

Alle Abhängigkeiten müssen für alle abhängigen Typparameter konsistent sein. Wenn der Typparameter S vom Typparameter T abhängig ist, gilt Folgendes:

* T darf keine Werteinschränkung aufweisen. Andernfalls wird T dadurch letztlich versiegelt, sodass S zwingend den gleichen Typ wie T besitzen muss, wodurch ein zweiter Parameter überflüssig wird.
* Wenn S die Werttypeinschränkung besitzt, darf T keine class-type-Einschränkung besitzen.
* Wenn S eine class-type-Einschränkung A und T eine class-type-Einschränkung B besitzt, muss eine Identitätskonvertierung oder implizite Verweiskonvertierung von A in B oder eine implizite Verweiskonvertierung von B in A vorhanden sein.
* Wenn S zudem vom Typparameter U abhängig ist, U eine class-type-Einschränkung A und T eine class-type-Einschränkung B besitzt, dann muss eine Identitätskonvertierung oder eine implizite Verweiskonvertierung von A in B oder eine implizite Verweiskonvertierung von B in A vorhanden sein.

Es ist zulässig, dass S die Werttypeinschränkung und T die Verweistypeinschränkung besitzt. In der Konsequenz beschränkt dies T auf die Typen System.Object, System.ValueType, System.Enum und alle Schnittstellentypen.

Wenn die where-Klausel für einen Typparameter eine Konstruktoreinschränkung (mit der Form new()) enthält, können Instanzen des Typs mit dem new-Operator erstellt werden (§7.6.10.1). Alle Typargumente, die für einen Typparameter mit Konstruktoreinschränkung verwendet werden, müssen einen öffentlichen Konstruktor ohne Parameter besitzen (dieser Konstruktor ist implizit für alle Werttypen vorhanden) oder ein Typparameter mit Werttypeinschränkung oder Konstruktoreinschränkung sein (Einzelheiten finden Sie in §10.1.5).

Nachstehend sind einige Beispiele für Einschränkungen aufgeführt:

interface IPrintable  
{  
 void Print();  
}

interface IComparable<T>  
{  
 int CompareTo(T value);  
}

interface IKeyProvider<T>  
{

T GetKey();  
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>  
 where K: IComparable<K>  
 where V: IPrintable, IKeyProvider<K>, new()  
{  
 ...  
}

Das folgende Beispiel enthält einen Fehler, weil es im Abhängigkeitsdiagramm der Typparameter zu einer Ringabhängigkeit kommt:

class Circular<S,T>  
 where S: T  
 where T: S // Error, circularity in dependency graph  
{  
 ...  
}

Im folgenden Beispiel werden weitere nicht gültige Fälle veranschaulicht:

class Sealed<S,T>  
 where S: T  
 where T: struct // Error, T is sealed  
{  
 ...  
}

class A {...}

class B {...}

class Incompat<S,T>  
 where S: A, T  
 where T: B // Error, incompatible class-type constraints  
{  
 ...  
}

class StructWithClass<S,T,U>  
 where S: struct, T  
 where T: U  
 where U: A // Error, A incompatible with struct  
{  
 ...  
}

Die effektive Basisklasse des Typparameters T ist wie folgt definiert:

* Wenn T keine primären Einschränkungen und keine Typparametereinschränkungen besitzt, ist seine effektive Basisklasse object.
* Wenn T die Werttypeinschränkung besitzt, ist seine effektive Basisklasse System.ValueType.
* Wenn T die class-type-Einschränkung C, aber keine type-parameter-Einschränkungen besitzt, ist seine effektive Basisklasse C.
* Wenn T keine class-type-Einschränkung, aber eine oder mehrere type-parameter-Einschränkungen besitzt, ist seine effektive Basisklasse der am stärksten umfasste Typ (§6.4.2) aus der Gruppe der effektiven Basisklassen seiner type-parameter-Einschränkungen. Die Konsistenzregeln stellen sicher, dass ein am stärksten umfasster Typ vorhanden ist.
* Wenn T eine class-type-Einschränkung und eine oder mehrere type-parameter-Einschränkungen besitzt, ist seine effektive Basisklasse der am stärksten umfasste Typ (§6.4.2) aus der Gruppe, die aus der class-type-Einschränkung von T und den effektiven Basisklassen seiner type-parameter-Einschränkungen besteht. Die Konsistenzregeln stellen sicher, dass ein am stärksten umfasster Typ vorhanden ist.
* Wenn T die Verweistypeinschränkung, aber keine class-type-Einschränkungen besitzt, ist seine effektive Basisklasse object.

Wenn T eine Einschränkung V besitzt, die ein value-type ist, verwenden Sie im Rahmen dieser Regeln stattdessen den spezifischsten Basistyp von V, der ein class-type ist. Dies kann niemals in einer explizit angegebenen Einschränkung der Fall sein, ist jedoch möglich, wenn die Einschränkungen einer generischen Methode implizit von einer überschreibenden Methodendeklaration oder einer expliziten Implementierung einer Schnittstellenmethode geerbt werden.

Durch die Regeln wird sichergestellt, dass die effektive Basisklasse immer ein class-type ist.

Die effektive Schnittstellengruppe des Typparameters T ist wie folgt definiert:

* Wenn T keine secondary-constraints besitzt, ist seine effektive Schnittstellengruppe leer.
* Wenn T interface-type-Einschränkungen, aber keine type-parameter-Einschränkungen besitzt, ist seine effektive Schnittstellengruppe die Gruppe seiner interface-type-Einschränkungen.
* Wenn T keine interface-type-Einschränkungen, aber type-parameter-Einschränkungen besitzt, ist seine effektive Schnittstellengruppe die Gesamtmenge der effektiven Schnittstellengruppe seiner type-parameter-Einschränkungen.
* Wenn T interface-type-Einschränkungen und type-parameter-Einschränkungen besitzt, ist seine effektive Schnittstellengruppe die Gesamtmenge seiner Gruppe von interface-type-Einschränkungen und der effektiven Schnittstellengruppen seiner type-parameter-Einschränkungen.

Ein Typparameter ist ein Typ von dem bekannt ist, dass er ein Verweistyp ist, wenn er die Verweistypeinschränkung besitzt oder seine effektive Basisklasse nicht object oder System.ValueType ist.

Werte eines eingeschränkten Typparametertyps können für den Zugriff auf die Instanzmember verwendet werden, die die Einschränkungen implizieren. In dem Beispiel

interface IPrintable  
{  
 void Print();  
}

class Printer<T> where T: IPrintable  
{  
 void PrintOne(T x) {  
 x.Print();  
 }  
}

können die Methoden von IPrintable direkt für x aufgerufen werden, da T die Einschränkung besitzt, immer IPrintable implementieren zu müssen.

### Klassendefinition

Der class-body einer Klasse definiert die Member dieser Klasse.

class-body:  
{ class-member-declarationsopt }

## Partielle Typen

Eine Typdeklaration kann auf mehrere partielle Typdeklarationen verteilt werden. Die Typdeklaration wird nach den in diesem Abschnitt beschriebenen Regeln aus seinen Bestandteilen zusammengesetzt. Während der restlichen Verarbeitung des Programms zur Kompilier- und Laufzeit wird sie dann als eine einzelne Deklaration behandelt.

Eine class-declaration, struct-declaration oder interface-declaration stellt eine partielle Typdeklaration dar, wenn sie einen partial-Modifizierer enthält. partial ist kein Schlüsselwort und fungiert nur dann als Modifizierer, wenn es in einer Typdeklaration direkt vor einem der Schlüsselwörter class, struct oder interface oder in einer Methodendeklaration vor dem Typ void steht. In anderen Kontexten kann es als normaler Bezeichner verwendet werden.

Jeder Teil einer partiellen Typdeklaration muss einen partial-Modifzierer enthalten. Er muss über denselben Namen wie die anderen Teile verfügen und in demselben Namespace oder derselben Typdeklaration deklariert werden. Der partial-Modifizierer gibt an, dass zusätzliche Teile der Typdeklaration an anderer Stelle vorhanden sein können, das Vorhandensein solcher zusätzlicher Teile ist aber nicht erforderlich. Ein Typ in einer einzelnen Deklaration darf den partial-Modifizierer enthalten.

Alle Teile eines partiellen Typs müssen gemeinsam kompiliert werden, damit die Teile zur Kompilierzeit in einer einzigen Typdeklaration zusammengeführt werden können. Mit partiellen Typen ist es nicht möglich, bereits kompilierte Typen zu erweitern.

Geschachtelte Typen können mithilfe des partial-Modifizierers in mehreren Teilen deklariert werden. Normalerweise wird der enthaltende Typ ebenfalls mit partial deklariert, und jeder Teil des geschachtelten Typs wird in einem anderen Teil des enthaltenden Typs deklariert.

Der partial-Modifizierer ist bei Delegaten oder Enumerationsdeklarationen nicht zulässig.

### Attribute

Die Attribute partieller Typen werden durch Kombination der Attribute der einzelnen Teile in nicht festgelegter Reihenfolge bestimmt. Wenn ein Attribut in mehreren Teilen enthalten ist, entspricht dies dem mehrmaligen Angeben des Attributs im Typ. Die beiden Teile

[Attr1, Attr2("hello")]  
partial class A {}

[Attr3, Attr2("goodbye")]  
partial class A {}

sind z. B. äquivalent mit den folgenden Deklarationen:

[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]  
class A {}

Attribute in Typparametern werden auf ähnliche Weise kombiniert.

### Modifizierer

Wenn eine partielle Typdeklaration eine Zugriffsspezifikation (die Modifizierer public, protected, internal und private) enthält, muss diese mit allen anderen Teilen übereinstimmen, in denen ebenfalls eine Zugriffsspezifikation angegeben ist. Wenn kein Teil eines partiellen Typs eine Zugriffsspezifikation enthält, erhält der Typ den entsprechenden Standardzugriff (§3.5.1).

Wenn mindestens eine partielle Deklaration eines geschachtelten Typs einen new-Modifizierer enthält, wird keine Warnung ausgegeben, wenn der geschachtelte Typ einen vererbten Member ausblendet (§3.7.1.2).

Wenn mindestens eine partielle Deklaration einer Klasse einen abstract-Modifizierer enthält, gilt die Klasse als abstrakt (§10.1.1.1). Andernfalls gilt die Klasse als nicht abstrakt.

Wenn mindestens eine partielle Deklaration einer Klasse einen sealed-Modifizierer enthält, gilt die Klasse als versiegelt (§10.1.1.2). Andernfalls gilt die Klasse als nicht versiegelt.

Beachten Sie, dass eine Klasse nicht gleichzeitig abstrakt und versiegelt sein kann.

Wenn der unsafe-Modifizierer in einer partiellen Typdeklaration verwendet wird, gilt nur dieser entsprechende Teil als in einem unsicheren Kontext (§18.1).

### Typparameter und Einschränkungen

Wenn ein generischer Typ in mehreren Teilen deklariert wird, müssen die Typparameter in jedem Teil angegeben werden. Jeder Teil muss über dieselbe Anzahl von Typparametern verfügen. Die Namen der Typparameter müssen ebenfalls auch in der Reihenfolge übereinstimmen.

Wenn eine partielle generische Typdeklaration Einschränkungen enthält (where-Klauseln), müssen die Einschränkungen mit allen anderen Teilen übereinstimmen, die Einschränkungen enthalten. Insbesondere muss jeder Teil, der Einschränkungen enthält, über Einschränkungen für dieselben Typparameter verfügen, und für jeden Typparameter müssen die primären, sekundären und Konstruktoreinschränkungen äquivalent sein. Zwei Sätze von Einschränkungen sind äquivalent, wenn sie dieselben Member enthalten. Wenn kein Teil eines partiellen generischen Typs Einschränkungen für Typparameter angibt, gelten die Typparameter als nicht eingeschränkt.

In dem Beispiel

partial class Dictionary<K,V>  
 where K: IComparable<K>  
 where V: IKeyProvider<K>, IPersistable  
{  
 ...  
}

partial class Dictionary<K,V>  
 where V: IPersistable, IKeyProvider<K>  
 where K: IComparable<K>  
{  
 ...  
}

partial class Dictionary<K,V>  
{  
 ...  
}

ist korrekt, da die Teile mit Einschränkungen (die ersten beiden) effektiv denselben Satz von primären, sekundären und Konstruktoreinschränkungen für jeweils denselben Satz von Typparametern enthalten.

### Basisklasse

Wenn eine partielle Klassendeklaration eine Basisklassenspezifikation enthält, muss diese allen anderen Teilen entsprechen, die eine Basisklassenspezifikation enthalten. Wenn kein Teil einer partiellen klasse eine Basisklassenspezifikation enthält, wird System.Object (§10.1.4.1) als Basisklasse verwendet.

### Basisschnittstellen

Der Satz von Basisschnittstellen für einen in mehreren Teilen deklarierten Typ ist die Union der in jedem Teil angegebenen Basisschnittstellen. Eine spezielle Basisschnittstelle darf in jedem Teil nur einmal benannt sein, mehrere Teile dürfen aber dieselbe(n) Basisschnittstelle(n) benennen. Es darf nur eine Implementierung von Membern einer bestimmten Basisschnittstelle geben.

In dem Beispiel

partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}

lautet der Satz von Basisschnittstellen für die Klasse C: IA, IB und IC.

Normalerweise enthält jeder Teil eine Implementierung der in dem Teil deklarierten Schnittstelle(n). Dies ist aber nicht erforderlich. Ein Teil kann auch die Implementierung einer in einem anderen Teil deklarierten Schnittstelle angeben:

partial class X  
{  
 int IComparable.CompareTo(object o) {...}  
}

partial class X: IComparable  
{  
 ...  
}

### Member

Mit Ausnahme von partiellen Methoden (§10.2.7) ist der Satz der Member eines in mehreren Teilen deklarierten Typs die Union der in den einzelnen Teilen deklarierten Member. Die Rümpfe aller Teile der Typdeklaration verwenden denselben Deklarationsabschnitt (§3.3), und der Bereich aller Member (§3.7) wird auf die Rümpfe aller Teile erweitert. Die Zugriffsdomäne eines Members enthält immer alle Teile des einschließenden Typs. Auf einen in einem Teil deklarierten private-Member kann problemlos von anderen Teilen zugegriffen werden. Wenn derselbe Member in mehreren Teilen eines Typs deklariert wird, tritt ein Kompilierungsfehler auf, sofern dieser Member nicht ein Typ mit dem partial-Modifizierer ist.

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int y;  
 }  
}

partial class A  
{  
 int x; // Error, cannot declare x more than once

partial class Inner // Ok, Inner is a partial type  
 {  
 int z;  
 }  
}

Die Anordnung der Member in einem Typ in C#-Code ist selten von Bedeutung, kann aber bei Überschneidungen mit anderen Sprachen und in anderen Umgebungen bedeutsam sein. In diesen Fällen ist die Anordnung der Member in einem in mehreren Teilen deklarierten Typ nicht definiert.

### Partielle Methoden

Partielle Methoden können in einem Teil einer Typdeklaration definiert und in einem anderen implementiert werden. Die Implementierung ist dabei optional. Wenn kein Teil die partielle Methode implementiert, werden die partielle Methode und ihre sämtlichen Aufrufe aus der Typdeklaration entfernt, die durch das Kombinieren der Teile erstellt wird.

Partielle Methoden können keine Zugriffsmodifizierer definieren, sie sind aber implizit private. Der Rückgabetyp muss void sein, und die Parameter dürfen nicht den out-Modifizierer enthalten. Der Bezeichner partial wird nur dann als Schlüsselwort in einer Methodendeklaration erkannt, wenn er direkt vor dem Typ void steht. In anderen Kontexten kann er wie ein normaler Bezeichner verwendet werden. Partielle Methoden können Schnittstellenmethoden nicht explizit implementieren.

Es gibt zwei Arten von partiellen Methodendeklarationen: Wenn der Text der Methodendeklaration ein Semikolon ist, handelt es sich bei der Deklaration um eine definierende partielle Methodendeklaration. Wenn der Text als block angegeben wird, ist diese Deklaration eine implementierende partielle Methodendeklaration. Innerhalb der einzelnen Teile einer Typdeklaration darf nur eine definierende partielle Methodendeklaration mit einer bestimmten Signatur auftreten. Außerdem darf auch nur eine implementierende partielle Methodendeklaration mit einer bestimmten Signatur enthalten sein. Wenn eine implementierende partielle Methode angegeben wird, muss auch eine zugehörige definierende partielle Methodendeklaration vorhanden sein, wobei diese Deklarationen in folgenden Punkten übereinstimmen müssen:

* Die Deklarationen müssen über dieselben Modifizierer (nicht unbedingt in derselben Reihenfolge), denselben Methodennamen und dieselbe Anzahl von Typparametern und Parametern verfügen.
* Zusammengehörige Parameter in den Deklarationen müssen über dieselben Modifizierer (nicht unbedingt in derselben Reihenfolge) und dieselben Typen (Modulounterschied bei Typparameternamen) verfügen.
* Zusammengehörige Parameter in den Deklarationen müssen über dieselben Einschränkungen (Modulounterschied bei Typparameternamen) verfügen.

Eine implementierende partielle Methode kann in demselben Teil wie die zugehörige definierende partielle Methodendeklaration stehen.

Bei der Überladungsauflösung wird nur eine definierende partielle Methode verwendet. Daher können Aufrufausdrücke unabhängig davon, ob eine implementierende Deklaration angegeben wird, in Aufrufe der partiellen Methode aufgelöst werden. Da eine partielle Methode immer void zurückgibt, sind solche Aufrufausdrücke immer Ausdrucksanweisungen. Außerdem treten diese Anweisungen immer in einem der Teile der Typdeklaration auf, in der die partielle Methode deklariert ist, da eine partielle Methode implizit private ist.

Wenn kein Teil einer partiellen Methodendeklaration eine implementierende Deklaration für eine bestimmte partielle Methode enthält, werden Ausdrucksanweisungen, die diese aufrufen, einfach aus der zusammengesetzten Typdeklaration entfernt. Aus diesem Grund hat dieser Aufrufausdruck, einschließlich aller enthaltenen Ausdrücke, zur Laufzeit keine Auswirkungen. Die partielle Methode selbst wird ebenfalls entfernt und kein Member der zusammengesetzten Typdeklaration.

Wenn für eine bestimmte partielle Methode eine implementierende Deklaration vorhanden ist, werden Aufrufe der partiellen Methoden beibehalten. Die partielle Methode führt zu einer Methodendeklaration, die mit folgenden Ausnahmen der der implementierenden partiellen Methodendeklaration entspricht:

* Es ist kein partial-Modifizierer enthalten.
* Die Attribute der resultierenden Methodendeklaration setzen sich aus den Attributen der definierenden und der implementierenden partiellen Methodendeklaration zusammen, wobei keine festgelegte Reihenfolge existiert. Duplikate werden nicht entfernt.
* Die Attribute in den Parametern der resultierenden Methodendeklaration setzen sich aus den Attributen der entsprechenden Parameter der definierenden und der implementierenden partiellen Methodendeklaration zusammen, wobei keine festgelegte Reihenfolge existiert. Duplikate werden nicht entfernt.

Wenn eine definierende Deklaration, aber keine implementierende Deklaration für eine partielle Methode M angegeben wird, gelten die folgenden Einschränkungen:

* Das Erstellen eines Delegaten zu der Methode führt zu einem Kompilierungsfehler (§7.6.10.5).
* Ein Verweis auf M innerhalb einer anonymen Funktion, die in einen Ausdrucksbaumstrukturtyp konvertiert wird, führt zu einem Kompilierungsfehler (§6.5.2).
* Ausdrücke, die als Teil eines Aufrufs von M auftreten, haben keine Auswirkung auf den definitiven Zuweisungszustand (§5.3). Dies kann möglicherweise zu Kompilierungsfehlern führen.
* M kann nicht der Einstiegspunkt für eine Anwendung sein (§3.1).

Partielle Methoden sind hilfreich, um einem Teil einer Typdeklaration das Anpassen des Verhaltens eines anderen Teils zu erlauben, z. B. eines Teils, der von einem Tool generiert wird. Betrachten Sie z. B. die folgende partielle Klassendeklaration:

partial class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

partial void OnNameChanging(string newName);

partial void OnNameChanged();  
}

Wenn diese Klasse ohne andere Teile kompiliert wird, werden die definierenden Methodendeklarationen und ihre Aufrufe entfernt. Die daraus resultierende zusammengesetzte Klassendeklaration entspricht der folgenden:

class Customer  
{  
 string name;

public string Name {

get { return name; }

set { name = value; }  
 }  
}

Angenommen, es wird ein weiterer Teil angegeben, der jedoch implementierende Deklarationen für die partiellen Methoden angibt:

partial class Customer  
{  
 partial void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

partial void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

Die resultierende zusammengesetzte Klassendeklaration entspricht dann der folgenden:

class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

### Namensbindung

Obwohl jeder Teil eines erweiterbaren Typs innerhalb desselben Namespaces deklariert werden muss, werden diese i. d. R. in unterschiedlichen Namespacedeklarationen geschrieben. Daher können für die einzelnen Teile unterschiedliche using-Direktiven (§9.4) vorhanden sein. Beim Interpretieren einfacher Namen (§7.5.2) in einem Teil werden nur die using-Direktiven der Namespacedeklaration(en) berücksichtigt, die den Teil einschließen. Dies kann dazu führen, dass derselbe Bezeichner in verschiedenen Teilen unterschiedliche Bedeutungen haben kann:

namespace N  
{  
 using List = System.Collections.ArrayList;

partial class A  
 {  
 List x; // x has type System.Collections.ArrayList  
 }  
}

namespace N  
{  
 using List = Widgets.LinkedList;

partial class A  
 {  
 List y; // y has type Widgets.LinkedList  
 }  
}

## Klassenmember

Die Member einer Klasse bestehen einerseits aus den Membern, die von den class-member-declarations eingeführt wurden, und andererseits aus den Membern, die von der direkten Basisklasse geerbt wurden.

class-member-declarations:  
class-member-declaration  
class-member-declarations class-member-declaration

class-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
destructor-declaration  
static-constructor-declaration  
type-declaration

Die Member eines Klassentyps werden in folgende Kategorien eingeteilt:

* Konstanten für die Darstellung konstanter Werte, die mit der Klasse verknüpft sind (§10.4).
* Felder, die Variablen der Klasse sind (§10.5).
* Methoden für die Implementierung der Berechnungen und Aktionen, die von der Klasse ausgeführt werden können (§10.6).
* Eigenschaften zum Definieren benannter Merkmale und von Aktionen, die mit dem Lesen und Schreiben dieser Merkmale verknüpft sind (§10.7).
* Ereignisse zum Definieren von Benachrichtigungen, die von der Klasse generiert werden können (§10.8).
* Indexer, mit denen Instanzen der Klasse auf dieselbe Weise wie Arrays (syntaktisch) indiziert werden können (§10.9).
* Operatoren zum Definieren der Ausdrucksoperatoren, die auf Instanzen der Klasse angewendet werden können (§10.10).
* Instanzkonstruktoren zum Implementieren der Aktionen, die zum Initialisieren von Instanzen der Klasse erforderlich sind (§10.11).
* Destruktoren zum Implementieren der Aktionen, die ausgeführt werden müssen, bevor Instanzen der Klasse permanent verworfen werden (§10.13).
* Statische Konstruktoren zum Implementieren der Aktionen, die zum Initialisieren der Klasse selbst erforderlich sind (§10.12).
* Typen, die in dieser Klasse lokal sind (§10.3.8).

Member, die ausführbaren Code enthalten, werden unter dem Begriff function members des Klassentyps zusammengefasst. Die Funktionsmember eines Klassentyps sind die Methoden, Eigenschaften, Ereignisse, Indexer, Operatoren, Instanzkonstruktoren, Destruktoren und statischen Konstruktoren dieses Klassentyps.

Eine class-declaration erstellt einen neuen Deklarationsabschnitt (§3.3), und durch die unmittelbar in der class-declaration enthaltenen class-member-declarations werden neue Member in diesen Deklarationsabschnitt eingeführt. Für die class-member-declarations gelten folgende Regeln:

* Instanzkonstruktoren, statische Konstruktoren und Destruktoren müssen denselben Namen wie die unmittelbar einschließende Klasse haben. Alle anderen Member müssen Namen haben, die sich von dem Namen der unmittelbar einschließenden Klasse unterscheiden.
* Der Name einer Konstanten, eines Feldes, einer Eigenschaft, eines Ereignisses oder eines Typs muss sich von den Namen aller anderen Member unterscheiden, die in derselben Klasse deklariert sind.
* Der Name einer Methode muss sich von den Namen aller anderen in derselben Klasse deklarierten Nicht-Methoden unterscheiden. Zusätzlich muss sich die Signatur (§3.6) einer Methode von den Signaturen aller anderen Methoden unterscheiden, die in der gleichen Klasse deklariert werden. Die Signaturen zweier Methoden in der gleichen Klasse dürfen sich nicht nur in ref und out unterscheiden.
* Die Signatur eines Instanzkonstruktors muss sich von den Signaturen aller anderen Instanzkonstruktoren unterscheiden, die in der gleichen Klasse deklariert werden. Die Signaturen zweier Konstruktoren in der gleichen Klasse dürfen sich nicht nur in ref und out unterscheiden.
* Die Signatur eines Indexers muss sich von den Signaturen aller anderen, in derselben Klasse deklarierten Indexer unterscheiden.
* Die Signatur eines Operators muss sich von den Signaturen aller anderen Operatoren unterscheiden, die in derselben Klasse deklariert sind.

Die geerbten Member eines Klassentyps (§10.3.3) sind nicht Teil des Deklarationsabschnitts dieser Klasse. Folglich kann eine abgeleitete Klasse einen Member mit demselben Namen oder derselben Signatur wie einen geerbten Member deklarieren (wodurch der geerbte Member eigentlich verdeckt wird).

### Der Instanztyp

Jede Klassendeklaration verfügt über einen zugeordneten gebundenen Typ (§4.4.3) – den Instanztyp. Bei einer generischen Klassendeklaration wird der Instanztyp durch das Erstellen eines konstruierten Typs (§4.4) aus der Typdeklaration erstellt, wobei jedes bereitgestellte Typargument der entsprechende Typparameter ist. Da der Instanztyp die Typparameter verwendet, kann er nur verwendet werden, wenn sich die Typparameter innerhalb des Bereichs befinden, d. h. innerhalb der Klassendeklaration. Der Instanztyp ist der Typ von this für Code, der innerhalb der Klassendeklaration geschrieben wurde. Bei nicht generischen Klassen ist der Instanztyp einfach die deklarierte Klasse. Im Folgenden werden einige Klassendeklarationen zusammen mit ihren Instanztypen gezeigt:

class A<T> // instance type: A<T>  
{  
 class B {} // instance type: A<T>.B

class C<U> {} // instance type: A<T>.C<U>  
}

class D {} // instance type: D

### Member konstruierter Typen

Die nicht vererbten Member eines konstruierten Typs werden durch Ersetzen jedes type-parameter in der Memberdeklaration durch das entsprechende type-argument des konstruierten Typs abgerufen. Die Ersetzung basiert auf der semantischen Bedeutung von Typdeklarationen und ist keine einfache Textersetzung.

Bei der generischen Klassendeklaration

class Gen<T,U>  
{  
 public T[,] a;

public void G(int i, T t, Gen<U,T> gt) {...}

public U Prop { get {...} set {...} }

public int H(double d) {...}  
}

der konstruierte Typ Gen<int[],IComparable<string>> verfügt über die folgenden Member:

public int[,][] a;

public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}

public IComparable<string> Prop { get {...} set {...} }

public int H(double d) {...}

Der Typ des Members a in der generischen Klassendeklaration Gen ist "zweidimensionales Array von T", sodass der Typ des Members a im oben stehenden konstruierten Typ "zweidimensionales Array von eindimensionalen Arrays von int" oder int[,][] lautet.

Innerhalb der Instanzfunktionsmember ist der Typ von this der Instanztyp (§10.3.1) der enthaltenden Deklaration.

Alle Member einer generischen Klasse können Typparameter aus einschließenden Klassen direkt oder als Teil eines konstruierten Typs verwenden. Wenn zur Laufzeit ein bestimmter geschlossen konstruierter Typ (§4.4.2) verwendet wird, wird jede Verwendung eines Typparameters durch das tatsächliche Typargument ersetzt, das im konstruierten Typ bereitgestellt wird. Beispiel:

class C<V>  
{  
 public V f1;  
 public C<V> f2 = null;

public C(V x) {  
 this.f1 = x;  
 this.f2 = this;  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>(1);  
 Console.WriteLine(x1.f1); // Prints 1

C<double> x2 = new C<double>(3.1415);  
 Console.WriteLine(x2.f1); // Prints 3.1415  
 }  
}

### Vererbung

Eine Klasse erbt die Member ihres direkten Basisklassentyps. Vererbung bedeutet, dass eine Klasse implizit alle Member ihres direkten Basisklassentyps enthält. Davon ausgenommen sind lediglich die Instanzkonstruktoren, die statischen Konstruktoren und die Destruktoren der Basisklasse. Hinsichtlich der Vererbung sind folgende Aspekte von Bedeutung:

* Vererbung ist transitiv. Wenn C von B und B von A abgeleitet wird, dann erbt C sowohl die Member, die in B deklariert wurden, als auch die Member, die in A deklariert wurden.
* Eine abgeleitete Klasse erweitert ihre direkte Basisklasse. Sie kann neue Member zu den geerbten hinzufügen, jedoch nicht die Definition eines geerbten Members entfernen.
* Instanzkonstruktoren, Destruktoren und statische Konstruktoren werden nicht vererbt. Alle anderen Member hingegen werden unabhängig von ihrem deklarierten Zugriff vererbt (§3.5). In Abhängigkeit von ihrem deklarierten Zugriff kann auf geerbte Member in einer abgeleiteten Klasse möglicherweise nicht zugegriffen werden.
* Eine abgeleitete Klasse kann geerbte Member verdecken (§3.7.1.2), indem neue Member mit demselben Namen oder derselben Signatur deklariert werden. Geerbte Member werden beim Verdecken nicht entfernt. Das Verdecken bewirkt lediglich, dass direkt über die abgeleitete Klasse nicht auf diesen Member zugegriffen werden kann.
* Eine Instanz einer Klasse enthält eine Gruppe aller Instanzfelder, die in der Klasse und deren Basisklassen deklariert wurden. Außerdem ist eine implizite Konvertierung (§6.1.6) eines abgeleiteten Klassentyps für jeden der Basisklassentypen vorhanden. Somit kann ein Verweis auf eine Instanz einer abgeleiteten Klasse als Verweis auf eine Instanz von einer beliebigen dazugehörigen Basisklasse behandelt werden.
* Eine Klasse kann virtuelle Methoden, Eigenschaften und Indexer deklarieren, wobei abgeleitete Klassen die Implementierung dieser Funktionsmember überschreiben können. Dies ermöglicht das polymorphe Verhalten von Klassen, in denen die von einem Funktionsmemberaufruf ausgeführten Aktionen je nach Laufzeittyp der Instanz variieren, über die der Funktionsmember aufgerufen wird.

Die geerbten Member eines konstruierten Klassentyps sind die Member des direkten Basisklassentyps (§10.1.4.1), der durch Ersetzen der Typargumente des konstruierten Typs für sämtliche Vorkommen der zugehörigen Typparameter in der base-class-specification ermittelt wird. Diese Member werden wiederum umgewandelt, indem jeder type-parameter in der Memberdeklaration durch das entsprechende type-argument der base-class-specification ersetzt wird.

class B<U>  
{  
 public U F(long index) {...}  
}

class D<T>: B<T[]>  
{  
 public T G(string s) {...}  
}

Im oben stehenden Beispiel besitzt der konstruierte Typ D<int> den nicht vererbten Member public int G(string s), der durch Ersetzung des Typarguments int für den Typparameter T abgerufen wurde. D<int> verfügt außerdem über einen vererbten Member aus der Klassendeklaration B. Dieser vererbte Member wird bestimmt, indem zunächst der Basisklassentyp B<int[]> von D<int> bestimmt wird, indem in der Basisklassenspezifikation B<T[]> int durch T ersetzt wird. Dann wird, als Typargument für B, U in public U F(long index) durch int[] ersetzt, woraus sich der geerbte Member public int[] F(long index) ergibt.

### Der new-Modifizierer

Eine class-member-declaration kann zum Deklarieren eines Members mit demselben Namen oder derselben Signatur als geerbter Member verwendet werden. Wenn dieser Fall eintritt, wird der Basisklassenmember vom abgeleiteten Klassenmember verdeckt. Das Verdecken eines geerbten Members ist kein Fehler, aber es führt zu einer Warnung seitens des Compilers. Zum Unterdrücken der Warnung kann die Deklaration des abgeleiteten Klassenmembers den new-Modifizierer enthalten, mit dem angezeigt wird, dass der abgeleitete Member den Basismember verdecken wird. Dieses Thema wird in §3.7.1.2 ausführlicher behandelt.

Wenn der new-Modifizierer in einer Deklaration enthalten ist, die keinen vererbten Member verdeckt, wird eine diesbezügliche Warnung ausgegeben. Diese Warnung wird durch Entfernen des new-Modifizierers unterdrückt.

### Zugriffsmodifizierer

Eine class-member-declaration kann über eine der fünf möglichen Arten des deklarierten Zugriffs verfügen (§3.5.1): public, protected internal, protected, internal oder private. Abgesehen von der Kombination protected internal kann immer nur ein Zugriffsmodifizierer angegeben werden. Ansonsten tritt ein Kompilierungsfehler auf. Wenn eine class-member-declaration keine Zugriffsmodifizierer enthält, wird private angenommen.

### Konstituierende Typen

Typen, die in der Deklaration eines Members verwendet werden, werden als konstituierende Typen des Members bezeichnet. Mögliche konstituierende Typen sind der Typ einer Konstanten, eines Feldes, einer Eigenschaft, eines Ereignisses oder eines Indexers, der Rückgabetyp einer Methode oder eines Operators sowie die Parametertypen einer Methode, eines Indexers, eines Operators oder eines Instanzkonstruktors. Der konstituierende Typ eines Members muss mindestens genauso verfügbar sein wie der Member selbst (§3.5.4).

### Statische Member und Instanzmember

Bei Membern einer Klasse handelt es sich entweder um statische Member oderr um Instanzmember. Prinzipiell ist es sinnvoll, davon auszugehen, dass statische Member zu Klassentypen und Instanzmember zu Objekten gehören (Instanzen von Klassentypen).

Wenn eine Felddeklaration, eine Methodendeklaration, eine Eigenschaftendeklaration, eine Ereignisdeklaration, eine Operatordeklaration oder eine Konstruktordeklaration einen static-Modifizierer enthält, deklariert sie einen statischen Member. Außerdem wird durch eine Konstantendeklaration oder eine Typdeklaration implizit ein statischer Member deklariert. Statische Member haben folgende Eigenschaften:

* Wenn in einem member-access (§7.6.4) mit der Form E.M auf einen statischen Member M verwiesen wird, muss E auf einen Typ deuten, der M enthält. Wenn E auf eine Instanz deutet, tritt während der Kompilierung ein Fehler auf.
* Ein statisches Feld gibt genau einen Speicherort an, der von allen Instanzen eines angegebenen geschlossenen Klassentyps gemeinsam verwendet werden soll. Unabhängig von der Anzahl der Instanzen eines angegebenen geschlossenen Klassentyps gibt es immer nur eine Kopie eines statischen Feldes.
* Ein statischer Funktionsmember (Methode, Eigenschaft, Ereignis, Operator oder Konstruktor) funktioniert nicht in einer bestimmten Instanz. In einem solchen Funktionsmember darf nicht auf this verwiesen werden, da andernfalls ein Kompilierungsfehler auftritt.

Wenn eine Felddeklaration, eine Methodendeklaration, eine Eigenschaftendeklaration, eine Ereignisdeklaration, eine Indexerdeklaration, eine Konstruktordeklaration oder eine Destruktordeklaration keinen static-Modifizierer enthält, deklariert sie einen Instanzmember. (Instanzmember werden auch als nicht statische Member bezeichnet.) Instanzmember haben folgende Eigenschaften:

* Wenn in einem member-access (§7.6.4) mit der Form E.M auf einen Instanzmember M verwiesen wird, muss E auf eine Instanz eines Typs deuten, der M enthält. Wenn E auf einen Typ deutet, tritt während der Bindung ein Fehler auf.
* Jede Instanz einer Klasse enthält eine separate Gruppe aller Instanzfelder der Klasse.
* Ein Instanzfunktionsmember (Methode, Eigenschaft, Indexer, Instanzkonstruktor oder Destruktor) wird auf eine spezifische Instanz der Klasse angewendet, und auf diese Instanz kann so wie auf this zugegriffen werden (§7.6.7).

Anhand des folgenden Beispiels werden die Regeln für den Zugriff auf statische Member und Instanzmember dargestellt:

class Test  
{  
 int x;  
 static int y;

void F() {  
 x = 1; // Ok, same as this.x = 1  
 y = 1; // Ok, same as Test.y = 1  
 }

static void G() {  
 x = 1; // Error, cannot access this.x  
 y = 1; // Ok, same as Test.y = 1  
 }

static void Main() {  
 Test t = new Test();  
 t.x = 1; // Ok  
 t.y = 1; // Error, cannot access static member through instance  
 Test.x = 1; // Error, cannot access instance member through type  
 Test.y = 1; // Ok  
 }  
}

Die F-Methode zeigt, dass in einem Instanzfunktionsmember ein simple-name (§7.6.2) für den Zugriff auf Instanzmember und statische Member verwendet werden kann. Anhand der G-Methode wird deutlich, dass in einem statischen Funktionsmember nicht über einen simple-name auf einen Instanzmember zugegriffen werden darf, da ansonsten ein Kompilierungsfehler auftritt. Die Main-Methode zeigt, dass in einem member-access (§7.6.4) auf Instanzmember über Instanzen und auf statische Member über Typen zugegriffen werden muss.

### Geschachtelte Typen

Ein Typ, der innerhalb einer Klassen- oder einer Strukturdeklaration deklariert wird, wird als geschachtelter Typ bezeichnet. Ein Typ, der innerhalb einer Kompilationseinheit oder eines Namespaces deklariert wird, wird als nicht geschachtelter Typ bezeichnet.

In dem Beispiel

using System;

class A  
{  
 class B  
 {  
 static void F() {  
 Console.WriteLine("A.B.F");  
 }  
 }  
}

ist Klasse B ein geschachtelter Typ, weil sie innerhalb von Klasse A deklariert wird, und Klasse A ist ein nicht geschachtelter Typ, weil sie innerhalb einer Kompilationseinheit deklariert wird.

#### Vollqualifizierter Name

Der vollqualifizierte Name (§3.8.1) für einen geschachtelten Typ lautet S.N, wobei S der vollqualifizierte Name des Typs ist, in dem N deklariert wird.

#### Deklarierter Zugriff

Nicht geschachtelte Typen können mit dem Zugriff public oder internal deklariert sein. Standardmäßig wird der als internal deklarierte Zugriff verwendet. Geschachtelte Typen können ebenfalls diese Formen sowie eine oder mehrere zusätzliche Formen des deklarierten Zugriffs haben (je nachdem, ob der enthaltende Typ eine Klasse oder eine Struktur ist):

* Ein geschachtelter Typ, der in einer Klasse deklariert wird, kann eine von fünf Formen des deklarierten Zugriffs haben (public, protected internal, protected, internal oder private). Standardmäßig wird genau wie bei anderen Klassenmembern der als private deklarierte Zugriff verwendet.
* Ein geschachtelter Typ, der in einer Struktur deklariert wird, kann eine von drei Formen des deklarierten Zugriffs haben (public, internal oder private). Standardmäßig wird genau wie bei anderen Strukturmembern der als private deklarierte Zugriff verwendet.

In dem Beispiel

public class List  
{  
 // Private data structure  
 private class Node  
 {   
 public object Data;  
 public Node Next;

public Node(object data, Node next) {  
 this.Data = data;  
 this.Next = next;  
 }  
 }

private Node first = null;  
 private Node last = null;

// Public interface

public void AddToFront(object o) {...}

public void AddToBack(object o) {...}

public object RemoveFromFront() {...}

public object RemoveFromBack() {...}

public int Count { get {...} }  
}

deklariert eine privat geschachtelte Klasse Node.

#### Verdecken

Ein geschachtelter Typ kann einen Basismember verdecken (§3.7.1). Der new-Modifizierer kann in Deklarationen mit geschachteltem Typ verwendet werden, sodass das Verdecken explizit ausgedrückt werden kann. In dem Beispiel

using System;

class Base  
{  
 public static void M() {  
 Console.WriteLine("Base.M");  
 }  
}

class Derived: Base   
{  
 new public class M   
 {  
 public static void F() {  
 Console.WriteLine("Derived.M.F");  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.M.F();  
 }  
}

enthält eine geschachtelte Klasse M, welche die Methode M verdeckt, die in Base definiert ist.

#### This-Zugriff

Ein geschachtelter Typ und der enthaltende Typ haben keine besondere Beziehung im Hinblick auf den this-access (§7.6.7). this kann insbesondere innerhalb eines geschachtelten Typs nicht zum Verweisen auf Instanzmember des enthaltenden Typs verwendet werden. Wenn ein geschachtelter Typ auf die Instanzmember seines enthaltenden Typs Zugriff erhalten soll, kann this für die Instanz des enthaltenden Typs als Konstruktorargument des geschachtelten Typs zur Verfügung gestellt werden. Durch folgendes Beispiel

using System;

class C  
{  
 int i = 123;

public void F() {  
 Nested n = new Nested(this);  
 n.G();  
 }

public class Nested  
 {  
 C this\_c;

public Nested(C c) {  
 this\_c = c;  
 }

public void G() {  
 Console.WriteLine(this\_c.i);  
 }  
 }  
}

class Test  
{  
 static void Main() {  
 C c = new C();  
 c.F();  
 }  
}

wird diese Methode dargestellt. Eine C-Instanz erstellt eine Instanz von Nested und übergibt ihr eigenes this an den Konstruktor von Nested, um anschließend den Zugriff auf die Instanzmember von C zu ermöglichen.

#### Zugriff auf private und geschützte Member des enthaltenden Typs

Ein geschachtelter Typ hat Zugriff auf alle Member, auf die sein enthaltender Typ zugreifen kann. Dazu zählen auch Member des enthaltenden Typs, die mit dem Zugriff private und protected deklariert wurden. In dem Beispiel

using System;

class C   
{  
 private static void F() {  
 Console.WriteLine("C.F");  
 }

public class Nested   
 {  
 public static void G() {  
 F();  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 C.Nested.G();  
 }  
}

zeigt eine Klasse C, die eine geschachtelte Klasse Nested enthält. Innerhalb von Nested ruft die Methode G die statische Methode F auf, die in C definiert ist, wobei F wiederum mit dem privaten Zugriff deklariert wurde.

Ein geschachtelter Typ kann auch auf geschützte Member zugreifen, die in einem Basistyp des enthaltenden Typs definiert sind. In dem Beispiel

using System;

class Base   
{  
 protected void F() {  
 Console.WriteLine("Base.F");  
 }  
}

class Derived: Base   
{  
 public class Nested   
 {  
 public void G() {  
 Derived d = new Derived();  
 d.F(); // ok  
 }  
 }  
}

class Test   
{  
 static void Main() {  
 Derived.Nested n = new Derived.Nested();  
 n.G();  
 }  
}

greift die geschachtelte Derived.Nested-Klasse auf die geschützte Methode F zu, die in der Basisklasse von Derived mit dem Namen Base definiert ist, indem sie über eine Instanz von Derived aufgerufen wird.

#### Geschachtelte Typen in generischen Klassen

Eine generische Klassendeklaration kann geschachtelte Typdeklarationen enthalten. Die Typparameter der einschließenden Klasse können in den geschachtelten Typen verwendet werden. Eine geschachtelte Typdeklaration kann zusätzliche Typparameter enthalten, die nur für den geschachtelten Typ gelten.

Jede in einer generischen Klassendeklaration enthaltene Typdeklaration ist implizit eine generische Typdeklaration. Beim Schreiben eines Verweises auf eine in einem generischen Typ geschachtelten Typ muss der enthaltende konstruierte Typ einschließlich seiner Typargumente benannt sein. Bei äußeren Klassen kann der geschachtelte Typ aber ohne Qualifizierung verwendet werden. Der Instanztyp der äußeren Klasse kann implizit beim Konstruieren des geschachtelten Typs verwendet werden. Im folgenden Beispiel werden drei unterschiedliche, aber korrekte Vorgehensweisen für das Verweisen auf einen aus Inner konstruierten Typ gezeigt. Die ersten beiden Beispiele sind äquivalent:

class Outer<T>  
{  
 class Inner<U>  
 {  
 public static void F(T t, U u) {...}  
 }

static void F(T t) {  
 Outer<T>.Inner<string>.F(t, "abc"); // These two statements have  
 Inner<string>.F(t, "abc"); // the same effect

Outer<int>.Inner<string>.F(3, "abc"); // This type is different

Outer.Inner<string>.F(t, "abc"); // Error, Outer needs type arg  
 }  
}

Obwohl dies keinen guten Programmierstil darstellt, kann ein Typparameter in einem geschachtelten Typ einen im äußeren Typ deklarierten Member oder Typparameter verdecken:

class Outer<T>  
{  
 class Inner<T> // Valid, hides Outer’s T  
 {  
 public T t; // Refers to Inner’s T  
 }  
}

### Reservierte Membernamen

Zur Vereinfachung der zugrunde liegenden C#-Laufzeitimplementierung muss die Implementierung für jede Quellmemberdeklaration, bei der es sich um eine Eigenschaft, ein Ereignis oder einen Indexer handelt, auf der Grundlage der Art der Memberdeklaration sowie des Namens und Typs des Members zwei Methodensignaturen reservieren. Ein Programm darf keinen Member deklarieren, dessen Signatur mit einer dieser reservierten Signaturen übereinstimmt, auch wenn die zugrunde liegende Laufzeitimplementierung diese Reservierungen nicht verwendet. Andernfalls tritt ein Kompilierungsfehler auf.

Da die reservierten Namen keine Deklarationen einführen, nehmen sie nicht an der Membersuche teil. Die verknüpften reservierten Methodensignaturen einer Deklaration nehmen dagegen an der Vererbung (§10.3.3) teil und können durch den new-Modifizierer (§10.3.4) verdeckt werden.

Mit der Reservierung dieser Namen werden drei Ziele verfolgt:

* Es soll möglich sein, dass die zugrunde liegende Implementierung zum Erhalten oder Festlegen des Zugriffs auf die C#-Sprachfunktion einen normalen Bezeichner als Methodenname verwendet.
* Es soll möglich sein, dass andere Sprachen zusammenwirken, indem sie als Methodenname zum Erhalten oder Festlegen des Zugriffs auf die C#-Sprachfunktion einen normalen Bezeichner verwenden.
* Es soll sichergestellt werden, dass die Quelle, die von einem entsprechenden Compiler akzeptiert wird, durch eine Vereinheitlichung der Spezifikationen der reservierten Membernamen in allen C#-Implementierungen auch von einem anderen Compiler akzeptiert werden kann.

Mit der Deklaration eines Destruktors (§10.13) wird auch eine Signatur reserviert (§10.3.9.4).

#### Für Eigenschaften reservierte Membernamen

Für eine EigenschaftP (§10.7) vom Typ T sind die folgenden Signaturen reserviert:

T get\_P();  
void set\_P(T value);

Beide Signaturen sind auch dann reserviert, wenn die Eigenschaft schreibgeschützt oder lesegeschützt ist.

In dem Beispiel

using System;

class A  
{  
 public int P {  
 get { return 123; }  
 }  
}

class B: A  
{  
 new public int get\_P() {  
 return 456;  
 }

new public void set\_P(int value) {  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 A a = b;  
 Console.WriteLine(a.P);  
 Console.WriteLine(b.P);  
 Console.WriteLine(b.get\_P());  
 }  
}

definiert eine Klasse A eine schreibgeschützte Eigenschaft P, wodurch Signaturen für die Methoden get\_P und set\_P reserviert werden. Eine Klasse B wird von A abgeleitet und verdeckt beide reservierten Signaturen. Das Beispiel erzeugt folgende Ausgabe:

123  
123  
456

#### Für Ereignisse reservierte Membernamen

Für ein Ereignis E (§10.8) vom Delegattyp T sind die folgenden Signaturen reserviert:

void add\_E(T handler);  
void remove\_E(T handler);

#### Für Indexer reservierte Membernamen

Für einen Indexer (§10.9) vom Typ T mit der Parameterliste L sind die folgenden Signaturen reserviert:

T get\_Item(L);  
void set\_Item(L, T value);

Beide Signaturen sind auch dann reserviert, wenn der Indexer schreibgeschützt oder lesegeschützt ist.

Darüber hinaus ist der Membername Item reserviert.

#### Für Destruktoren reservierte Membernamen

Für eine Klasse, die einen Destruktor enthält (§10.13), ist die folgende Signatur reserviert:

void Finalize();

## Konstanten

Eine Konstante ist ein Klassenmember, der einen konstanten Wert repräsentiert, d. h. einen Wert, der während der Kompilierung berechnet werden kann. Eine constant-declaration führt eine oder mehrere Konstanten eines bestimmten Typs ein.

constant-declaration:  
attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers:  
constant-modifier  
constant-modifiers constant-modifier

constant-modifier:  
new  
public  
protected  
internal  
private

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

Eine constant-declaration kann eine Gruppe von attributes (§17), einen new-Modifizierer (§10.3.4) und eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5) enthalten. Die Attribute und Modifizierer gelten für alle Member, die durch die constant-declaration deklariert wurden. Obwohl Konstanten als statische Member angesehen werden, sind static-Modifizierer in einer constant-declaration weder erforderlich noch zulässig. Wenn derselbe Modifizierer mehrmals in einer Konstantendeklaration aufgeführt ist, wird ein Fehler verursacht.

Der type einer constant-declaration gibt den Typ der Member an, die durch die Deklaration eingeführt wurden. Auf den Typ folgt eine Liste von constant-declarators, die jeweils einen neuen Member einführen. Ein constant-declarator besteht aus einem identifier, der den Member benennt, gefolgt von einem "="-Token, an das sich wiederum eine constant-expression (§7.19) anschließt, die den Wert des Members angibt.

Bei dem in einer Konstantendeklaration angegebenen type muss es sich um sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, einen enum-type oder einen reference-type handeln. Jede constant-expression muss einen Wert des Zieltyps oder eines Typs liefern, der implizit in den Zieltyp konvertiert (§6.1) werden kann.

Der type einer Konstanten muss mindestens genauso verfügbar sein wie die Konstante selbst (§3.5.4).

Der Wert einer Konstanten wird in einem Ausdruck mithilfe eines simple-name (§7.6.2) oder mithilfe eines member-access (§7.6.4) ermittelt.

Eine Konstante kann Teil einer constant-expression sein. Demnach kann eine Konstante in einem beliebigen Konstrukt verwendet werden, für das eine constant-expression erforderlich ist. Solche Konstrukte sind beispielsweise case-Bezeichnungen, goto case-Anweisungen, enum-Memberdeklarationen, Attribute und andere Konstantendeklarationen.

Eine constant-expression ist ein Ausdruck, der während der Kompilierung vollständig ausgewertet werden kann (siehe §7.19). Da die einzige Möglichkeit, von einem reference-type, der nicht string ist, einen Wert ungleich null zu erstellen, in der Anwendung des new-Operators besteht und da der new-Operator in einer constant-expression nicht verwendet werden darf, ist null der einzige zulässige Wert für Konstanten von reference-types, die nicht string sind.

Wenn Sie für einen konstanten Wert einen symbolischen Namen verwenden möchten, aber entweder der Typ des Werts in einer konstanten Deklaration nicht zulässig ist oder sich der Wert während der Kompilierung von einer constant-expression nicht berechnen lässt, kann stattdessen ein readonly-Feld (§10.5.2) verwendet werden.

Eine Konstantendeklaration, die mehrere Konstanten deklariert, ist äquivalent mit mehreren Deklarationen einzelner Konstanten, die dieselben Attribute, Modifizierer und Typen aufweisen. Beispiel:

class A  
{  
 public const double X = 1.0, Y = 2.0, Z = 3.0;  
}

äquivalent zu

class A  
{  
 public const double X = 1.0;  
 public const double Y = 2.0;  
 public const double Z = 3.0;  
}

Konstanten können innerhalb eines Programms von anderen Konstanten abhängig sein, solange keine zirkulären Abhängigkeiten entstehen. Der Compiler ordnet die Konstantendeklarationen für die Berechnung automatisch in der geeigneten Reihenfolge an. In dem Beispiel

class A  
{  
 public const int X = B.Z + 1;  
 public const int Y = 10;  
}

class B  
{  
 public const int Z = A.Y + 1;  
}

berechnet der Compiler zunächst A.Y, dann B.Z und schließlich A.X und erzeugt dabei die Werte 10, 11 und 12. Konstantendeklarationen können von Konstanten aus anderen Programmen abhängig sein. Allerdings sind diese Abhängigkeiten nur in einer Richtung möglich. Wenn im oben genannten Beispiel A und B in separaten Programmen deklariert wurden, kann zwar A.X von B.Z, aber B.Z nicht gleichzeitig von A.Y abhängen.

## Felder

Ein field ist ein Member, der eine mit einem Objekt oder einer Klasse verknüpfte Variable repräsentiert. Eine field-declaration führt ein oder mehrere Felder eines bestimmten Typs ein.

field-declaration:  
attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:  
field-modifier  
field-modifiers field-modifier

field-modifier:  
new  
public  
protected  
internal  
private  
static  
readonly  
volatile

variable-declarators:  
variable-declarator  
variable-declarators , variable-declarator

variable-declarator:  
identifier  
identifier = variable-initializer

variable-initializer:  
expression  
array-initializer

Eine field-declaration kann eine Gruppe von attributes (§17), einen new-Modifizierer (§10.3.4), eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5) und einen static-Modifizierer (§10.5.1) enthalten. Außerdem kann eine field-declaration entweder einen readonly-Modifizierer (§10.5.2) oder einen volatile-Modifizierer (§10.5.3), nicht jedoch beide gleichzeitig enthalten. Die Attribute und Modifizierer treffen für alle Member zu, die durch die field-declaration deklariert wurden. Wenn derselbe Modifizierer mehrmals in einer Felddeklaration aufgeführt ist, wird ein Fehler verursacht.

Mit dem type einer field-declaration wird der Typ der Member angegeben, die durch die Deklaration eingeführt wurden. Auf den Typ folgt eine Liste von variable-declarators, die jeweils einen neuen Member einführen. Ein variable-declarator besteht aus einem identifier, der den Member benennt, optional gefolgt von einem "="-Token und einem variable-initializer (§10.5.5), der den Anfangswert des Members angibt.

Der type eines Felds muss mindestens genauso verfügbar sein wie das Feld selbst (§3.5.4).

Der Wert eines Felds wird in einem Ausdruck mithilfe eines simple-name (§7.6.2) oder einem member-access (§7.6.4) ermittelt. Die Änderung des Werts eines nicht schreibgeschützten Felds erfolgt mithilfe eines assignment (§7.17). Der Wert eines nicht schreibgeschützten Felds kann mithilfe von Postfix-Inkrementoperatoren und Postfix-Dekrementoperatoren (§7.6.9) sowie Präfix-Inkrementoperatoren und Präfix-Dekrementoperatoren (§7.7.5) ermittelt und geändert werden.

Eine Felddeklaration, die mehrere Felder deklariert, ist äquivalent mit mehreren Deklarationen einzelner Felder, die dieselben Attribute und Modifizierer sowie denselben Typ enthalten. Beispiel:

class A  
{  
 public static int X = 1, Y, Z = 100;  
}

äquivalent zu

class A  
{  
 public static int X = 1;  
 public static int Y;  
 public static int Z = 100;  
}

### Statische Felder und Instanzfelder

Wenn eine Felddeklaration einen static-Modifizierer enthält, handelt es sich bei den von der Deklaration eingeführten Feldern um statische Felder. Wenn kein static-Modifizierer vorhanden ist, sind die von der Deklaration eingeführten Felder instance fields. Statische Felder und Instanzfelder sind zwei von vielen Variablenarten (§5), die von C# unterstützt werden. Sie werden auch als statische Variablen bzw. Instanzvariablen bezeichnet.

Ein statisches Feld ist nicht Teil einer bestimmten Instanz, sondern wird vielmehr von allen Instanzen eines geschlossenen Typs gemeinsam verwendet (§4.4.2). Unabhängig von der Anzahl der Instanzen eines geschlossenen Klassentyps gibt es immer nur eine Kopie eines statischen Feldes für die zugehörige Anwendungsdomäne.

Beispiel:

class C<V>  
{  
 static int count = 0;

public C() {  
 count++;  
 }

public static int Count {  
 get { return count; }  
 }  
}

class Application  
{  
 static void Main() {  
 C<int> x1 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<double> x2 = new C<double>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<int> x3 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 2  
 }  
}

Ein Instanzfeld gehört zu einer Instanz. Jede Instanz einer Klasse enthält eine separate Gruppe aller Instanzfelder der Klasse.

Wenn in einem member-access (§7.6.4) der Form E.M auf ein Feld verwiesen wird und M ein statisches Feld ist, muss E einen Typ angeben, der M enthält. Wenn M ein Instanzfeld ist, muss E eine Instanz eines Typs angeben, der M enthält.

Die Unterschiede zwischen statischen Membern und Instanzmembern werden in §10.3.7 näher erläutert.

### Schreibgeschützte Felder

Wenn eine field-declaration einen readonly-Modifizierer enthält, gelten die von der Deklaration eingeführten Felder als schreibgeschützte Felder. Direkte Zuweisungen zu einem schreibgeschützten Feld können nur als Teil der Deklaration oder in einem Instanzkonstruktor bzw. einem statischen Konstruktor derselben Klasse vorgenommen werden. (In diesen Kontexten kann ein schreibgeschütztes Feld mehrere Male zugewiesen werden.) Direkte Zuweisungen zu einem readonly-Feld sind nur in den folgenden Kontexten zulässig:

* Im variable-declarator, der das Feld einführt (durch Hinzufügen einer variable-initializer in der Deklaration).
* Für ein Instanzfeld in den Instanzkonstruktoren der Klasse, die die Felddeklaration enthält; für ein statisches Feld im statischen Konstruktor der Klasse, die die Felddeklaration enthält. Hierbei handelt sich auch um die einzigen Kontexte, in denen ein readonly-Feld als out-Parameter oder als ref-Parameter übergeben werden darf.

In allen anderen Kontexten führt das Zuweisen zu einem readonly-Feld bzw. das Übergeben dieses Felds als out-Parameter oder ref-Parameter zu einem Kompilierungsfehler.

#### Verwenden von statischen schreibgeschützten Feldern für Konstanten

Ein static readonly-Feld ist sinnvoll, wenn ein konstanter Wert einen symbolischen Namen haben soll, aber der Typ des Werts in einer const-Deklaration nicht zulässig ist, oder wenn der Wert während der Kompilierung nicht berechnet werden kann. In dem Beispiel

public class Color  
{  
 public static readonly Color Black = new Color(0, 0, 0);  
 public static readonly Color White = new Color(255, 255, 255);  
 public static readonly Color Red = new Color(255, 0, 0);  
 public static readonly Color Green = new Color(0, 255, 0);  
 public static readonly Color Blue = new Color(0, 0, 255);

private byte red, green, blue;

public Color(byte r, byte g, byte b) {  
 red = r;  
 green = g;  
 blue = b;  
 }  
}

können der Black-, White-, Red-, Green- und Blue-Member nicht als const-Member deklariert werden, da ihre Werte während der Kompilierung nicht berechnet werden können. Wenn sie stattdessen als static readonly deklariert werden, erzielen Sie denselben Effekt.

#### Versionskontrolle von Konstanten und statischen schreibgeschützten Feldern

Die binäre Versionskontrollensemantik von Konstanten und schreibgeschützten Feldern ist unterschiedlich. Wenn ein Ausdruck auf eine Konstante verweist, wird der Wert der Konstanten während der Kompilierung ermittelt, aber wenn ein Ausdruck auf ein schreibgeschütztes Feld verweist, wird der Wert nicht vor der Laufzeit ermittelt. Im folgenden Beispiel wird davon ausgegangen, dass eine Anwendung aus zwei separaten Programmen besteht:

using System;

namespace Program1  
{  
 public class Utils  
 {  
 public static readonly int X = 1;  
 }  
}

namespace Program2  
{  
 class Test  
 {  
 static void Main() {  
 Console.WriteLine(Program1.Utils.X);  
 }  
 }  
}

Der Program1-Namespace und der Program2-Namespace deuten hier auf zwei Programme, die separat kompiliert werden. Da Program1.Utils.X als statisches schreibgeschütztes Feld deklariert wurde, ist der von der Console.WriteLine-Anweisung ausgegebene Wert während der Kompilierung nicht bekannt, sondern wird während der Laufzeit ermittelt. Wenn also der Wert von X geändert und Program1 neu kompiliert wird, gibt die Console.WriteLine-Anweisung den neuen Wert auch dann aus, wenn Program2 nicht neu kompiliert wurde. Ist X jedoch eine Konstante, wird der Wert von X während der Kompilierung von Program2 zwar ermittelt, aber die Änderungen in Program1 werden erst nach dem erneuten Kompilieren in Program2 wirksam.

### Flüchtige Felder

Wenn eine field-declaration einen volatile-Modifizierer enthält, handelt es sich bei den von der Deklaration eingeführten Feldern um flüchtige Felder.

Bei nicht flüchtigen Feldern können Optimierungsmethoden, die Anweisungen neu anordnen, zu unerwarteten und unvorhersehbaren Ergebnissen in Multithreadingprogrammen führen, die ohne Synchronisierung auf Felder zugreifen, z. B. ohne die vom lock-statement zur Verfügung gestellte Synchronisierung (§8.12). Diese Optimierungen können von einem Compiler, dem Laufzeitsystem oder von der Hardware ausgeführt werden. Bei flüchtigen Feldern sind solche Optimierungen in Form einer Neuanordnung beschränkt:

* Das Lesen eines flüchtigen Felds wird als flüchtiger Lesevorgang bezeichnet. Ein flüchtiger Lesevorgang verfügt über eine „ermittelnde Semantik“: Er erfolgt garantiert vor jeglichen Verweisen auf den Speicher, die in der Anweisungsfolge nach diesem Vorgang auftreten.
* Das Schreiben eines flüchtigen Felds wird als flüchtiger Schreibvorgang bezeichnet. Ein flüchtiger Schreibvorgang verfügt über eine „freigebende Semantik“: Er erfolgt garantiert nach jeglichen Speicherverweisen, die in der Anweisungsfolge vor der Schreibanweisung stehen.

Durch diese Einschränkungen wird sichergestellt, dass flüchtige, von anderen Threads ausgeführte Schreibvorgänge von allen Threads in der Reihenfolge ihrer Ausführung berücksichtigt werden. Es ist keine entsprechende Implementierung erforderlich, um eine einheitliche Gesamtanordnung der flüchtigen Schreibvorgänge für alle beteiligten Ausführungsthreads darzustellen. Der Typ eines flüchtigen Feldes muss einem der folgenden Typen entsprechen:

* Einem reference-type.
* Der Typ byte, sbyte, short, ushort, int, uint, char, float, bool, System.IntPtr oder System.UIntPtr.
* Ein enum-type mit dem Enumerationsbasistyp byte, sbyte, short, ushort, int oder uint.

In dem Beispiel

using System;  
using System.Threading;

class Test  
{  
 public static int result;   
 public static volatile bool finished;

static void Thread2() {  
 result = 143;   
 finished = true;   
 }

static void Main() {  
 finished = false;

// Run Thread2() in a new thread  
 new Thread(new ThreadStart(Thread2)).Start();

// Wait for Thread2 to signal that it has a result by setting  
 // finished to true.  
 for (;;) {  
 if (finished) {  
 Console.WriteLine("result = {0}", result);  
 return;  
 }  
 }  
 }  
}

erzeugt die Ausgabe:

result = 143

In diesem Beispiel startet die Main-Methode einen neuen Thread, der die Methode Thread2 ausführt. Diese Methode speichert zunächst einen Wert im nicht flüchtigen Feld result und anschließend den Wert true im flüchtigen Feld finished. Der Hauptthread wartet, bis das Feld finished auf true gesetzt wird, und liest dann das Feld result. Da finished als volatile deklariert wurde, muss der Hauptthread den Wert 143 aus dem Feld result lesen. Wäre das Feld finished nicht als volatile deklariert worden, wäre es möglich gewesen, dass der Speichervorgang für result nach dem Speichervorgang für finished für den Hauptthread sichtbar gewesen wäre. Daher hätte der Hauptthread den Wert 0 aus dem Feld result lesen können. Durch die Deklaration von finished als volatile-Feld werden derartige Inkonsistenzen vermieden.

### Feldinitialisierung

Der anfängliche Wert eines Felds ist, unabhängig davon, ob es sich um ein statisches Feld oder ein Instanzfeld handelt, der Standardwert (§5.2) des Feldtyps. Da der Wert eines Felds vor dieser Standardinitialisierung nicht berücksichtigt werden kann, ist ein Feld unter keinen Umständen „nicht initialisiert“. In dem Beispiel

using System;

class Test  
{  
 static bool b;  
 int i;

static void Main() {  
 Test t = new Test();  
 Console.WriteLine("b = {0}, i = {1}", b, t.i);  
 }  
}

erzeugt die Ausgabe

b = False, i = 0

weil b und i automatisch mit den Standardwerten initialisiert werden.

### Variableninitialisierungen

Felddeklarationen können variable-initializers enthalten. Bei statischen Feldern entsprechen Variableninitialisierungen den Zuweisungsanweisungen, die während der Klasseninitialisierung ausgeführt werden. Bei Instanzfeldern entsprechen Variableninitialisierungen den Zuweisungsanweisungen, die ausgeführt werden, wenn eine Instanz der Klasse erstellt wird.

In dem Beispiel

using System;

class Test  
{  
 static double x = Math.Sqrt(2.0);  
 int i = 100;  
 string s = "Hello";

static void Main() {  
 Test a = new Test();  
 Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);  
 }  
}

erzeugt die Ausgabe

x = 1.4142135623731, i = 100, s = Hello

weil beim Ausführen des Initialisierers für statische Felder eine Zuweisung zu x und beim Ausführen des Instanzfeldinitialisierers Zuweisungen zu i und s erfolgen.

Die Standardwertinitialisierung (siehe §10.5.4) erfolgt für alle Felder, einschließlich der Felder, die über Variableninitialisierungen verfügen. Bei der Initialisierung einer Klasse werden demnach zunächst alle statischen Felder der Klasse mit ihren Standardwerten initialisiert. Anschließend werden die Initialisierungen der statischen Felder in der Textreihenfolge ausgeführt. Beim Erstellen einer Instanz einer Klasse werden zunächst alle dazugehörigen Instanzfelder mit ihren Standardwerten initialisiert. Anschließend werden die Instanzfeldinitialisierungen in der Textreihenfolge ausgeführt.

Statische Felder mit Variableninitialisierungen können bei Angabe des Standardwerts beobachtet werden. Allerdings wird aus stilistischen Gründen dringend davon abgeraten. In dem Beispiel

using System;

class Test  
{  
 static int a = b + 1;  
 static int b = a + 1;

static void Main() {  
 Console.WriteLine("a = {0}, b = {1}", a, b);  
 }  
}

wird dieses Verhalten dargestellt. Trotz der zirkulären Definitionen von a und b ist dieses Programm gültig. Es erzeugt folgende Ausgabe

a = 1, b = 2

weil die statischen Felder a und b vor der Ausführung des Initialisierer mit 0 (dem Standardwert für int) initialisiert werden. Wenn der Initialisierer für a ausgeführt wird, hat b den Wert 0, sodass a mit 1 initialisiert wird. Wenn der Initialisierer für b ausgeführt wird, hat a bereits den Wert 1, sodass b mit 2 initialisiert wird.

#### Initialisierung statischer Felder

Die Variableninitialisierung der statischen Felder einer Klasse entspricht einer Folge von Zuweisungen, die in der Textreihenfolge ausgeführt werden, in der sie in der Klassendeklaration vorkommen. Wenn in der Klasse ein statischer Konstruktor (§) vorhanden ist, wird die Initialisierung statischer Felder unmittelbar vor der Ausführung dieses statischen Konstruktors ausgeführt. Andernfalls werden die Initialisierungen statischer Felder zu einem implementierungsabhängigen Zeitpunkt ausgeführt, der vor der ersten Verwendung eines statischen Feldes dieser Klasse liegt. In dem Beispiel

using System;

class Test   
{   
 static void Main() {  
 Console.WriteLine("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Console.WriteLine(s);  
 return 1;  
 }  
}

class A  
{  
 public static int X = Test.F("Init A");  
}

class B  
{  
 public static int Y = Test.F("Init B");  
}

kann entweder folgende Ausgabe erzeugen

Init A  
Init B  
1 1

oder folgende Ausgabe

Init B  
Init A  
1 1

Dies liegt daran, dass die Ausführung der Initialisierer für X und Y in einer der beiden Reihenfolgen erfolgen kann. Sie müssen lediglich vor den Verweisen auf diese Felder ausgeführt werden. In diesem Beispiel

using System;

class Test  
{  
 static void Main() {  
 Console.WriteLine("{0} {1}", B.Y, A.X);  
 }

public static int F(string s) {  
 Console.WriteLine(s);  
 return 1;  
 }  
}

class A  
{  
 static A() {}

public static int X = Test.F("Init A");  
}

class B  
{  
 static B() {}

public static int Y = Test.F("Init B");  
}

muss die Ausgabe wie folgt aussehen:

Init B  
Init A  
1 1

Dies liegt daran, dass die Regeln für den Ausführungszeitpunkt statischer Konstruktoren (wie in §10.12 definiert) vorsehen, dass der statische Konstruktor von B (und folglich Initialisierer statischer Felder von B) vor dem statischen Konstruktor und den Feldinitialisierer von A ausgeführt werden müssen.

#### Initialisierung von Instanzfeldern

Die Variableninitialisierung von Instanzfeldern einer Klasse entspricht einer Reihe von Zuweisungen, die unmittelbar nach Eingang in einen der Instanzkonstruktoren (§10.11.1) der Klasse ausgeführt werden. Die Variableninitialisierungen werden in der Textreihenfolge ausgeführt, in der sie in der Klassendeklaration aufgeführt werden. Die Klasseninstanzerstellung und der Initialisierungsprozess werden in §10.11 näher erläutert.

Eine Variableninitialisierung für ein Instanzfeld kann nicht auf die zu erstellende Instanz verweisen. Somit darf in einem Variableninitialisierer nicht auf this verwiesen werden, da es als Kompilierungsfehler gilt, wenn ein Variableninitialisierer über einen simple-name auf einen Instanzmember verweist. In dem Beispiel

class A  
{  
 int x = 1;  
 int y = x + 1; // Error, reference to instance member of this  
}

führt der Variableninitialisierer für y zu einem Kompilierungsfehler, weil auf einen Member der zu erstellenden Instanz verwiesen wird.

## Methoden

Methoden sind Member, die eine Berechnung oder Aktion implementieren, die von einem Objekt oder einer Klasse ausgeführt werden kann. Methoden werden mithilfe von method-declarations deklariert:

method-declaration:  
method-header method-body

method-header:  
attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers:  
method-modifier  
method-modifiers method-modifier

method-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern  
async

return-type:  
type  
void

member-name:  
identifier  
interface-type . identifier

method-body:  
block  
;

Eine method-declaration kann eine Gruppe von attributes (§17) und eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5), dem new (§10.3.4)-, static (§10.6.2)-, virtual (§10.6.3)-, override (§10.6.4)-, sealed (§10.6.5)-, abstract (§10.6.6)- und extern (§10.6.7)-Modifizierer enthalten.

Eine Deklaration verfügt über eine gültige Kombination von Modifizierern, wenn alle folgenden Voraussetzungen zutreffen:

* Die Deklaration umfasst eine gültige Kombination aus Zugriffsmodifizierern (§10.3.5).
* Die Deklaration enthält keinen Modifizierer mehrfach.
* Die Deklaration enthält maximal einen der folgenden Modifizierer: static, virtual und override.
* Die Deklaration enthält maximal einen der folgenden Modifizierer: new und override.
* Wenn der Modifizierer abstract in der Deklaration enthalten ist, darf zusätzlich keiner der folgenden Modifizierer enthalten sein: static, virtual, sealed oder extern.
* Wenn der Modifizierer private in der Deklaration enthalten ist, darf zusätzlich keiner der folgenden Modifizierer enthalten sein: virtual, override oder abstract.
* Wenn der Modifizierer sealed in der Deklaration enthalten ist, muss zusätzlich der Modifizierer override vorhanden sein.
* Wenn der Modifizierer partial in der Deklaration enthalten ist, darf zusätzlich keiner der folgenden Modifizierer enthalten sein: new, public, protected, internal, private, virtual, sealed, override, abstract oder extern.

Eine Methode mit dem Async-Modifizierer ist eine Async-Funktion und folgt den in §10.14 beschriebenen Regeln.

Der return-type einer Methodendeklaration gibt den Typ des berechneten und von der Methode zurückgegebenen Werts an. Wenn die Methode keinen Wert zurückgibt, ist der return-type void. Wenn die Deklaration den partial-Modifizierer enthält, muss der Rückgabetyp void sein.

Der member-name gibt den Namen der Methode an. Sofern die Methode keine explizite Implementierung eines Schnittstellenmembers (§13.4.1) ist, handelt es sich bei dem member-name einfach um einen identifier. Damit ein Schnittstellenmember explizit implementiert werden kann (§REF), besteht der member-name aus einem interface-type gefolgt von einem "." und einem identifier.

Die optionale type-parameter-list gibt die Typparameter der Methode an (§10.1.3). Wenn eine type-parameter-list angegeben wurde, handelt es sich bei der Methode um eine generische Methode.. Wenn die Methode einen extern-Modifizierer enthält, kann keine type-parameter-list angegeben werden.

Die optionale formal-parameter-list gibt die Parameter der Methode an (§10.6.1).

Die optionalen type-parameter-constraints-clauses geben Einschränkungen für einzelne Typparameter (§10.1.5). Sie können nur dann angegeben werden, wenn auch eine type-parameter-list bereitgestellt wird und die Methode keinen override-Modifizierer enthält.

Der return-type und alle Typen, auf die in der formal-parameter-list einer Methode verwiesen wird, müssen mindestens genauso verfügbar sein wie die Methode selbst (§3.5.4).

Bei der abstract-Methode und der extern-Methode besteht der method-body einfach aus einem Semikolon. Bei partial-Methoden besteht der method-body entweder aus einem Semikolon oder einem block. Bei allen anderen Methoden besteht der method-body-body aus einem block, der die Anweisungen angibt, die beim Aufruf der Methode auszuführen sind.

Wenn der method-body aus einem Semikolon besteht, darf die Deklaration keinen async-Modifizierer enthalten.

Anhand des Namens, der Typparameterliste und der Liste formaler Parameter einer Methode wird die Signatur (§3.6) einer Methode definiert. Die Signatur einer Methode besteht aus dem Namen der Methode, der Anzahl von Typparametern sowie der Anzahl, den Modifizierern und den Typen der formalen Parameter. Zu diesem Zweck werden sämtliche Typparameter der Methode, die im Typ eines formalen Parameters enthalten sind, nicht anhand ihres Namens, sondern anhand ihrer Ordinalposition in der Typargumentliste der Methode identifiziert. Der Rückgabetyp ist nicht Teil der Signatur einer Methode, dies gilt auch für die Namen der Typparameter oder der formalen Parameter.

Der Name einer Methode muss sich von den Namen aller anderen in derselben Klasse deklarierten Nicht-Methoden unterscheiden. Zusätzlich muss sich die Signatur einer Methode von den Signaturen aller anderen Methoden unterscheiden, die in der gleichen Klasse deklariert werden. Die Signaturen zweier Methoden in der gleichen Klasse dürfen sich nicht nur in ref und out unterscheiden.

Der Gültigkeitsbereich der type-parameter der Methode ist die gesamte method-declaration, und sie können verwendet werden, um Typen in diesem gesamten Gültigkeitsbereich in return-type, method-body und type-parameter-constraints-clauses zu bilden, jedoch nicht in attributes.

Alle formalen Parameter und Typparameter müssen über unterschiedliche Namen verfügen.

### Methodenparameter

Die Parameter einer Methode (sofern vorhanden) werden von der formal-parameter-list der Methode deklariert.

formal-parameter-list:  
fixed-parameters  
fixed-parameters , parameter-array  
parameter-array

fixed-parameters:  
fixed-parameter  
fixed-parameters , fixed-parameter

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

Die Liste formaler Parameter besteht aus einem oder mehreren durch Trennzeichen getrennten Parametern, von denen nur der letzte ein parameter-array sein kann.

Ein fixed-parameter besteht aus einer optionalen Gruppe von attributes (§17), einem optionalen ref-, out- oder this-Modifizierer, einem type, einem identifier und einem optionalen default-argument. Jeder fixed-parameter deklariert einen Parameter des angegebenen Typs mit dem angegebenen Namen. Der this-Modifizierer kennzeichnet die Methode als eine Erweiterungsmethode und ist nur für den ersten Parameter einer statischen Methode zulässig. Einzelheiten zu Erweiterungsmethoden finden Sie in §10.6.9.

Ein fixed-parameter mit einem default-argument ist als optionaler Parameter bekannt, wohingegen ein fixed-parameter ohne default-argument ein erforderlicher Parameter ist. In einer formal-parameter-list darf ein erforderlicher Parameter nicht nach einem optionalen Parameter stehen.

Ein ref- oder out-Parameter darf kein default-argument besitzen. Die expression in einem default-argument muss eine der folgenden Optionen sein:

* Eine constant-expression.
* Ein Ausdruck in der Form new S(), wobei S ein Werttyp ist.
* Ein Ausdruck in der Form new default(S), wobei S ein Werttyp ist.

Die expression muss von einer Identität oder Konvertierung, die NULL-Werte zulässt, implizit in den Parametertyp konvertiert werden können.

Wenn optionale Parameter in einer implementierenden partiellen Methodendeklaration (§10.2.7), einer expliziten Schnittstellenmemberimplementation (§13.4.1) oder einer Indexerdeklaration mit einem einzigen Parameter (§10.9) auftreten, sollte der Compiler eine Warnung ausgeben, da diese Member niemals auf eine Art und Weise aufgerufen werden können, dass das Auslassen von Argumente zulässt.

Ein parameter-array besteht aus einer optionalen Gruppe von attributes (§17), einem params-Modifizierer, einem array-type und einem identifier. Ein Parameterarray deklariert einen einzelnen Parameter des angegebenen Arraytyps mit dem angegebenen Namen. Der array-type eines Parameterarrays muss ein eindimensionaler Arraytyp sein (§12.1). In einem Methodenaufruf ermöglicht ein Parameterarray entweder die Angabe eines einzelnen Arguments eines bestimmten Arraytyps oder die Angabe keines oder mehrerer Argumente des Arrayelementtyps. Parameterarrays werden in §10.6.1.4 näher beschrieben.

Ein parameter-array kann nach einem optionalen Parameter auftreten, kann jedoch keinen Standardwert aufweisen – die Auslassung von Argumenten für ein parameter-array würde stattdessen zur Erstellung eines leeren Arrays führen.

Das folgende Beispiel veranschaulicht die unterschiedlichen Arten von Parametern:

public void M(  
 ref int i,  
 decimal d,  
 bool b = false,  
 bool? n = false,  
 string s = "Hello",  
 object o = null,  
 T t = default(T),  
 params int[] a  
) { }

In der formal-parameter-list für M ist i ein erforderlicher ref-Parameter, d ist ein erforderlicher Werteparameter, b, s, o und t sind optionale Wertparameter, und a ist ein Parameterarray.

Eine Methodendeklaration erstellt einen separaten Deklarationsabschnitt für Parameter, Typparameter und lokale Variablen. Namen werden in diesen Deklarationsabschnitt anhand der Typparameterliste und der Liste formaler Parameter der Methode sowie anhand von Deklarationen lokaler Variablen im block der Methode eingeführt. Wenn zwei Member eines Methodendeklarationsabschnitts den gleichen Namen besitzen, führt dies zu einem Fehler. Ebenso stellt es einen Fehler dar, wenn der Methodendeklarationsabschnitt und der Deklarationsabschnitt für lokale Variablen eines geschachtelten Deklarationsabschnitts Elemente mit dem gleichen Namen enthalten.

Ein Methodenaufruf (§7.6.5.1) erstellt speziell für diesen Aufruf eine Kopie der formalen Parameter und lokalen Variablen der Methode. Dabei weist die Argumentenliste des Aufrufs den neu erstellten formalen Parametern Werte oder Variablenverweise zu. Innerhalb des block einer Methode kann auf formale Parameter anhand der dazugehörigen Bezeichner in simple-name-Ausdrücken (§7.6.2) verwiesen werden.

Es gibt vier Arten von formalen Parametern:

* Werteparameter, die ohne Modifizierer deklariert werden.
* Verweisparameter, die mit dem ref-Modifizierer deklariert werden.
* Ausgabeparameter, die mit dem out-Modifizierer deklariert werden.
* Parameterarrays, die mit dem params-Modifizierer deklariert werden.

Der ref-Modifizierer und der out-Modifizierer sind Teil einer Methodensignatur. Für den params-Modifizierer gilt dies nicht (siehe §3.6).

#### Werteparameter

Ein Parameter, der ohne Modifizierer deklariert wird, ist ein Werteparameter. Ein Werteparameter entspricht einer lokalen Variablen, die den anfänglichen Wert von dem entsprechenden Argument erhält, das in dem Methodenaufruf angegeben wird.

Wenn ein formaler Parameter ein Werteparameter ist, muss das entsprechende Argument in einem Methodenaufruf ein Ausdruck sein, der implizit in den Typ des formalen Parameters konvertierbar ist (§6.1).

Es besteht die Möglichkeit, dass eine Methode einem Werteparameter neue Werte zuweist. Solche Zuweisungen wirken sich lediglich auf den lokalen Speicherort aus, der durch den Werteparameter dargestellt wird. Auf das Argument in dem Methodenaufruf haben sie keinen Einfluss.

#### Verweisparameter

Ein Parameter, der mit einem ref-Modifizierer deklariert wird, ist ein Verweisparameter. Im Gegensatz zum Werteparameter erstellt der Verweisparameter keinen neuen Speicherort. Ein Verweisparameter stellt denselben Speicherort wie die Variable dar, die in dem Methodenaufruf als Argument angegeben wurde.

Wenn es sich bei einem formalen Parameter um einen Verweisparameter handelt, muss das entsprechende Argument im Methodenaufruf aus dem Schlüsselwort ref, gefolgt von einer variable-reference (§5.3.3) desselben Typs wie der formale Parameter bestehen. Eine Variable muss endgültig zugewiesen werden, bevor sie als Verweisparameter weitergegeben werden kann.

Es wird davon ausgegangen, dass Verweisparameter, die sich innerhalb einer Methode befinden, endgültig zugewiesen wurden.

Eine als Iterator (§10.14) deklarierte Methode darf keine Verweisparameter enthalten.

In dem Beispiel

using System;

class Test  
{  
 static void Swap(ref int x, ref int y) {  
 int temp = x;  
 x = y;  
 y = temp;  
 }

static void Main() {  
 int i = 1, j = 2;  
 Swap(ref i, ref j);  
 Console.WriteLine("i = {0}, j = {1}", i, j);  
 }  
}

erzeugt die Ausgabe

i = 2, j = 1

Für den Aufruf von Swap in Main wird i durch x und j durch y dargestellt. Demnach hat der Aufruf zur Folge, dass die Werte von i und j vertauscht werden.

In einer Methode, in der Verweisparameter verwendet werden, können mehrere Namen zum Darstellen eines Speicherortes verwendet werden. In dem Beispiel

class A  
{  
 string s;

void F(ref string a, ref string b) {  
 s = "One";  
 a = "Two";  
 b = "Three";  
 }

void G() {  
 F(ref s, ref s);  
 }  
}

gibt der Aufruf von F in G einen Verweis auf s für a und b weiter. Demnach verweisen die Namen s, a und b auf denselben Speicherort, und die drei Zuweisungen bewirken eine Änderung des Instanzfelds s.

#### Ausgabeparameter

Ein Parameter, der mit einem out-Modifizierer deklariert wird, ist ein Ausgabeparameter. Ähnlich wie Verweisparameter erstellt ein Ausgabeparameter keinen neuen Speicherort. Stattdessen stellt er denselben Speicherort wie die Variable dar, die in dem Methodenaufruf als Argument angegeben wurde.

Wenn es sich bei einem formalen Parameter um einen Ausgabeparameter handelt, muss das entsprechende Argument in dem Methodenaufruf aus dem Schlüsselwort out, gefolgt von einer variable-reference (§5.3.3) desselben Typs wie der formale Parameter bestehen. Eine Variable muss nicht endgültig zugewiesen sein, bevor sie als Ausgabeparameter weitergegeben werden kann. Wenn sie aber auf einen Aufruf folgt, in dem eine Variable als Ausgabeparameter weitergegeben wurde, wird die Variable als endgültig zugewiesen betrachtet.

Innerhalb einer Methode wird ein Ausgabeparameter anfänglich so wie eine lokale Variable als nicht zugewiesen betrachtet und muss vor der Verwendung seines Wertes endgültig zugewiesen werden.

Alle Ausgabeparameter einer Methode müssen vor Rückgabe der Methode endgültig zugewiesen werden.

Eine als partielle Methode (§10.2.7) oder als Iterator (§10.14) deklarierte Methode darf keine Ausgabeparameter enthalten.

Ausgabeparameter werden in der Regel in Methoden verwendet, die mehrere Rückgabewerte erzeugen. Beispiel:

using System;

class Test  
{  
 static void SplitPath(string path, out string dir, out string name) {  
 int i = path.Length;  
 while (i > 0) {  
 char ch = path[i – 1];  
 if (ch == '\\' || ch == '/' || ch == ':') break;  
 i--;  
 }  
 dir = path.Substring(0, i);  
 name = path.Substring(i);  
 }

static void Main() {  
 string dir, name;  
 SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);  
 Console.WriteLine(dir);  
 Console.WriteLine(name);  
 }  
}

Das Beispiel erzeugt folgende Ausgabe:

c:\Windows\System\  
hello.txt

Beachten Sie, dass die dir-Variable und die name-Variable vor der Weitergabe an SplitPath nicht zugewiesen werden müssen und nach dem Aufruf als definitiv zugewiesen betrachtet werden.

#### Parameterarrays

Ein Parameter, der mit einem params-Modifizierer deklariert wird, ist ein Parameterarray. Wenn eine Liste formaler Parameter ein Parameterarray enthält, muss es sich um den letzten Parameter in der Liste und einen eindimensionalen Arraytyp handeln. Beispielsweise können die Typen string[] und string[][] als Typ eines Parameterarrays verwendet werden, der Typ string[,] jedoch nicht. Der params-Modifizierer kann nicht mit dem ref-Modifizierer und dem out-Modifizierer kombiniert werden.

Mit einem Parameterarray können Argumente in einem Methodenaufruf auf zweierlei Weise festgelegt werden:

* Das Argument, das für ein Parameterarray angegeben wird, kann ein einzelner Ausdruck sein, der implizit in den Parameterarraytyp konvertierbar ist (§6.1). In diesem Fall hat das Parameterarray dieselbe Funktion wie ein Werteparameter.
* Außerdem besteht die Möglichkeit, dass der Aufruf keines oder mehrere Argumente für das Parameterarray angibt, wobei jedes Argument ein Ausdruck ist, der implizit in den Elementtyp des Parameterarrays konvertierbar ist (§6.1). In diesem Fall erstellt der Aufruf eine Instanz des Parameterarraytyps mit einer Länge, die der Anzahl der Argumente entspricht, initialisiert die Elemente der Arrayinstanz mit den angegebenen Argumentwerten und verwendet die neu erstellte Arrayinstanz als das tatsächliche Argument.

Abgesehen davon, dass eine variable Anzahl von Argumenten in einem Aufruf zulässig ist, stimmt ein Parameterarray stets mit einem Werteparameter (§10.6.1.1) desselben Typs überein.

In dem Beispiel

using System;

class Test  
{  
 static void F(params int[] args) {  
 Console.Write("Array contains {0} elements:", args.Length);  
 foreach (int i in args)   
 Console.Write(" {0}", i);  
 Console.WriteLine();  
 }

static void Main() {  
 int[] arr = {1, 2, 3};  
 F(arr);  
 F(10, 20, 30, 40);  
 F();  
 }  
}

erzeugt die Ausgabe

Array contains 3 elements: 1 2 3  
Array contains 4 elements: 10 20 30 40  
Array contains 0 elements:

Der erste Aufruf von F gibt einfach das Array a als Werteparameter weiter. Der zweite Aufruf von F erstellt automatisch ein int[] aus vier Elementen mit den angegebenen Elementwerten und gibt die Arrayinstanz als Werteparameter weiter. Der dritte Aufruf von F erstellt ein int[] ohne Elemente und gibt diese Instanz als Werteparameter weiter. Der zweite Aufruf und der dritte Aufruf sehen wie folgt aus:

F(new int[] {10, 20, 30, 40});  
F(new int[] {});

Beim Ausführen der Überladungsauflösung kann eine Methode mit einem Parameterarray entweder in der normalen Form oder in der erweiterten Form angewendet werden (§7.5.3.1). Die erweiterte Form einer Methode ist nur verfügbar, wenn die normale Form der Methode nicht anwendbar ist und im selben Typ noch keine anwendbare Methode deklariert wurde, die über dieselbe Signatur wie die erweiterte Form verfügt.

In dem Beispiel

using System;

class Test  
{  
 static void F(params object[] a) {  
 Console.WriteLine("F(object[])");  
 }

static void F() {  
 Console.WriteLine("F()");  
 }

static void F(object a0, object a1) {  
 Console.WriteLine("F(object,object)");  
 }

static void Main() {  
 F();  
 F(1);  
 F(1, 2);  
 F(1, 2, 3);  
 F(1, 2, 3, 4);  
 }  
}

erzeugt die Ausgabe

F();  
F(object[]);  
F(object,object);  
F(object[]);  
F(object[]);

In diesem Beispiel sind zwei der möglichen erweiterten Formen der Methode mit einem Parameterarray bereits in der Klasse als reguläre Methoden enthalten. Diese erweiterten Formen werden demnach bei der Überladungsauflösung nicht berücksichtigt, sodass im ersten und dritten Methodenaufruf die reguläre Methode gewählt wird. Wenn eine Klasse eine Methode mit einem Parameterarray deklariert, ist es nicht unüblich, auch einige der erweiterten Formen als reguläre Methoden hinzuzufügen. Dadurch kann die Reservierung einer Arrayinstanz vermieden werden. Diese Reservierung tritt ein, wenn eine erweiterte Form einer Methode mit einem Parameterarray aufgerufen wird.

Wenn der Typ eines Parameterarrays object[] lautet, kann es zwischen der normalen Form der Methode und der erweiterten Form für einen einzelnen object-Parameter zu Mehrdeutigkeiten kommen. Der Grund für die Unklarheit besteht darin, dass ein object[] selbst implizit in den Typ object konvertierbar ist. Die Unklarheit stellt zwar kein Problem dar, sie kann jedoch bei Bedarf durch Einfügen einer Umwandlung aufgelöst werden.

In dem Beispiel

using System;

class Test  
{  
 static void F(params object[] args) {  
 foreach (object o in args) {  
 Console.Write(o.GetType().FullName);  
 Console.Write(" ");  
 }  
 Console.WriteLine();  
 }

static void Main() {  
 object[] a = {1, "Hello", 123.456};  
 object o = a;  
 F(a);  
 F((object)a);  
 F(o);  
 F((object[])o);  
 }  
}

erzeugt die Ausgabe

System.Int32 System.String System.Double  
System.Object[]  
System.Object[]  
System.Int32 System.String System.Double

In dem ersten und dem letzten Aufruf von F ist die normale Form von F anwendbar, da eine implizite Konvertierung aus dem Argumenttyp in den Parametertyp vorhanden ist (beide vom Typ object[]). Daher wählt die Überladungsauflösung die normale Form von F, und das Argument wird als regulärer Werteparameter weitergegeben. In dem zweiten und dritten Aufruf ist die normale Form von F nicht anwendbar, da keine implizite Konvertierung aus dem Argumenttyp in den Parametertyp vorhanden ist (der Typ object kann nicht implizit in den Typ object[] konvertiert werden). Die erweiterte Form von F ist jedoch anwendbar und wird daher von der Überladungsauflösung gewählt. Daraufhin wird ein object[] mit einem Element erstellt und ein einzelnes Element des Arrays mit dem angegebenen Argumentwert initialisiert (das selbst ein Verweis auf object[] ist).

### Statische Methoden und Instanzmethoden

Wenn eine Methodendeklaration einen static-Modifizierer enthält, wird die Methode als statische Methode bezeichnet. Wenn kein static-Modifizierer vorhanden ist, wird die Methode als Instanzmethode bezeichnet.

Eine statische Methode funktioniert nicht in einer bestimmten Instanz. Wird in einer statischen Methode auf this verwiesen, wird ein Kompilierungsfehler ausgegeben.

Eine Instanzmethode funktioniert in einer bestimmten Instanz einer Klasse, auf die so wie auf this zugegriffen werden kann (§7.6.7).

Wenn in einem member-access (§7.6.4) der Form E.M auf eine Methode verwiesen wird und M eine statische Methode ist, muss E einen Typ mit einem Feld M bezeichnen. Wenn M eine Instanzmethode ist, muss E eine Instanz eines Typs bezeichnen, der M enthält.

Die Unterschiede zwischen statischen Membern und Instanzmembern werden in §10.3.7 näher erläutert.

### Virtuelle Methoden

Wenn die Deklaration einer Instanzmethode einen virtual-Modifizierer enthält, wird diese Methode als virtuelle Methode bezeichnet. Wenn kein virtual-Modifizierer vorhanden ist, wird die Methode als nicht virtuelle Methode bezeichnet.

Die Implementierung einer nicht virtuellen Methode ist unveränderlich, d. h., sie bleibt gleich, unabhängig davon, ob die Methode in einer Instanz der Klasse, in der sie deklariert wird, oder in einer Instanz einer abgeleiteten Klasse aufgerufen wird. Im Gegensatz dazu können bei der Implementierung einer virtuellen Methode abgeleitete Klassen Vorrang haben. Der Prozess, bei dem ein anderes Element vor der Implementierung einer vererbten virtuellen Methode Vorrang hat, wird als Überschreiben der Methode bezeichnet (§10.6.4).

In einem Aufruf einer virtuellen Methode legt der Laufzeittyp der Instanz, für die der Aufruf ausgeführt wird, die tatsächlich aufzurufende Methodenimplementierung fest. In einem Aufruf für eine nicht virtuelle Methode ist der Kompilierungstyp der Instanz der entscheidende Faktor. Wenn also eine Methode mit der Bezeichnung N mit einer Argumentenliste A in einer Instanz mit einem Kompilierungstyp C und einem Laufzeittyp R aufgerufen wird (wobei R entweder C oder eine von C abgeleitete Klasse ist), wird der Aufruf wie folgt verarbeitet:

* Zunächst wird auf C, N und A die Überladungsauflösung angewendet, um eine bestimmte Methode M aus den Methoden auszuwählen, die in C deklariert und von C geerbt wurden. Dies wird in §7.6.5.1 beschrieben.
* Wenn es sich bei M um eine nicht virtuelle Methode handelt, wird anschließend M aufgerufen.
* Andernfalls ist M eine virtuelle Methode, und es wird die am meisten abgeleitete Implementierung von M bezüglich R aufgerufen.

Für jede virtuelle Methode, die in einer Klasse deklariert oder von einer Klasse geerbt wird, ist eine am meisten abgeleitete Implementierung der Methode bezüglich dieser Klasse vorhanden. Die am meisten abgeleitete Implementierung einer virtuellen Methode M bezüglich einer Klasse R wird folgendermaßen bestimmt:

* Wenn R die einleitende virtual-Deklaration von M enthält, handelt es sich hierbei um die am meisten abgeleitete Implementierung von M.
* Wenn R dagegen override von M enthält, dann ist dies die am meisten abgeleitete Implementierung von M.
* Andernfalls entspricht die am meisten abgeleitete Implementierung von M in Bezug auf R der am meisten abgeleiteten Implementierung von M in Bezug auf die direkte Basisklasse von R.

Anhand des folgenden Beispiels werden die Unterschiede zwischen virtuellen und nicht virtuellen Methoden dargestellt:

using System;

class A  
{  
 public void F() { Console.WriteLine("A.F"); }

public virtual void G() { Console.WriteLine("A.G"); }  
}

class B: A  
{  
 new public void F() { Console.WriteLine("B.F"); }

public override void G() { Console.WriteLine("B.G"); }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 A a = b;  
 a.F();  
 b.F();  
 a.G();  
 b.G();  
 }  
}

In diesem Beispiel führt A eine nicht virtuelle Methode F und eine virtuelle Methode G ein. Die Klasse B führt eine neue nicht virtuelle Methode F ein, wodurch die geerbte Methode F verdeckt und die geerbte Methode G ebenfalls überschrieben wird. Das Beispiel erzeugt folgende Ausgabe:

A.F  
B.F  
B.G  
B.G

Durch die Anweisung a.G() wird nicht A.G, sondern B.G aufgerufen. Das liegt daran, dass die tatsächlich aufzurufende Methodenimplementierung nicht anhand des Kompilierzeittyps der Instanz (d. h. B), sondern anhand des Laufzeittyps der Instanz (d. h. A) festgelegt wird.

Da Methoden berechtigt sind, geerbte Methoden zu verdecken, kann eine Klasse mehrere virtuelle Methoden mit derselben Signatur enthalten. Dadurch entstehen keine Mehrdeutigkeiten, da mit Ausnahme der am meisten abgeleiteten Methode alle Methoden ausgeblendet sind. In dem Beispiel

using System;

class A  
{  
 public virtual void F() { Console.WriteLine("A.F"); }  
}

class B: A  
{  
 public override void F() { Console.WriteLine("B.F"); }  
}

class C: B  
{  
 new public virtual void F() { Console.WriteLine("C.F"); }  
}

class D: C  
{  
 public override void F() { Console.WriteLine("D.F"); }  
}

class Test  
{  
 static void Main() {  
 D d = new D();  
 A a = d;  
 B b = d;  
 C c = d;  
 a.F();  
 b.F();  
 c.F();  
 d.F();  
 }  
}

enthalten die C-Klasse und die D-Klasse zwei virtuelle Methoden mit derselben Signatur. Dabei wird eine virtuelle Methode von A und die andere virtuelle Methode von C eingeführt. Durch die von C eingeführte Methode wird die von A geerbte Methode verdeckt. Somit überschreibt die override-Deklaration in D die von C eingeführte Methode, und D kann die von A eingeführte Methode nicht überschreiben. Das Beispiel erzeugt folgende Ausgabe:

B.F  
B.F  
D.F  
D.F

Die verdeckte virtuelle Methode kann aufgerufen werden, indem auf eine Instanz von D über einen geringer abgeleiteten Typ zugegriffen wird, in dem die Methode nicht verdeckt ist.

### Überschreibungsmethoden

Wenn eine Instanzmethodendeklaration einen override-Modifizierer enthält, wird die Methode als Überschreibungsmethode bezeichnet. Eine Überschreibungsmethode überschreibt eine geerbte virtuelle Methode mit derselben Signatur. Während eine virtuelle Methodendeklaration eine neue Methode einführt, spezialisiert eine Überschreibungsmethodendeklaration eine vorhandene vererbte virtuelle Methode, indem eine neue Implementierung dieser Methode zur Verfügung gestellt wird.

Die Methode, die durch eine override-Deklaration überschrieben wurde, wird als überschriebene Basismethode bezeichnet. Bei einer Überschreibungsmethode M, die in der C-Klasse deklariert ist, wird die überschriebene Basismethode ermittelt, indem alle Basisklassentypen von C, beginnend mit dem direkten Basisklassentyp von C und fortlaufend mit den nachfolgenden direkten Basisklassentypen, untersucht werden, bis in einem bestimmten Basisklassentyp mindestens eine zugreifbare Methode gefunden wird, die über dieselbe Signatur wie M verfügt, nachdem die Typargumente ersetzt wurden. Für die Suche der überschriebenen Basismethode wird die Methode als verfügbar betrachtet, wenn sie public, protected oder protected internal ist oder wenn sie internal und in demselben Programm wie C deklariert ist.

Wenn nicht alle der folgenden Bedingungen für eine override-Deklaration zutreffen, tritt während der Kompilierung ein Fehler auf:

* Eine überschriebene Basismethode kann so wie oben beschrieben gefunden werden.
* Es gibt genau eine solche überschriebene Basismethode. Diese Einschränkung gilt nur, wenn der Basisklassentyp ein konstruierter Typ ist, in dem durch Ersetzen von Typargumenten die Signaturen von zwei Methoden identisch werden.
* Bei der überschriebenen Basismethode handelt es sich um eine virtual-Methode, eine abstract-Methode oder eine override-Methode. Das bedeutet, dass die überschriebene Basismethode weder statisch noch nicht virtuell sein kann.
* Bei der überschriebenen Basismethode handelt es sich nicht um eine sealed-Methode.
* Die override-Methode und die überschriebene Basismethode haben denselben Rückgabetyp.
* Die override-Deklaration und die überschriebene Basismethode haben denselben deklarierten Zugriff. Das bedeutet, dass der Zugriff der virtual-Methode durch eine override-Deklaration nicht geändert werden kann. Wenn jedoch die überschriebene Basismethode als protected internal und in einer anderen Assembly deklariert ist, als die Assembly mit der override-Methode, dann muss die override-Methode mit protected-Zugriff deklariert werden.
* Die override-Überschreibung gibt keine type-parameter-constraints-clauses an. Stattdessen werden die Einschränkungen von der überschriebenen Basismethode geerbt. Beachten Sie, dass Einschränkungen, die Typparameter in der überschriebenen Methode sind, in der vererbten Einschränkung durch Typargumente ersetzt werden können. Dies kann zu Einschränkungen führen, die nicht gültig sind, wenn sie explizit angegeben werden, z. B. Werttypen oder versiegelte Typen.

Im folgenden Beispiel werden die bei generischen Klassen gültigen Regeln für das Überschreiben veranschaulicht:

abstract class C<T>  
{  
 public virtual T F() {...}

public virtual C<T> G() {...}

public virtual void H(C<T> x) {...}  
}

class D: C<string>  
{  
 public override string F() {...} // Ok

public override C<string> G() {...} // Ok

public override void H(C<T> x) {...} // Error, should be C<string>  
}

class E<T,U>: C<U>  
{  
 public override U F() {...} // Ok

public override C<U> G() {...} // Ok

public override void H(C<T> x) {...} // Error, should be C<U>  
}

Eine override-Deklaration kann auf die überschriebene Basismethode unter Verwendung eines base-access (§7.6.8) zugreifen. In dem Beispiel

class A  
{  
 int x;

public virtual void PrintFields() {  
 Console.WriteLine("x = {0}", x);  
 }  
}

class B: A  
{  
 int y;

public override void PrintFields() {  
 base.PrintFields();  
 Console.WriteLine("y = {0}", y);  
 }  
}

ruft der base.PrintFields()-Aufruf in B die in A deklarierte PrintFields-Methode auf. Ein base-access deaktiviert den virtuellen Aufrufmechanismus und behandelt die Basismethode einfach wie eine nicht virtuelle Methode. Wenn der Aufruf in B in der Form ((A)this).PrintFields() geschrieben wird, wird die unter B deklarierte PrintFields-Methode aufgerufen, und nicht die unter A deklarierte Methode, da PrintFields virtuell und B der Laufzeittyp von ((A)this) ist.

Eine Methode kann eine andere Methode nur durch Einfügen eines override-Modifizierers überschreiben. In allen anderen Fällen verdeckt eine Methode, die dieselbe Signatur wie eine geerbte Methode aufweist, einfach die geerbte Methode. In dem Beispiel

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 public virtual void F() {} // Warning, hiding inherited F()  
}

enthält die F-Methode in B keinen override-Modifizierer und überschreibt daher die F-Methode in A nicht. Stattdessen verdeckt die F-Methode in B die Methode in A, und es wird eine Warnung ausgegeben, weil die Deklaration keinen new-Modifizierer enthält.

In dem Beispiel

class A  
{  
 public virtual void F() {}  
}

class B: A  
{  
 new private void F() {} // Hides A.F within body of B  
}

class C: B  
{  
 public override void F() {} // Ok, overrides A.F  
}

verdeckt die F-Methode in B die virtuelle, von A geerbte F-Methode. Da die neue F-Methode in B über privaten Zugriff verfügt, umfasst deren Gültigkeitsbereich nur den Klassentext von B und erstreckt sich nicht auf C. Die Deklaration von F in C kann daher die von A geerbte F überschreiben.

### Versiegelte Methoden

Wenn eine Instanzmethodendeklaration einen sealed-Modifizierer enthält, wird die Methode als versiegelte Methode bezeichnet. Wenn eine Instanzmethodendeklaration den sealed-Modifizierer enthält, muss zusätzlich der override-Modifizierer vorhanden sein. Durch die Verwendung des sealed-Modifizierers wird das erneute Überschreiben der Methode durch eine abgeleitete Klasse verhindert.

In dem Beispiel

using System;

class A  
{  
 public virtual void F() {  
 Console.WriteLine("A.F");  
 }

public virtual void G() {  
 Console.WriteLine("A.G");  
 }  
}

class B: A  
{  
 sealed override public void F() {  
 Console.WriteLine("B.F");  
 }

override public void G() {  
 Console.WriteLine("B.G");  
 }   
}

class C: B  
{  
 override public void G() {  
 Console.WriteLine("C.G");  
 }   
}

stellt die Klasse B zwei override-Methoden zur Verfügung: eine F-Methode, die über den sealed-Modifizierer verfügt, und eine G-Methode, die nicht über diesen Modifizierer verfügt. Da B den sealed-modifier verwendet, wird das weitere Überschreiben von F durch C verhindert.

### Abstrakte Methoden

Wenn eine Instanzmethodendeklaration einen abstract-Modifizierer enthält, wird die Methode als abstrakte Methode bezeichnet. Obwohl eine abstrakte Methode implizit auch eine virtuelle Methode ist, kann sie keinen virtual-Modifizierer enthalten.

Eine abstrakte Methodendeklaration führt eine neue virtuelle Methode ein, stellt jedoch keine Implementierung dieser Methode zur Verfügung. Stattdessen sind nicht abstrakte, abgeleitete Klassen erforderlich, um die Implementierung durch Überschreiben der Methode zu ermöglichen. Da eine abstrakte Methode keine Implementierung zur Verfügung stellt, besteht der method-body einer abstrakten Methode einfach aus einem Semikolon.

Abstrakte Methodendeklarationen sind nur in abstrakten Klassen zulässig (§10.1.1.1).

In dem Beispiel

public abstract class Shape  
{  
 public abstract void Paint(Graphics g, Rectangle r);  
}

public class Ellipse: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawEllipse(r);  
 }  
}

public class Box: Shape  
{  
 public override void Paint(Graphics g, Rectangle r) {  
 g.DrawRect(r);  
 }  
}

definiert die Shape-Klasse den abstrakten Begriff eines geometrischen Formobjekts, das sich selbst zeichnen kann. Die Paint-Methode ist abstrakt, weil keine bedeutungsvolle Standardimplementierung erfolgt. Bei der Ellipse-Klasse und der Box-Klasse handelt es sich um konkrete Shape-Implementierungen. Da diese Klassen nicht abstrakt sind, sind sie zum Überschreiben der Paint-Methode erforderlich und stellen die tatsächliche Implementierung zur Verfügung.

Wenn in einem base-access (§7.6.8) auf eine abstrakte Methode verwiesen wird, entsteht ein Kompilierungsfehler. In dem Beispiel

abstract class A  
{  
 public abstract void F();  
}

class B: A  
{  
 public override void F() {  
 base.F(); // Error, base.F is abstract  
 }  
}

wird für den base.F()-Aufruf während der Kompilierung ein Fehler ausgegeben, weil dieser auf eine abstrakte Methode verweist.

Abstrakte Methodendeklarationen können zum Überschreiben einer virtuellen Methode verwendet werden. Dadurch kann eine abstrakte Klasse die erneute Implementierung der Methode in abgeleitete Klassen erzwingen. Ferner ist die ursprüngliche Implementierung der Methode nicht mehr verfügbar. In dem Beispiel

using System;

class A  
{  
 public virtual void F() {  
 Console.WriteLine("A.F");  
 }  
}

abstract class B: A  
{  
 public abstract override void F();  
}

class C: B  
{  
 public override void F() {  
 Console.WriteLine("C.F");  
 }  
}

deklariert Klasse A eine virtuelle Methode, Klasse B überschreibt diese Methode mit einer abstrakten Methode, und Klasse C überschreibt die abstrakte Methode, um die eigene Implementierung zur Verfügung zu stellen.

### Externe Methoden

Wenn eine Methodendeklaration einen extern-Modifizierer enthält, wird die Methode als externe Methode bezeichnet. Externe Methoden werden extern implementiert, wobei meist eine andere Sprache als C# verwendet wird. Da eine externe Methodendeklaration keine Implementierung zur Verfügung stellt, besteht der method-body einer externen Methode einfach aus einem Semikolon. Eine externe Methode darf nicht generisch sein.

Der extern-Modifizierer wird in der Regel zusammen mit einem DllImport-Attribut verwendet (§17.5.1), sodass externe Methoden von DLLs (Dynamic Link Libraries) implementiert werden können. Die Ausführungsumgebung unterstützt möglicherweise andere Mechanismen, sodass die Implementierungen von externen Methoden zur Verfügung gestellt werden können.

Wenn eine externe Methode ein DllImport-Attribut enthält, muss die Methodendeklaration auch einen static-Modifizierer enthalten. Anhand des folgenden Beispiels wird dargestellt, wie der extern-Modifizierer und das DllImport-Attribut verwendet werden:

using System.Text;  
using System.Security.Permissions;  
using System.Runtime.InteropServices;

class Path  
{  
 [DllImport("kernel32", SetLastError=true)]  
 static extern bool CreateDirectory(string name, SecurityAttribute sa);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool RemoveDirectory(string name);

[DllImport("kernel32", SetLastError=true)]  
 static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

[DllImport("kernel32", SetLastError=true)]  
 static extern bool SetCurrentDirectory(string name);  
}

### Partielle Methoden

Wenn eine Methodendeklaration einen partial-Modifizierer enthält, wird die Methode als partielle Methode bezeichnet. Partielle Methoden können nur als Member partieller Typen (§10.2) deklariert werden und unterliegen einigen Einschränkungen. Einzelheiten zu partiellen Methoden finden Sie in §10.2.7.

### Erweiterungsmethoden

Wenn der erste Parameter einer Methodendeklaration den this-Modifizierer enthält, wird die Methode als Erweiterungsmethode bezeichnet. Erweiterungsmethoden können ausschließlich in nicht generischen und nicht geschachtelten statischen Klassen deklariert werden. Der erste Parameter einer Erweiterungsmethode darf lediglich den this-Modifizierer enthalten, und der Parametertyp darf kein Zeigertyp sein.

Im Folgenden finden Sie ein Beispiel für eine statische Klasse, die zwei Erweiterungsmethoden deklariert:

public static class Extensions  
{  
 public static int ToInt32(this string s) {  
 return Int32.Parse(s);  
 }

public static T[] Slice<T>(this T[] source, int index, int count) {  
 if (index < 0 || count < 0 || source.Length – index < count)  
 throw new ArgumentException();  
 T[] result = new T[count];  
 Array.Copy(source, index, result, 0, count);  
 return result;  
 }  
}

Erweiterungsmethoden sind reguläre statische Methoden. Darüber hinaus kann eine Erweiterungsmethode, wenn ihre einschließende statische Klasse in ihrem Gültigkeitsbereich liegt, mithilfe der Syntax für den Aufruf von Instanzmethoden (§7.6.5.2) aufgerufen werden, wobei der Empfängerausdruck als erstes Argument verwendet wird.

Im folgenden Programm werden die oben deklarierten Erweiterungsmethoden verwendet:

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in strings.Slice(1, 2)) {  
 Console.WriteLine(s.ToInt32());  
 }  
 }  
}

Die Slice-Methode ist in string[] verfügbar, und die ToInt32-Methode ist in string verfügbar, da beide als Erweiterungsmethoden deklariert wurden. Die Bedeutung des Programms ist mit folgendem Beispiel identisch, bei dem herkömmliche Methodenaufrufe verwendet werden:

static class Program  
{  
 static void Main() {  
 string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in Extensions.Slice(strings, 1, 2)) {  
 Console.WriteLine(Extensions.ToInt32(s));  
 }  
 }  
}

### Methodentext

Der method-body einer Methodendeklaration besteht entweder aus einem block oder einem Semikolon.

Da abstrakte und externe Methodendeklarationen keine Methodenimplementierung zur Verfügung stellen, bestehen deren Methodentexte einfach aus einem Semikolon. Bei allen anderen Methoden besteht der Methodentext aus einem Block (§8.2), der die Anweisungen enthält, die beim Aufrufen der Methode ausgeführt werden.

Der Ergebnistyp einer Methode ist void, wenn der Rückgabetyp void ist oder wenn die Methode "async" und der Rückgabetyp System.Threading.Tasks.Task ist. Andernfalls entspricht der Ergebnistyp einer nicht asynchronen Methode dem Rückgabetyp, und der Ergebnistyp einer asynchronen Methode mit dem Rückgabetyp System.Threading.Tasks.Task<*T*> ist *T*.

Wenn der Ergebnistyp einer Methode void lautet, sind im Methodentext für die Angabe eines Ausdrucks keine return-Anweisungen (§8.9.4) zulässig. Wenn die Ausführung des Methodentexts einer void-Methode normal abgeschlossen wird (d. h., die Steuerung geht über das Ende des Methodentexts hinaus), kehrt die Methode einfach zum aktuellen Aufrufer zurück.

Wenn der Ergebnistyp einer Methode nicht void lautet, müssen alle return-Anweisungen in dem Methodentext einen Ausdruck angeben, der implizit in den Ergebnistyp konvertierbar ist. Der Endpunkt des Methodentexts einer Methode, die einen Wert zurückgibt, darf nicht erreichbar sein. Das bedeutet, dass in einer Methode, die einen Wert zurückgibt, die Steuerung nicht über das Ende des Methodentexts hinausgehen darf.

In dem Beispiel

class A  
{  
 public int F() {} // Error, return value required

public int G() {  
 return 1;  
 }

public int H(bool b) {  
 if (b) {  
 return 1;  
 }  
 else {  
 return 0;  
 }  
 }  
}

führt die F-Methode, die einen Wert zurückgibt, zu einem Kompilierungsfehler, da die Steuerung über das Ende des Methodentexts hinausgehen kann. Die G-Methode und die H-Methode sind korrekt, da alle möglichen Ausführungspfade in einer Rückgabeanweisung enden, die einen Rückgabewert angibt.

### Methodenüberladungen

Die Auflösungsregeln für die Methodenüberladung werden in §7.5.2 erläutert.

## Eigenschaften

Eine Eigenschaft ist ein Member, der Zugriff auf ein Merkmal eines Objekts oder einer Klasse bietet. Beispiele für Eigenschaften sind die Länge einer Zeichenfolge, die Größe einer Schriftart, die Beschriftung eines Fensters, der Name eines Kunden usw. Eigenschaften sind die natürliche Erweiterung von Feldern. Beide sind bezeichnete Member mit verknüpften Typen. Auch ist die Syntax für den Zugriff auf Felder und Eigenschaften identisch. Im Unterschied zu Feldern kennzeichnen Eigenschaften jedoch keine Speicherorte. Stattdessen besitzen Eigenschaften Accessoren, die die Anweisungen angeben, die beim Lesen bzw. Schreiben ihrer Werte auszuführen sind. Eigenschaften stellen demnach einen Mechanismus zur Verfügung, durch den Aktionen mit dem Lesen und Schreiben der Attribute eines Objekts verknüpft werden. Außerdem ermöglichen sie das Berechnen solcher Attribute.

Eigenschaften werden mithilfe von property-declarations deklariert:

property-declaration:  
attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:  
property-modifier  
property-modifiers property-modifier

property-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

member-name:  
identifier  
interface-type . identifier

Eine property-declaration kann eine Gruppe von attributes (§17) und eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5), dem new (§10.3.4)-, static (§10.6.2)-, virtual (§10.6.3)-, override (§10.6.4)-, sealed (§10.6.5)-, abstract (§10.6.6)- und extern (§10.6.7)-Modifizierer enthalten.

Eigenschaftendeklarationen unterliegen im Hinblick auf zulässige Modifiziererkombinationen denselben Regeln wie Methodendeklarationen (§10.6).

Der type einer Eigenschaftendeklaration gibt den Typ der Eigenschaft an, der von der Deklaration eingeführt wird. Der member-name gibt den Namen der Eigenschaft an. Sofern die Eigenschaft keine explizite Implementierung eines Schnittstellenmembers ist, handelt es sich bei dem member-name einfach um einen identifier. Damit ein Schnittstellenmember explizit implementiert werden kann (§13.4.1), besteht der member-name aus einem interface-type gefolgt von einem "." und einem identifier.

Der type einer Eigenschaft muss mindestens genauso verfügbar sein wie die Eigenschaft selbst (§3.5.4).

Die accessor-declarations, die in ein "{"-Token und ein "}"-Token eingeschlossen werden müssen, deklarieren die Accessoren (§10.7.2) der Eigenschaft. Die Accessoren geben die ausführbaren Anweisungen an, die mit dem Lesen und Schreiben der Eigenschaft verknüpft sind.

Obwohl die Syntax für den Zugriff auf eine Eigenschaft dieselbe ist wie die für ein Feld, wird eine Eigenschaft nicht als Variable klassifiziert. Aus diesem Grund kann eine Eigenschaft nicht als ref-Argument oder out-Argument übergeben werden.

Wenn eine Eigenschaftendeklaration einen extern-Modifizierer enthält, wird die Eigenschaft als externe Eigenschaft bezeichnet. Da eine externe Eigenschaftendeklaration keine tatsächliche Implementierung zur Verfügung stellt, bestehen alle accessor-declarations einer Eigenschaft aus einem Semikolon.

### Statische Eigenschaften und Instanzeigenschaften

Wenn eine Eigenschaftendeklaration einen static-Modifizierer enthält, wird die Eigenschaft als statische Eigenschaft bezeichnet. Wenn kein static-Modifizierer vorhanden ist, wird die Eigenschaft als Instanzeigenschaft bezeichnet.

Für statische Eigenschaften gilt, dass sie nicht mit einer bestimmten Instanz verknüpft sind und in den Accessoren einer statischen Eigenschaft nicht auf this verwiesen werden darf, da andernfalls ein Kompilierungsfehler auftritt.

Eine Instanzeigenschaft ist mit einer bestimmten Instanz einer Klasse verknüpft, und auf diese Instanz kann so wie auf this (§7.6.7) in den Accessoren der Eigenschaft zugegriffen werden.

Wenn in einem member-access (§7.6.4) der Form E.M auf eine Eigenschaft verwiesen wird und M eine statische Eigenschaft ist, muss E einen Typ mit einem Feld M bezeichnen. Wenn M eine Instanzeigenschaft ist, muss E eine Instanz eines Typs bezeichnen, der M enthält.

Die Unterschiede zwischen statischen Membern und Instanzmembern werden in §10.3.7 näher erläutert.

### Accessoren

Die accessor-declarations einer Eigenschaft geben die ausführbaren Anweisungen an, die mit dem Lesen und Schreiben der Eigenschaft verknüpft sind.

accessor-declarations:  
get-accessor-declaration set-accessor-declarationopt  
set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:  
attributesopt accessor-modifieropt  get accessor-body

set-accessor-declaration:  
attributesopt accessor-modifieropt set accessor-body

accessor-modifier:  
protected  
internal  
private  
protected internal  
internal protected

accessor-body:  
block  
;

Die Accessordeklarationen bestehen aus einer get-accessor-declaration und/oder einer set-accessor-declaration. Eine Accessordeklaration besteht aus dem Token get oder set, gefolgt von einem optionalen accessor-modifier und einem accessor-body.

Die Verwendung von accessor-modifiers wird durch die folgenden Einschränkungen gesteuert:

* Ein accessor-modifier darf nicht in der Implementierung einer Schnittstelle oder eines expliziten Schnittstellenmembers verwendet werden.
* Für eine Eigenschaft oder einen Indexer ohne override-Modifizierer ist ein accessor-modifier nur dann zulässig, wenn die Eigenschaft bzw. der Indexer sowohl über einen get-Accessor als auch über einen set-Accessor verfügt, und in diesem Fall ist er nur für einen dieser Accessoren zulässig.
* Für eine Eigenschaft oder einen Indexer mit einem override-Modifizierer muss ein Accessor dem accessor-modifier (falls vorhanden) des überschriebenen Accessors entsprechen.
* Der accessor-modifier muss einen Zugriff deklarieren, der bei weitem restriktiver als der deklarierte Zugriff der Eigenschaft bzw. des Indexers selbst ist. Dies bedeutet Folgendes:
* Wenn für die Eigenschaft oder den Indexer der Zugriff public deklariert ist, kann der accessor-modifier den Wert protected internal, internal, protected oder private annehmen.
* Wenn für die Eigenschaft oder den Indexer der Zugriff protected internal deklariert ist, kann der accessor-modifier den Wert internal, protected oder private annehmen.
* Wenn für die Eigenschaft oder den Indexer der Zugriff internal oder protected deklariert ist, muss der accessor-modifier private sein.
* Wenn für die Eigenschaft oder den Indexer der Zugriff private deklariert ist, kann kein accessor-modifier verwendet werden.

Bei der abstract-Eigenschaft und der extern-Eigenschaft besteht der accessor-body des angegebenen Accessors einfach aus einem Semikolon. Eine nicht abstrakte, nicht externe Eigenschaft ist eine automatisch implementierte Eigenschaft, wenn sowohl der get-Accessor als auch der set-Accessor angegeben wird und bei beiden der Text aus einem Semikolon besteht (§10.7.3). Für Accessoren einer anderen nicht abstrakten und nicht externen Eigenschaft ist der accessor-body ein block, in dem die Anweisungen angegeben werden, die beim Aufruf des entsprechenden Accessors ausgeführt werden.

Ein get-Accessor entspricht einer parameterlosen Methode mit einem Rückgabewert vom Typ der Eigenschaft. Wenn auf eine Eigenschaft in einem Ausdruck verwiesen wird, wird der get-Accessor der Eigenschaft aufgerufen, um den Wert der Eigenschaft zu berechnen (§7.1.1). Dies gilt nicht, wenn der Accessor das Ziel einer Zuweisung ist. Der Text eines get-Accessors muss den Regeln für Methoden entsprechen, die Werte zurückgeben (siehe §10.6.10). Insbesondere müssen alle return-Anweisungen im Text eines get-Accessors einen Ausdruck angeben, der implizit in den Eigenschaftentyp konvertierbar ist. Außerdem darf der Endpunkt eines get-Accessors nicht erreichbar sein.

Ein set-Accessor entspricht einer Methode mit einem einzigen Werteparameter des Eigenschaftentyps und einem void-Rückgabetyp. Der implizite Parameter eines set-Accessors lautet immer value. Wenn auf eine Eigenschaft als Ziel einer Zuweisung (§7.17) oder als der Operand von ++ oder -- (§7.6.9 und §7.7.5) verwiesen wird, wird der set-Accessor mit einem Argument aufgerufen (dessen Wert entweder dem auf der rechten Seite der Zuweisung oder dem Operanden des ++-Operators bzw. des ---Operators entspricht), das den neuen Wert zur Verfügung stellt (§7.17.1). Der Text eines set-Accessors muss den Regeln für void-Methoden entsprechen (siehe §10.6.10). Insbesondere dürfen keine return-Anweisungen im Text des set-Accessors zum Angeben eines Ausdrucks verwendet werden. Da ein set-Accessor implizit einen Parameter mit der Bezeichnung value enthält, tritt ein Kompilierungsfehler auf, wenn eine lokale Variable oder Konstantendeklaration in einem set-Accessor diesen Namen aufweist.

Eigenschaften werden in Abhängigkeit vom Vorhandensein des get-Accessors und des set-Accessors folgendermaßen klassifiziert:

* Eine Eigenschaft, die sowohl einen get-Accessor als auch einen set-Accessor enthält, wird als Lesen-Schreiben-Eigenschaft bezeichnet.
* Eine Eigenschaft, die nur einen get-Accessor aufweist, wird als schreibgeschützte Eigenschaft bezeichnet. Während der Kompilierung tritt ein Fehler auf, wenn eine Nur-Lesen-Eigenschaft das Ziel einer Zuweisung ist.
* Eine Eigenschaft, die nur einen set-Accessor aufweist, wird als Nur-Schreiben-Eigenschaft bezeichnet. In einem Ausdruck darf lediglich auf Nur-Schreiben-Eigenschaften verwiesen werden, wenn es sich um das Ziel einer Zuweisung handelt, andernfalls tritt ein Kompilierungsfehler auf.

In dem Beispiel

public class Button: Control  
{  
 private string caption;

public string Caption {  
 get {  
 return caption;  
 }  
 set {  
 if (caption != value) {  
 caption = value;  
 Repaint();  
 }  
 }  
 }

public override void Paint(Graphics g, Rectangle r) {  
 // Painting code goes here  
 }  
}

deklariert das Button-Steuerelement eine öffentliche Caption-Eigenschaft. Der get-Accessor der Caption-Eigenschaft gibt die Zeichenfolge zurück, die im privaten caption-Feld gespeichert ist. Der set-Accessor überprüft, ob sich der neue Wert von dem aktuellen Wert unterscheidet. Wenn dies der Fall ist, wird der neue Wert gespeichert und das Steuerelement neu aufgebaut. Für Eigenschaften wird häufig das oben dargestellte Muster verwendet: Der get-Accessor gibt einfach einen Wert zurück, der in einem privaten Feld gespeichert ist. Der set-Accessor ändert dieses private Feld und führt anschließend alle zusätzlichen Aktionen aus, die zum vollständigen Aktualisieren des Objektzustands erforderlich sind.

Ausgehend von der oben genannten Button-Klasse kann die Caption-Eigenschaft wie im folgenden Beispiel verwendet werden:

Button okButton = new Button();  
okButton.Caption = "OK"; // Invokes set accessor  
string s = okButton.Caption; // Invokes get accessor

Hier wird einerseits der set-Accessor durch Zuweisen eines Werts zur Eigenschaft und andererseits der get-Accessor durch Verweisen auf die Eigenschaft in einem Ausdruck aufgerufen.

Bei dem get-Accessor und dem set-Accessor einer Eigenschaft handelt es nicht um unterschiedliche Member. Die Accessoren einer Eigenschaft können nicht getrennt deklariert werden. Die beiden Accessoren einer Lesen-Schreiben-Eigenschaft können nicht unterschiedlichen Zugriff haben. In dem Beispiel

class A  
{  
 private string name;

public string Name { // Error, duplicate member name  
 get { return name; }  
 }

public string Name { // Error, duplicate member name  
 set { name = value; }  
 }  
}

wird keine einzelne Lesen-Schreiben-Eigenschaft deklariert. Vielmehr werden zwei Eigenschaften mit demselben Namen deklariert: eine Nur-Lesen-Eigenschaft und eine Nur-Schreiben-Eigenschaft. Da zwei Member, die in derselben Klasse deklariert werden, nicht denselben Namen haben können, führt das Beispiel während der Kompilierung zu einem Fehler.

Wenn eine abgeleitete Klasse eine Eigenschaft mit demselben Namen wie in der geerbten Eigenschaft deklariert, verdeckt die abgeleitete Eigenschaft die geerbte Eigenschaft sowohl bezüglich des Lesens als auch des Schreibens. In dem Beispiel

class A  
{  
 public int P {  
 set {...}  
 }  
}

class B: A  
{  
 new public int P {  
 get {...}  
 }  
}

verdeckt die P-Eigenschaft in B die P-Eigenschaft in A sowohl bezüglich des Lesens als auch des Schreibens. Folglich wird bei den Anweisungen

B b = new B();  
b.P = 1; // Error, B.P is read-only  
((A)b).P = 1; // Ok, reference to A.P

durch die Zuweisung zu b.P ein Kompilierungsfehler ausgegeben, da die Nur-Lesen-Eigenschaft P in B die Nur-Schreiben-Eigenschaft P in A verdeckt. Der Zugriff auf die verdeckte P-Eigenschaft kann jedoch durch eine Umwandlung realisiert werden.

Im Gegensatz zu öffentlichen Feldern sind in Eigenschaften der interne Status von Objekten und deren öffentliche Schnittstelle getrennt. Beispiel:

class Label  
{  
 private int x, y;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.x = x;  
 this.y = y;  
 this.caption = caption;  
 }

public int X {  
 get { return x; }  
 }

public int Y {  
 get { return y; }  
 }

public Point Location {  
 get { return new Point(x, y); }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Die Label-Klasse verwendet hier zum Speichern ihrer Position zwei int-Felder: x und y. Die Position wird sowohl als X-Eigenschaft als auch als Y-Eigenschaft und als Location-Eigenschaft vom Typ Point öffentlich verfügbar gemacht. Wenn es in einer zukünftigen Version von Label sinnvoller ist, die Position intern als Point zu speichern, kann die Änderung ohne Auswirkungen auf die öffentliche Schnittstelle der Klasse vorgenommen werden:

class Label  
{  
 private Point location;  
 private string caption;

public Label(int x, int y, string caption) {  
 this.location = new Point(x, y);  
 this.caption = caption;  
 }

public int X {  
 get { return location.x; }  
 }

public int Y {  
 get { return location.y; }  
 }

public Point Location {  
 get { return location; }  
 }

public string Caption {  
 get { return caption; }  
 }  
}

Wenn es sich bei x und y stattdessen um public readonly-Felder handelt, kann diese Änderung in der Label-Klasse nicht vorgenommen werden.

Es ist nicht unbedingt weniger effizient, den Status über Eigenschaften verfügbar zu machen, als die Felder direkt offenzulegen. Wenn eine Eigenschaft nicht virtuell ist und nur wenig Code enthält, kann die Ausführungsumgebung Aufrufe an Accessoren durch den tatsächlichen Code des Accessors ersetzen. Dieser Prozess wird als Inlining bezeichnet. Durch diesen Prozess wird der Zugriff auf Eigenschaften ebenso effizient wie der Zugriff auf Felder, wobei die erhöhte Flexibilität von Eigenschaften erhalten bleibt.

Da das Aufrufen eines get-Accessors konzeptionell mit dem Lesen des Werts eines Felds übereinstimmt, gilt es als schlechter Programmierstil, wenn ein get-Accessor sichtbare Nebeneffekte hat. In dem Beispiel

class Counter  
{  
 private int next;

public int Next {  
 get { return next++; }  
 }  
}

hängt der Wert der Next-Eigenschaft von der Anzahl der bisherigen Zugriffe auf die Eigenschaft ab. Somit erzeugt der Zugriff auf die Eigenschaft einen sichtbaren Nebeneffekt, sodass die Eigenschaft stattdessen als Methode implementiert werden sollte.

Die Konvention, dass get-Accessoren keine Nebeneffekte haben sollten, bedeutet nicht, dass get-Accessoren immer so geschrieben werden müssen, dass sie einfach in Feldern gespeicherte Werte zurückgeben. get-Accessoren berechnen häufig den Wert einer Eigenschaft, indem auf mehrere Felder zugegriffen wird oder Methoden aufgerufen werden. Ein richtig entworfener get-Accessor führt jedoch keine Aktionen aus, die sichtbare Änderungen des Zustands des Objekts verursachen.

Mithilfe von Eigenschaften kann die Initialisierung einer Ressource so lange verzögert werden, bis auf die Ressource zum ersten Mal verwiesen wird. Beispiel:

using System.IO;

public class Console  
{  
 private static TextReader reader;  
 private static TextWriter writer;  
 private static TextWriter error;

public static TextReader In {  
 get {  
 if (reader == null) {  
 reader = new StreamReader(Console.OpenStandardInput());  
 }  
 return reader;  
 }  
 }

public static TextWriter Out {  
 get {  
 if (writer == null) {  
 writer = new StreamWriter(Console.OpenStandardOutput());  
 }  
 return writer;  
 }  
 }

public static TextWriter Error {  
 get {  
 if (error == null) {  
 error = new StreamWriter(Console.OpenStandardError());  
 }  
 return error;  
 }  
 }  
}

Die Console-Klasse enthält drei Eigenschaften (In, Out und Error), die jeweils die Standardeingabegeräte, Standardausgabegeräte und Standardfehlergeräte darstellen. Wenn diese Member als Eigenschaften verfügbar gemacht werden, kann die Console-Klasse deren Initialisierung so lange verzögern, bis sie tatsächlich verwendet werden. Beispiel: Nach dem ersten Verweis auf die Out-Eigenschaft wie in

Console.Out.WriteLine("hello, world");

wird das zugrunde liegende TextWriter für das Ausgabegerät erstellt. Wenn die Anwendung jedoch nicht auf die In-Eigenschaft und die Error-Eigenschaft verweist, werden keine Objekte für diese Geräte erstellt.

### Automatisch implementierte Eigenschaften

Wenn eine Eigenschaft als automatisch implementierte Eigenschaft gekennzeichnet wird, steht automatisch ein ausgeblendetes dahinter liegendes Feld für die Eigenschaft zur Verfügung, und die Accessoren werden zum Lesen und Schreiben in diesem dahinter liegenden Feld implementiert.

Beispiel:

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

entspricht der folgenden Deklaration:

public class Point {  
 private int x;  
 private int y;  
 public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Da auf das dahinter liegende Feld nicht zugegriffen werden kann, ist das Lesen und Schreiben in diesem Feld nur über die Eigenschaftenaccessoren möglich, auch innerhalb des enthaltenden Typs. Aus diesem Grund sind automatisch implementierte schreibgeschützte oder Nur-Lesen-Eigenschaften nicht sinnvoll und auch nicht zulässig. Es ist jedoch möglich, die Zugriffsebene für einzelne Accessoren unterschiedlich festzulegen. Somit kann der Effekt einer schreibgeschützten Eigenschaft mit einem privaten dahinter liegenden Feld folgendermaßen imitiert werden:

public class ReadOnlyPoint {  
 public int X { get; private set; }  
 public int Y { get; private set; }  
 public ReadOnlyPoint(int x, int y) { X = x; Y = y; }  
}

Diese Einschränkung bedeutet auch, dass definitive Zuweisungen von Strukturtypen mit automatisch implementierten Eigenschaften nur mit dem Standardkonstruktor der Struktur möglich sind, da es für das Zuweisen der Eigenschaft selbst erforderlich ist, dass die Struktur definitiv zugewiesen wird. Das bedeutet, dass benutzerdefinierte Konstruktoren den Standardkonstruktor aufrufen müssen.

### Barrierefreiheit

Wenn ein Accessor über einen accessor-modifier verfügt, wird die Zugriffsdomäne (§3.5.2) des Accessors anhand des deklarierten Zugriffs des accessor-modifier bestimmt. Wenn ein Accessor keinen accessor-modifier besitzt, wird die Zugriffsdomäne des Accessors anhand des deklarierten Zugriffs der Eigenschaft oder des Indexers bestimmt.

Das Vorhandensein eines accessor-modifier wirkt sich nie auf die Membersuche (§7.3) oder Überladungsauflösung (§7.5.3) aus. Die Modifizierer für die Eigenschaft oder den Indexer bestimmen immer, an welche Eigenschaft bzw. an welchen Indexer eine Bindung erfolgt, ungeachtet des Kontexts des Zugriffs.

Wenn eine bestimmte Eigenschaft bzw. ein bestimmter Indexer ausgewählt wurde, wird mithilfe der Zugriffsdomänen der beteiligten Accessoren jeweils bestimmt, ob die Verwendung zulässig ist:

* Wenn die Verwendung über einen Wert (§7.1.1) erfolgt, muss der get-Accessor vorhanden und zugreifbar sein.
* Wenn die Verwendung über das Ziel einer einfachen Zuweisung (§7.17.1) erfolgt, muss der set-Accessor vorhanden und zugreifbar sein.
* Wenn die Verwendung über das Ziel einer Verbundzuweisung (§7.17.2) oder das Ziel der ++- bzw. ---Operatoren (§7.5.9, §7.6.5) erfolgt, müssen sowohl der get-Accessor als auch der set-Accessor vorhanden und zugreifbar sein.

Im folgenden Beispiel wird die Eigenschaft A.Text durch die Eigenschaft B.Text verborgen, selbst in Kontexten, in denen nur der set-Accessor aufgerufen wird. Im Gegensatz dazu kann von der Klasse M nicht auf die Eigenschaft B.Count zugegriffen werden, sodass stattdessen die Eigenschaft A.Count verwendet wird, auf die zugegriffen werden kann.

class A  
{  
 public string Text {  
 get { return "hello"; }  
 set { }  
 }

public int Count {  
 get { return 5; }  
 set { }  
 }  
}

class B: A  
{  
 private string text = "goodbye";   
 private int count = 0;

new public string Text {  
 get { return text; }  
 protected set { text = value; }  
 }

new protected int Count {   
 get { return count; }  
 set { count = value; }  
 }  
}

class M  
{  
 static void Main() {  
 B b = new B();  
 b.Count = 12; // Calls A.Count set accessor  
 int i = b.Count; // Calls A.Count get accessor  
 b.Text = "howdy"; // Error, B.Text set accessor not accessible  
 string s = b.Text; // Calls B.Text get accessor  
 }  
}

Ein Accessor, mit dessen Hilfe eine Schnittstelle implementiert wird, darf keinen accessor-modifier besitzen. Wenn eine Schnittstelle nur mit einem Accessor implementiert wird, kann der andere Accessor mit einem accessor-modifier deklariert werden:

public interface I  
{  
 string Prop { get; }  
}

public class C: I  
{  
 public Prop {  
 get { return "April"; } // Must not have a modifier here  
 internal set {...} // Ok, because I.Prop has no set accessor  
 }  
}

### Virtuelle, versiegelte, überschreibungsbezogene und abstrakte Accessoren

Eine virtual-Eigenschaftendeklaration gibt an, dass die Accessoren der Eigenschaft virtuell sind. Da nicht nur ein Accessor einer Lesen-Schreiben-Eigenschaft virtuell sein kann, trifft der virtual-Modifizierer auf beide Accessoren der Lesen-Schreiben-Eigenschaft zu.

Eine abstract-Eigenschaftendeklaration gibt an, dass die Accessoren der Eigenschaft virtuell sind, stellt jedoch keine Implementierung der Accessoren zur Verfügung. Stattdessen sind nicht abstrakte abgeleitete Klassen erforderlich, um die Implementierung für die Accessoren durch Überschreiben der Eigenschaft zu ermöglichen. Da ein Accessor für eine abstrakte Eigenschaftendeklaration keine Implementierung zur Verfügung stellt, besteht dessen accessor-body einfach aus einem Semikolon.

Mit einer Eigenschaftendeklaration, die sowohl den abstract-Modifizierer als auch den override-Modifizierer enthält, wird angegeben, dass die Eigenschaft abstrakt ist und eine Basiseigenschaft überschreibt. Die Accessoren einer solchen Eigenschaft sind ebenfalls abstrakt.

Abstrakte Eigenschaftendeklarationen sind nur in abstrakten Klassen zulässig (§10.1.1.1)). Die Accessoren einer geerbten virtuellen Eigenschaft können in einer abgeleiteten Klasse durch Einschließen einer Eigenschaftendeklaration überschrieben werden, die eine override-Direktive angibt. Diese Deklaration wird als überschreibende Eigenschaftendeklaration bezeichnet. Mit einer überschriebenen Eigenschaftendeklaration wird keine neue Eigenschaft deklariert. Vielmehr werden durch eine solche Deklaration einfach die Implementierungen der Accessoren einer vorhandenen virtuellen Eigenschaft spezialisiert.

Eine überschreibende Eigenschaftendeklaration muss genau dieselben Zugriffsmodifizierer, denselben Typ und denselben Namen wie die geerbte Eigenschaft angeben. Wenn die geerbte Eigenschaft nur über einen Accessor verfügt (d. h., wenn es sich bei der geerbten Eigenschaft um eine Nur-Lesen-Eigenschaft oder eine Nur-Schreiben-Eigenschaft handelt), darf die überschreibende Eigenschaft nur diesen Accessor enthalten. Wenn die geerbte Eigenschaft beide Accessoren enthält (d. h., wenn es sich bei der geerbten Eigenschaft um eine Lesen-Schreiben-Eigenschaft handelt), kann die überschreibende Eigenschaft entweder einen einzelnen Accessor oder beide Accessoren enthalten.

Eine überschreibende Eigenschaftendeklaration kann den sealed-Modifizierer enthalten. Durch Verwenden dieses Modifizierers wird verhindert, dass die Eigenschaft durch eine abgeleitete Klasse weiter überschrieben wird. Die Accessoren einer versiegelten Eigenschaft sind ebenfalls versiegelt.

Virtuelle Accessoren, versiegelte Accessoren, Überschreibungsaccessoren und abstrakte Accessoren verhalten sich abgesehen von Unterschieden bei der Deklarationssyntax und der Aufrufsyntax genauso wie virtuelle Methoden, versiegelte Methoden, Überschreibungsmethoden und abstrakte Methoden. Insbesondere in den in §10.6.3, §10.6.4, §10.6.5 und §10.6.6 beschriebenen Regeln werden Accessoren wie die entsprechenden Methoden behandelt:

* Ein get-Accessor entspricht einer parameterlosen Methode mit einem Rückgabewert des Eigenschaftentyps und denselben Modifizierern wie die enthaltende Eigenschaft.
* Ein set-Accessor entspricht einer Methode mit einem Werteparameter des Eigenschaftentyps, einem void-Rückgabetyp und denselben Modifizierern wie die enthaltende Eigenschaft.

In dem Beispiel

abstract class A  
{  
 int y;

public virtual int X {  
 get { return 0; }  
 }

public virtual int Y {  
 get { return y; }  
 set { y = value; }  
 }

public abstract int Z { get; set; }  
}

ist X eine virtuelle Nur-Lesen-Eigenschaft, Y eine virtuelle Lesen-Schreiben-Eigenschaft und Z eine abstrakte Lesen-Schreiben-Eigenschaft. Da Z abstrakt ist, muss die enthaltende Klasse A ebenfalls als abstrakt deklariert werden.

Die unten dargestellte Klasse wurde von A abgeleitet:

class B: A  
{  
 int z;

public override int X {  
 get { return base.X + 1; }  
 }

public override int Y {  
 set { base.Y = value < 0? 0: value; }  
 }

public override int Z {  
 get { return z; }  
 set { z = value; }  
 }  
}

Die Deklarationen von X, Y und Z überschreiben hier die Eigenschaftendeklarationen. Eine Eigenschaftendeklaration stimmt mit den Zugriffsmodifizierern, dem Typ und dem Namen der entsprechenden geerbten Eigenschaft überein. Der get-Accessor von X und der set-Accessor von Y greifen mithilfe des base-Schlüsselworts auf die geerbten Accesoren zu. Die Deklaration von Z überschreibt die beiden abstrakten Accessoren, sodass keine ausstehenden abstrakten Funktionsmember in B vorhanden sind und B eine nicht abstrakte Klasse sein darf.

Wenn eine Eigenschaft als override deklariert ist, muss der überschreibende Code auf sämtliche überschriebenen Accessoren zugreifen können. Darüber hinaus muss der deklarierte Zugriff der Eigenschaft bzw. des Indexers selbst sowie der Accessoren dem des überschriebenen Members und der überschriebenen Accessoren entsprechen. Beispiel:

public class B  
{  
 public virtual int P {  
 protected set {...}  
 get {...}  
 }  
}

public class D: B  
{  
 public override int P {  
 protected set {...} // Must specify protected here  
 get {...} // Must not have a modifier here  
 }  
}

## Ereignisse

Ein Ereignis ist ein Member, der einem Objekt oder einer Klasse das Bereitstellen von Benachrichtigungen ermöglicht. Clients können ausführbaren Code für Ereignisse anfügen, indem sie Ereignishandler zur Verfügung stellen.

Ereignisse werden mithilfe von event-declarations deklariert:

event-declaration:  
attributesopt event-modifiersopt event type variable-declarators ;  
attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:  
event-modifier  
event-modifiers event-modifier

event-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

event-accessor-declarations:  
add-accessor-declaration remove-accessor-declaration  
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:  
attributesopt add block

remove-accessor-declaration:  
attributesopt remove block

Eine event-declaration kann eine Gruppe von attributes (§17) und eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5) sowie die Modifizierer new (§10.3.4), static (§10.6.2), virtual (§10.6.3), override (§10.6.4), sealed (§10.6.5), abstract (§10.6.6) und extern (§10.6.7) enthalten.

Ereignisdeklarationen unterliegen im Hinblick auf zulässige Modifiziererkombinationen denselben Regeln wie Methodendeklarationen (§10.6).

Bei dem type einer Ereignisdeklaration muss es sich um einen delegate-type handeln (§4.2), und dieser delegate-type muss mindestens genauso verfügbar sein wie das Ereignis selbst (§3.5.4).

Eine Ereignisdeklaration kann event-accessor-declarations enthalten. Sollten jedoch keine enthalten sein, stellt der Compiler diese für nicht externe, nicht abstrakte Ereignisse automatisch bereit (§10.8.1); für externe Ereignisse werden die Accessoren extern zur Verfügung gestellt.

Mit einer Ereignisdeklaration, die die event-accessor-declarations auslässt, wird für jeden vorhandenen variable-declarator ein Ereignis definiert. Die Attribute und Modifizierer gelten für alle Member, die von einer solchen event-declaration deklariert wurden.

Es tritt ein Kompilierungsfehler auf, wenn eine event-declaration sowohl den abstract-Modifizierer als auch event-accessor-declarations enthält, die durch geschweifte Klammern getrennt sind.

Wenn eine Ereignisdeklaration einen extern-Modifizierer enthält, wird das Ereignis als externes Ereignis bezeichnet. Da eine externe Ereignisdeklaration keine tatsächliche Implementierung bereitstellt, tritt ein Fehler auf, wenn sie sowohl den extern-Modifizierer als auch event-accessor-declarations enthält.

Es tritt ein Kompilierungsfehler auf, wenn ein variable-declarator einer Ereignisdeklaration mit einem abstract-Modifizierer oder einem external-Modifizierer einen variable-initializer enthält.

Ein Ereignis kann als linker Operand des +=-Operators und des -=-Operators (§7.17.3) verwendet werden. Mit diesen Operatoren können Ereignishandler an ein Ereignis angefügt bzw. daraus entfernt werden. Die Zugriffsmodifizierer des Ereignisses steuern die Kontexte, in denen solche Operationen zulässig sind.

Da += und -= die einzigen Operationen sind, die für ein Ereignis außerhalb des Typs, der das Ereignis deklariert, zulässig sind, kann externer Code zwar Handler für ein Ereignis hinzufügen und entfernen, aber in keiner Weise die zugrunde liegende Liste der Ereignishandler abrufen oder ändern.

Das Ergebnis der Operation der Form x += y oder x -= y hat den Typ void (im Gegensatz zum Typ von x mit dem Wert von x nach der Zuweisung), wenn x ein Ereignis ist und der Verweis außerhalb des Typs erfolgt, der die Deklaration von x enthält. Diese Regel lässt nicht zu, dass der zugrunde liegende Delegat eines Ereignisses durch externen Code indirekt überprüft wird.

Im folgenden Beispiel wird dargestellt, wie Ereignishandler an Instanzen der Button-Klasse angefügt werden:

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;  
}

public class LoginDialog: Form  
{  
 Button OkButton;  
 Button CancelButton;

public LoginDialog() {  
 OkButton = new Button(...);  
 OkButton.Click += new EventHandler(OkButtonClick);  
 CancelButton = new Button(...);  
 CancelButton.Click += new EventHandler(CancelButtonClick);  
 }

void OkButtonClick(object sender, EventArgs e) {  
 // Handle OkButton.Click event  
 }

void CancelButtonClick(object sender, EventArgs e) {  
 // Handle CancelButton.Click event  
 }  
}

Der Instanzkonstruktor für LoginDialog erstellt hier zwei Button-Instanzen und fügt Ereignishandler an die Click-Ereignisse an.

### Feldähnliche Ereignisse

Innerhalb des Programmtextes der Klasse oder Struktur, welche die Deklaration eines Ereignisses enthält, können bestimmte Ereignisse wie Felder verwendet werden. Wenn ein Ereignis als solches verwendet werden soll, darf es weder abstract noch extern sein und explizit keine event-accessor-declarations enthalten. Solche Ereignisse können in einem beliebigen Kontext verwendet werden, in dem Felder zulässig sind. Das Feld enthält einen Delegaten (§15), der auf die Liste der Ereignishandler verweist, die dem Ereignis hinzugefügt wurden. Falls keine Ereignishandler hinzugefügt wurden, enthält das Feld den Wert null.

In dem Beispiel

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control  
{  
 public event EventHandler Click;

protected void OnClick(EventArgs e) {  
 if (Click != null) Click(this, e);  
 }

public void Reset() {  
 Click = null;  
 }  
}

wird Click als Feld innerhalb der Button-Klasse verwendet. Anhand des Beispiels wird deutlich, dass das Feld überprüft, geändert und in Delegataufrufausdrücken verwendet werden kann. Durch die OnClick-Methode in der Button-Klasse wird das Click-Ereignis "ausgelöst". Da das Auslösen eines Ereignisses dem Aufrufen des Delegaten entspricht, der von dem Ereignis dargestellt wird, sind keine speziellen Sprachkonstrukte zum Auslösen von Ereignissen erforderlich. Vor dem Delegataufruf wird durch eine Überprüfung sichergestellt, dass der Delegat nicht null ist.

Außerhalb der Deklaration der Button-Klasse kann der Click-Member jedoch nur auf der linken Seite der Operatoren += und –= verwendet werden. So wird beispielsweise durch

b.Click += new EventHandler(…);

ein Delegat an die Aufrufliste des Click-Ereignisses angehängt und durch

b.Click –= new EventHandler(…);

ein Delegat aus der Aufrufliste des Click-Ereignisses entfernt.

Beim Kompilieren eines feldähnlichen Ereignisses erstellt der Compiler automatisch Speicher für den Delegaten sowie Accessoren für das Ereignis, die dem Delegatfeld Ereignishandler hinzufügen bzw. daraus entfernen. Die Operationen zum Hinzufügen oder Entfernen sind threadsicher und können (müssen jedoch nicht zwangsläufig) ausgeführt werden, während das enthaltende Objekt für ein Instanzereignis oder das Typobjekt (§8.12) für ein statisches Ereignis mit einer Sperre (§7.6.10.6) belegt ist.

Nach der Kompilierung einer Instanzereignisdeklaration der Form

class X  
{  
 public event D Ev;  
}

ergibt sich demnach folgender Code:

class X  
{  
 private D \_\_Ev; // field to hold the delegate

public event D Ev {  
 add {  
 /\* add the delegate in a thread safe way \*/  
 }

remove {  
 /\* remove the delegate in a thread safe way \*/  
 }  
 }  
}

Innerhalb der Klasse X führen Verweise auf Ev auf der linken Seite des +=- und des –=-Operators zum Aufruf der Accessoren zum Hinzufügen und Entfernen. Alle anderen Verweise auf Ev werden so kompiliert, dass stattdessen auf das verdeckte Feld \_\_Ev verwiesen wird (§7.6.4). Der Name "\_\_Ev" ist willkürlich gewählt. Das verdeckte Feld kann einen beliebigen anderen oder überhaupt keinen Namen haben.

### Ereignisaccessoren

Ereignisdeklarationen lassen in der Regel keine event-accessor-declarations zu (siehe oben dargestelltes Button-Beispiel). Dieser Deklarationstyp wäre aber empfehlenswert, wenn die Speicherkosten eines Feldes pro Ereignis eine nicht akzeptable Höhe erreichen. In solchen Fällen kann eine Klasse event-accessor-declarations enthalten und die Liste der Ereignishandler mittels eines privaten Mechanismus speichern.

Die event-accessor-declarations eines Ereignisses geben die ausführbaren Anweisungen an, die mit dem Hinzufügen und Entfernen der Ereignishandler verknüpft sind.

Die Accessordeklarationen bestehen aus einer add-accessor-declaration und einer remove-accessor-declaration. Eine Accessordeklaration besteht aus dem Token add oder remove gefolgt von einem block. Der mit einer add-accessor-declaration verknüpfte block gibt die Anweisungen an, die beim Hinzufügen eines Ereignishandlers ausgeführt werden. Der block, der mit einer remove-accessor-declaration verknüpft ist, gibt die Anweisungen zum Entfernen eines Ereignishandlers an.

Jede add-accessor-declaration und remove-accessor-declaration entspricht einer Methode mit einem einzigen Werteparameter des Eigenschaftentyps und einem void-Rückgabetyp. Der implizite Parameter eines Ereignisaccessors lautet value. Wenn ein Ereignis in einer Ereigniszuweisung vorkommt, wird der entsprechende Ereignisaccessor verwendet. Konkret bedeutet dies, dass beim Zuweisungsoperator += der add-Accessor und beim Zuweisungsoperator -= der remove-Accessor zum Einsatz kommt. In beiden Fällen wird der Operand auf der rechten Seite des Zuweisungsoperators als Argument für den Ereignisaccessor verwendet. Der Block einer add-accessor-declaration bzw. einer remove-accessor-declaration muss den Regeln für void-Methoden entsprechen (siehe §10.6.10). Insbesondere sind in einem solchen Block keine return-Anweisungen zum Angeben eines Ausdrucks zulässig.

Da ein Ereignisaccessor implizit einen Parameter mit der Bezeichnung value enthält, tritt während der Kompilierung ein Fehler auf, wenn eine lokale Variable oder Konstante, die in einem Ereignisaccessor deklariert wird, diesen Namen aufweist.

In dem Beispiel

class Control: Component  
{  
 // Unique keys for events  
 static readonly object mouseDownEventKey = new object();  
 static readonly object mouseUpEventKey = new object();

// Return event handler associated with key  
 protected Delegate GetEventHandler(object key) {...}

// Add event handler associated with key  
 protected void AddEventHandler(object key, Delegate handler) {...}

// Remove event handler associated with key  
 protected void RemoveEventHandler(object key, Delegate handler) {...}

// MouseDown event  
 public event MouseEventHandler MouseDown {  
 add { AddEventHandler(mouseDownEventKey, value); }  
 remove { RemoveEventHandler(mouseDownEventKey, value); }  
 }

// MouseUp event  
 public event MouseEventHandler MouseUp {  
 add { AddEventHandler(mouseUpEventKey, value); }  
 remove { RemoveEventHandler(mouseUpEventKey, value); }  
 }

// Invoke the MouseUp event  
 protected void OnMouseUp(MouseEventArgs args) {  
 MouseEventHandler handler;   
 handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);  
 if (handler != null)  
 handler(this, args);  
 }  
}

implementiert die Control-Klasse einen internen Speichermechanismus für Ereignisse. Die AddEventHandler-Methode verknüpft einen Delegatwert mit einem Schlüssel, die GetEventHandler-Methode gibt den gegenwärtig mit einem Schlüssel verknüpften Delegaten zurück, und die RemoveEventHandler-Methode entfernt einen Delegaten als Ereignishandler für das angegebene Ereignis. Der zugrunde liegende Speichermechanismus wurde vermutlich so entworfen, dass keine Kosten für das Verknüpfen eines null-Delegatwerts mit einem Schlüssel entstehen. Demnach benötigen nicht behandelte Ereignisse keinen Speicherplatz.

### Statische Ereignisse und Instanzereignisse

Wenn eine Ereignisdeklaration einen static-Modifizierer enthält, wird das Ereignis als statisches Ereignis bezeichnet. Wenn kein static-Modifizierer vorhanden ist, wird das Ereignis als Instanzereignis bezeichnet.

Für statische Ereignisse gilt, dass sie nicht mit einer bestimmten Instanz verknüpft sind und in den Accessoren eines statischen Ereignisses nicht auf this verwiesen werden darf, da andernfalls ein Kompilierungsfehler auftritt.

Ein Instanzereignis ist mit einer bestimmten Instanz einer Klasse verknüpft. Auf diese Instanz kann so wie auf this (§7.6.7) in den Accessoren der Eigenschaft zugegriffen werden.

Wenn in einem member-access (§7.6.4) der Form E.M auf ein Ereignis verwiesen wird und M eine statische Methode ist, muss E auf einen Typ deuten, der M enthält. Wenn M ein Instanzereignis ist, muss E auf eine Instanz eines Typs deuten, der M enthält.

Die Unterschiede zwischen statischen Membern und Instanzmembern werden in §10.3.7 näher erläutert.

### Virtuelle, versiegelte, überschreibungsbezogene und abstrakte Accessoren

Eine virtual-Ereignisdeklaration gibt an, dass die Accessoren des Ereignisses virtuell sind. Der virtual-Modifizierer gilt für beide Accessoren eines Ereignisses.

Eine abstract-Ereignisdeklaration gibt an, dass die Accessoren des Ereignisses virtuell sind, stellt jedoch keine Implementierung der Accessoren zur Verfügung. Stattdessen sind nicht abstrakte, abgeleitete Klassen erforderlich, um die Implementierung für die Accessoren durch Überschreiben des Ereignisses zu ermöglichen. Da eine abstrakte Ereignisdeklaration keine Implementierung zur Verfügung stellt, kann sie auch keine event-accessor-declarations bereitstellen, die durch geschweifte Klammern getrennt sind.

Eine Ereignisdeklaration, die sowohl den abstract-Modifizierer als auch den override-Modifizierer enthält, gibt an, dass das Ereignis abstrakt ist, und überschreibt eine Basiseigenschaft. Die Accessoren eines solchen Ereignisses sind ebenfalls abstrakt.

Abstrakte Ereignisdeklarationen sind nur in abstrakten Klassen zulässig (§10.1.1.1).

Die Accessoren eines geerbten virtuellen Ereignisses können in einer abgeleiteten Klasse durch Einschließen einer Ereignisdeklaration, die einen override-Modifizierer angibt, überschrieben werden. Dies wird als überschreibende Ereignisdeklaration bezeichnet. Mit einer überschreibenden Ereignisdeklaration wird kein neues Ereignis deklariert. Vielmehr werden mit ihr einfach die Implementierungen der Accessoren eines vorhandenen virtuellen Ereignisses spezialisiert.

Eine überschreibende Ereignisdeklaration muss genau dieselben Zugriffsmodifizierer, denselben Typ und denselben Namen wie das überschreibende Ereignis angeben.

Überschreibende Ereignisdeklarationen können den sealed-Modifizierer enthalten. Bei Verwendung dieses Modifizierers wird das weitere Überschreiben des Ereignisses durch eine abgeleitete Klasse verhindert. Die Accessoren eines versiegelten Ereignisses sind ebenfalls versiegelt.

Eine überschreibende Ereignisdeklaration darf keinen new-Modifizierer enthalten, da andernfalls ein Kompilierungsfehler ausgelöst wird.

Virtuelle Accessoren, versiegelte Accessoren, Überschreibungsaccessoren und abstrakte Accessoren verhalten sich abgesehen von Unterschieden bei der Deklarationssyntax und der Aufrufsyntax genauso wie virtuelle Methoden, versiegelte Methoden, Überschreibungsmethoden und abstrakte Methoden. Insbesondere in den in §10.6.3, §10.6.4, §10.6.5 und §10.6.6 beschriebenen Regeln werden Accessoren wie die entsprechenden Methoden behandelt. Jeder Accessor entspricht einer Methode mit einem Werteparameter des Eigenschaftentyps, einem void-Rückgabetyp und denselben Modifizierern wie die enthaltende Eigenschaft.

## Indexer

Ein Indexer ist ein Member, der es einem Objekt ermöglicht, auf die gleiche Art wie ein Array indiziert zu werden. Indexer werden mithilfe von indexer-declarations deklariert:

indexer-declaration:  
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:  
indexer-modifier  
indexer-modifiers indexer-modifier

indexer-modifier:  
new  
public  
protected  
internal  
private   
virtual  
sealed  
override  
abstract  
extern

indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

Eine indexer-declaration kann eine Gruppe von attributes (§17) und eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5), dem new (§10.3.4)-, virtual (§10.6.3)-, override (§10.6.4)-, sealed (§10.6.5)-, abstract (§10.6.6)- und extern (§10.6.7)-Modifizierer enthalten.

Indexerdeklarationen unterliegen im Hinblick auf gültige Modifiziererkombinationen denselben Regeln wie Methodendeklarationen (§10.6). Eine Ausnahme besteht jedoch darin, dass static-Modifizierer in Indexerdeklarationen nicht zulässig sind.

Die Modifizierer virtual, override und abstract schließen sich bis auf eine Ausnahme gegenseitig aus. Die Modifizierer abstract und override können zusammen verwendet werden, sodass ein virtueller durch einen abstrakten Indexer überschrieben werden kann.

Der type einer Indexerdeklaration gibt den Elementtyp des Indexers an, der durch die Deklaration eingeführt wurde. Wenn es sich beim Indexer nicht um eine explizite Implementierung eines Schnittstellenmembers handelt, folgt auf type das Schlüsselwort this. Bei einer expliziten Implementierung eines Schnittstellenmembers folgt auf den type ein interface-type, ein P"." und das Schlüsselwort this. Im Gegensatz zu anderen Membern haben Indexer keine benutzerdefinierten Namen.

Die formal-parameter-list gibt die Parameter des Indexers an. Bei einem Indexer unterscheidet sich diese Liste von der Liste formaler Parameter einer Methode (§10.6.1) dadurch, dass mindestens ein Parameter angegeben werden muss und dass die Parametermodifizierer ref und out nicht zulässig sind.

Der type eines Indexers und die einzelnen Typen, auf die in der formal-parameter-list verwiesen wird, müssen genauso verfügbar sein wie der Indexer selbst (§3.5.4).

Die accessor-declarations (§10.7.2), die in ein "{"-Token und ein "}"-Token eingeschlossen werden müssen, deklarieren die Accessoren des Indexers. Die Accessoren geben die ausführbaren Anweisungen an, die mit dem Lesen und Schreiben von Indexerelementen verknüpft sind.

Obwohl die Syntax für den Zugriff auf ein Indexerelement mit der Syntax für ein Arrayelement übereinstimmt, wird ein Indexerelement nicht als Variable klassifiziert. Deshalb ist es nicht möglich, ein Indexerelement als ref-Argument oder als out-Argument zu übergeben.

Anhand der Liste formaler Parameter eines Indexers wird die Signatur (§3.6) eines Indexers definiert. Die Signatur eines Indexers besteht aus der Anzahl und den Typen seiner formalen Parameter. Der Elementtyp und die Namen der formalen Parameter sind nicht Teil der Signatur eines Indexers.

Die Signatur eines Indexers muss sich von den Signaturen aller anderen, in derselben Klasse deklarierten Indexer unterscheiden.

Indexer und Eigenschaften sind konzeptionell sehr ähnlich, unterscheiden sich aber in folgenden Punkten:

* Eine Eigenschaft wird anhand des Namens identifiziert, während ein Indexer anhand der Signatur identifiziert wird.
* Auf eine Eigenschaft wird über einen simple-name (§7.6.2) oder einen member-access (§7.6.4) zugegriffen, während auf ein Indexerelement über einen element-access (§7.6.6.2) zugegriffen wird.
* Bei einer Eigenschaft kann es sich um einen static-Member handeln, während es sich bei einem Indexer immer um einen Instanzmember handelt.
* Der get-Accessor einer Eigenschaft entspricht einer Methode ohne Parameter, während der get-Accessor eines Indexers einer Methode mit derselben Liste formaler Parameter wie der Indexer entspricht.
* Der set-Accessor einer Eigenschaft entspricht einer Methode mit dem Parameter value, während der set-Accessor eines Indexers einer Methode entspricht, die über dieselbe Liste formaler Parameter wie der Indexer und darüber hinaus über einen zusätzlichen Parameter mit der Bezeichnung value verfügt.
* Ein Indexeraccessor darf keine lokale Variable mit demselben Namen wie ein Indexerparameter deklarieren, andernfalls wird ein Kompilierungsfehler erzeugt.
* In einer überschreibenden Eigenschaftendeklaration wird auf die geerbte Eigenschaft mithilfe der Syntax base.P zugegriffen, wobei P der Eigenschaftenname ist. In einer überschreibenden Indexerdeklaration wird auf den geerbten Indexer mithilfe der Syntax base[E] zugegriffen, wobei E eine durch Trennzeichen getrennte Liste von Ausdrücken ist.

Unter Berücksichtigung dieser Unterschiede gelten alle in §10.7.2 und §10.7.3 definierten Regeln für Indexeraccessoren sowie für Eigenschaftenaccessoren.

Wenn eine Indexerdeklaration einen extern-Modifizierer enthält, wird der Indexer als externer Indexer bezeichnet. Da eine externe Indexerdeklaration keine tatsächliche Implementierung zur Verfügung stellt, bestehen alle accessor-declarations eines Indexers aus einem Semikolon.

In dem unten dargestellten Beispiel wird eine BitArray-Klasse deklariert, die einen Indexer für den Zugriff der einzelnen Bits im Bitarray implementiert.

using System;

class BitArray  
{  
 int[] bits;  
 int length;

public BitArray(int length) {  
 if (length < 0) throw new ArgumentException();  
 bits = new int[((length - 1) >> 5) + 1];  
 this.length = length;  
 }

public int Length {  
 get { return length; }  
 }

public bool this[int index] {  
 get {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 return (bits[index >> 5] & 1 << index) != 0;  
 }  
 set {  
 if (index < 0 || index >= length) {  
 throw new IndexOutOfRangeException();  
 }  
 if (value) {  
 bits[index >> 5] |= 1 << index;  
 }  
 else {  
 bits[index >> 5] &= ~(1 << index);  
 }  
 }  
 }  
}

Eine Instanz der BitArray-Klasse nutzt im Wesentlichen weniger Speicher als eine entsprechende bool[] (da jeder Wert der vorherigen nur ein Bit und nicht das eine Byte der nachfolgenden belegt), lässt aber dieselben Operationen wie bool[] zu.

Die folgende CountPrimes-Klasse verwendet für die Berechnung der Anzahl der Primzahlen zwischen 1 und einem angegebenen Maximum ein BitArray und den klassischen "Sieb"-Algorithmus:

class CountPrimes  
{  
 static int Count(int max) {  
 BitArray flags = new BitArray(max + 1);  
 int count = 1;  
 for (int i = 2; i <= max; i++) {  
 if (!flags[i]) {  
 for (int j = i \* 2; j <= max; j += i) flags[j] = true;  
 count++;  
 }  
 }  
 return count;  
 }

static void Main(string[] args) {  
 int max = int.Parse(args[0]);  
 int count = Count(max);  
 Console.WriteLine("Found {0} primes between 1 and {1}", count, max);  
 }  
}

Die Syntax für den Zugriff auf Elemente von BitArray ist mit der Syntax für eine bool[]-Klasse identisch.

Im folgenden Beispiel wird eine Klasse mit einem Datenblatt 26 × 10 dargestellt, die einen Indexer mit zwei Parametern enthält. Dabei muss der erste Parameter ein Klein- oder Großbuchstabe im Bereich A–Z und der zweite Parameter eine Ganzzahl zwischen 0 und 9 sein.

using System;

class Grid  
{  
 const int NumRows = 26;  
 const int NumCols = 10;

int[,] cells = new int[NumRows, NumCols];

public int this[char c, int col] {  
 get {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 return cells[c - 'A', col];  
 }

set {  
 c = Char.ToUpper(c);  
 if (c < 'A' || c > 'Z') {  
 throw new ArgumentException();  
 }  
 if (col < 0 || col >= NumCols) {  
 throw new IndexOutOfRangeException();  
 }  
 cells[c - 'A', col] = value;  
 }  
 }  
}

### Indexerüberladung

Die Auflösungsregeln für die Indexerüberladung werden in §7.5.2 erläutert.

## Operatoren

Ein operator ist ein Member, der die Bedeutung eines Ausdrucksoperators definiert, der auf die Instanzen der Klasse angewendet werden kann. Operatoren werden mithilfe von operator-declarations deklariert:

operator-declaration:  
attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:  
operator-modifier  
operator-modifiers operator-modifier

operator-modifier:  
public  
static  
extern

operator-declarator:  
unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

unary-operator-declarator:  
type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of  
+ - ! ~ ++ -- true false

binary-operator-declarator:  
type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:  
+  
-  
\*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
<=

conversion-operator-declarator:  
implicit operator type ( type identifier )  
explicit operator type ( type identifier )

operator-body:  
block  
;

Es gibt drei Kategorien von überladbaren Operatoren: unäre Operatoren (§10.10.1), binäre Operatoren (§10.10.2) und Konvertierungsoperatoren (§10.10.3).

Wenn eine Operatordeklaration einen extern-Modifizierer enthält, wird der Operator als externer Operator bezeichnet. Da ein externer Operator keine tatsächliche Implementierung zur Verfügung stellt, besteht der operator-body aus einem Semikolon. Bei allen anderen Operatoren besteht der operator-body aus einem block, der die Anweisungen angibt, die beim Aufruf des Operators auszuführen sind. Der block eines Operators muss den Regeln für Methoden entsprechen, die Werte zurückgeben (siehe §10.6.10).

Für alle Operatordeklarationen gelten folgende Regeln:

* Eine Operatordeklaration muss sowohl einen public-Modifizierer als auch einen static-Modifizierer enthalten.
* Bei den Parametern eines Operators muss es sich um Werteparameter handeln (§5.1.4). Wenn in einer Operatordeklaration ein ref-Parameter oder out-Parameter angegeben wird, tritt ein Kompilierungsfehler auf.
* Die Signatur eines Operators (§10.10.1, §10.10.2, §10.10.3) muss sich von der Signatur aller anderen, in derselben Klasse deklarierten Operatoren unterscheiden.
* Alle Typen, auf die in einer Operatordeklaration verwiesen wird, müssen mindestens genauso verfügbar sein wie der Operator selbst (§3.5.4).
* Wenn derselbe Modifizierer mehrmals in einer Operatordeklaration aufgeführt ist, wird ein Fehler verursacht.

Die jeweilige Operatorkategorie umfasst zusätzliche Einschränkungen. Diese werden in den folgenden Abschnitten näher beschrieben.

Operatoren, die in einer Basisklasse deklariert werden, werden wie bei anderen Membern von abgeleiteten Klassen geerbt. Da Operatordeklarationen immer die Klasse oder die Struktur erfordern, in der der Operator für die Beteiligung an der Signatur des Operators deklariert wird, kann ein Operator, der in einer abgeleiteten Klasse deklariert wird, keinen Operator verdecken, der in einer Basisklasse deklariert wurde. Somit ist der new-Modifizierer in einer Operatordeklaration nie erforderlich und auch nicht zulässig.

Weitere Informationen zu unären und binären Operatoren finden Sie in §7.3.

Weitere Informationen zu Konvertierungsoperatoren finden Sie in §6.4.

### Unäre Operatoren

Die folgenden Regeln gelten für unäre Operatordeklarationen, wobei T auf den Instanztyp der Klasse oder Struktur deutet, der die Operatordeklaration enthält:

* Ein unärer +-, --, !- oder ~-Operator muss einen Parameter vom Typ T oder T? enthalten und kann einen beliebigen Typ zurückgeben.
* Ein unärer ++- oder -- -Operator muss einen Parameter vom Typ T oder T? akzeptieren und denselben Typ oder einen von diesem abgeleiteten Typ zurückgeben.
* Ein unärer true- oder false-Operator muss einen einzelnen Parameter vom Typ T oder T? akzeptieren und den Typ bool zurückgeben.

Die Signatur eines unären Operators besteht aus dem Operatortoken (+, -, !, ~, ++, --, true oder false) und dem Typ des formalen Parameters. Weder der Rückgabetyp noch der Name des formalen Parameters sind Teil der Signatur des unären Operators.

Für den unären true-Operator und den unären false-Operator ist eine paarweise Deklaration erforderlich. Wenn eine Klasse einen dieser Operatoren deklariert, ohne auch den anderen zu deklarieren, tritt ein Kompilierungsfehler auf. Der true-Operator und der false-Operator werden in §7.12.2 und §7.20 näher erläutert.

Im folgenden Beispiel wird die Implementierung und die anschließende Verwendung von operator ++ für eine Ganzzahlvektorklasse dargestellt:

public class IntVector  
{  
 public IntVector(int length) {...}

public int Length {...} // read-only property

public int this[int index] {...} // read-write indexer

public static IntVector operator ++(IntVector iv) {  
 IntVector temp = new IntVector(iv.Length);  
 for (int i = 0; i < iv.Length; i++)  
 temp[i] = iv[i] + 1;  
 return temp;  
 }  
}

class Test  
{  
 static void Main() {  
 IntVector iv1 = new IntVector(4); // vector of 4 x 0  
 IntVector iv2;

iv2 = iv1++; // iv2 contains 4 x 0, iv1 contains 4 x 1  
 iv2 = ++iv1; // iv2 contains 4 x 2, iv1 contains 4 x 2  
 }  
}

Beachten Sie, dass die Operatormethode den Wert zurückgibt, der durch Hinzufügen von 1 zum Operanden erzeugt wird, genauso wie die in §7.6.9 beschriebenen Postfix-Inkrementoperatoren und -Dekrementoperatoren und die in §7.7.5 beschriebenen Präfix-Inkrementoperatoren und -Dekrementoperatoren. Im Unterschied zu C++ muss der Wert des Operanden von dieser Methode nicht direkt geändert werden. Eine Änderung des Operandenwerts würde sogar gegen die Standardsemantik des Postfix-Inkrementoperators verstoßen.

### Binäre Operatoren

Die folgenden Regeln gelten für binäre Operatordeklarationen, wobei T auf den Instanztyp der Klasse oder Struktur deutet, der die Operatordeklaration enthält:

* Ein binärer Nicht-Schiebe-Operator muss zwei Parameter enthalten, von denen mindestens einer vom Typ T oder T? ist, und kann einen beliebigen Typ zurückgeben.
* Ein binärer <<- oder >>-Operator muss zwei Parameter enthalten, von denen der erste vom Typ T oder T? und der zweite vom Typ int oder int?sein muss, und kann einen beliebigen Typ zurückgeben.

Die Signatur eines binären Operators besteht aus dem Operatortoken (+, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >= oder <=) und den Typen der beiden formalen Parameter. Der Rückgabetyp und die Namen der formalen Parameter sind nicht Teil der Signatur eines binären Operators.

Für bestimmte binäre Operatoren ist die paarweise Deklaration erforderlich. Bei jeder Deklaration für einen der beiden Operatoren eines Paares muss eine entsprechende Deklaration des anderen Operators des Paares vorhanden sein. Zwei Operatordeklarationen stimmen überein, wenn sie für jeden Parameter denselben Rückgabetyp haben. Für folgende Operatoren ist die paarweise Deklaration erforderlich:

* operator == und operator !=
* operator > und operator <
* operator >= und operator <=

### Konvertierungsoperatoren

Eine Konvertierungsoperatordeklaration führt eine benutzerdefinierte Konvertierung (§6.4) ein, mit der die vordefinierten impliziten und expliziten Konvertierungen erweitert werden.

Eine Konvertierungsoperatordeklaration, die das implicit-Schlüsselwort enthält, führt eine implizite benutzerdefinierte Konvertierung ein. Implizite Konvertierungen können in einer Vielzahl von Situationen auftreten, z. B. in Funktionsmemberaufrufen, Umwandlungsausdrücken und Zuweisungen. Dies wird in §6.1 näher erläutert.

Eine Konvertierungsoperatordeklaration, die das explicit-Schlüsselwort enthält, führt eine explizite benutzerdefinierte Konvertierung ein. Explizite Konvertierungen können in Umwandlungsausdrücken auftreten (siehe §6.2).

Ein Konvertierungsoperator konvertiert aus einem Quelltyp, der von dem Parametertyp des Konvertierungsoperators angegeben wird, in den Zieltyp, der von dem Rückgabetyp des Konvertierungsoperators angegeben wird.

Für einen angegebenen Quelltyp S und einen Zieltyp T, wobei S oder T Typen sind, die NULL-Werte zulassen, verweisen S0 und T0 auf ihre zugrunde liegenden Typen, andernfalls sind S0und T0 gleich S bzw. T. Eine Klasse oder Struktur darf eine Konvertierung von einem Quelltyp S in einen Zieltyp T nur deklarieren, wenn die folgenden Voraussetzungen erfüllt sind:

* S0 und T0 sind unterschiedliche Typen.
* Der Klassen- oder Strukturtyp, in dem die Operatordeklaration erfolgt, ist entweder S0 oder T0.
* Weder S0 noch T0 ist als interface-type angegeben.
* Mit Ausnahme von benutzerdefinierten Konvertierungen sind keine weiteren Konvertierungen von S in T oder von T in S vorhanden.

Wenn diese Regeln angewendet werden, werden alle S oder T zugeordneten Typparameter als eindeutige Typen angesehen, die über keine Vererbungsbeziehung mit anderen Typen verfügen, sodass alle Einschränkungen für diese Typparameter ignoriert werden.

In dem Beispiel

class C<T> {...}

class D<T>: C<T>  
{  
 public static implicit operator C<int>(D<T> value) {...} // Ok

public static implicit operator C<string>(D<T> value) {...} // Ok

public static implicit operator C<T>(D<T> value) {...} // Error  
}

sind die ersten beiden Operatordeklarationen zulässig, da zum Zweck von §10.9.3 T und int bzw. string als eindeutige Typen ohne Beziehungen angesehen werden. Der dritte Operator führt aber zu einem Fehler, da C<T> die Basisklasse von D<T> ist.

Aus der zweiten Regel ergibt sich, dass ein Konvertierungsoperator entweder in oder aus dem Klassentyp bzw. Strukturtyp konvertieren muss, in dem der Operator deklariert wird. Ein Klassen- bzw. Strukturtyp C kann beispielsweise eine Konvertierung von C in int und von int in C, jedoch nicht von int in bool definieren.

Eine vordefinierte Konvertierung kann nicht direkt neu definiert werden. Konvertierungsoperatoren können demnach nicht aus oder in object konvertieren, weil zwischen object und allen anderen Typen bereits implizite und explizite Konvertierungen vorhanden sind. Ebenso können weder die Quelltypen noch die Zieltypen einer Konvertierung der Basistyp des anderen Typs sein, weil dann bereits eine Konvertierung vorhanden wäre.

Es ist aber möglich, Operatoren für generische Typen zu deklarieren, die für bestimmte Typargumente Konvertierungen angeben, die bereits als vordefinierte Konvertierungen vorhanden sind. In dem Beispiel

struct Convertible<T>  
{  
 public static implicit operator Convertible<T>(T value) {...}

public static explicit operator T(Convertible<T> value) {...}  
}

deklariert der zweite Operator, wenn der Typ object als Typargument für T angegeben wird, eine bereits vorhandene Konvertierung (eine implizite und daher auch eine explizite Konvertierung aus jedem Typ in den Typ object).

Wenn bereits eine vordefinierte Konvertierung zwischen zwei Typen vorhanden ist, werden die benutzerdefinierten Konvertierungen zwischen diesen Typen ignoriert. Genauer gesagt:

* Wenn eine vordefinierte implizite Konvertierung (§6.1) vom Typ S in den Typ T vorhanden ist, werden alle benutzerdefinierten (impliziten und expliziten) Konvertierungen aus S in T ignoriert.
* Wenn eine vordefinierte explizite Konvertierung (§6.2) vom Typ S in den Typ T vorhanden ist, werden alle benutzerdefinierten expliziten Konvertierungen aus S in T ignoriert. Außerdem gilt:
* Wenn T ein Schnittstellentyp ist, werden benutzerdefinierte implizite Konvertierungen von S in T ignoriert:
* Andernfalls werden benutzerdefinierte implizite Konvertierungen von S in T aber weiterhin berücksichtigt.

Bei allen Typen außer object führen die durch den oben stehenden Typ Convertible<T> deklarierten Operatoren nicht zu Konflikten mit vordefinierten Konvertierungen. Beispiel:

void F(int i, Convertible<int> n) {  
 i = n; // Error  
 i = (int)n; // User-defined explicit conversion  
 n = i; // User-defined implicit conversion  
 n = (Convertible<int>)i; // User-defined implicit conversion  
}

Beim Typ object werden allerdings benutzerdefinierte Konvertierungen in allen Fällen außer einem durch die vordefinierten Konvertierungen verdeckt:

void F(object o, Convertible<object> n) {  
 o = n; // Pre-defined boxing conversion  
 o = (object)n; // Pre-defined boxing conversion  
 n = o; // User-defined implicit conversion  
 n = (Convertible<object>)o; // Pre-defined unboxing conversion  
}

Benutzerdefinierte Konvertierungen dürfen nicht aus oder in interface-types konvertieren. Diese Einschränkung stellt insbesondere sicher, dass bei Konvertierungen in einen interface-type keine benutzerdefinierten Transformationen ausgeführt werden und eine Konvertierung in einen interface-type nur erfolgreich ausgeführt wird, wenn das konvertierte Objekt den angegebenen interface-type tatsächlich implementiert.

Die Signatur eines Konvertierungsoperators besteht aus dem Quelltyp und dem Zieltyp. (Dies ist die einzige Form eines Members, bei dem der Rückgabetyp an der Signatur teilnimmt.) Die implicit-Klassifizierung bzw. die explicit-Klassifizierung eines Konvertierungsoperators ist nicht Teil der Signatur des Operators. Daher kann eine Klasse oder eine Struktur nicht sowohl einen implicit-Konvertierungsoperator als auch einen explicit-Konvertierungsoperator mit denselben Quelltypen und Zieltypen deklarieren.

Im Allgemeinen sollten benutzerdefinierte implizite Konvertierungen so entworfen werden, dass dabei unter keinen Umständen Ausnahmen ausgelöst werden und Informationen verloren gehen. Wenn eine benutzerdefinierte Konvertierung Ausnahmen hervorruft (z. B. weil das Quellargument außerhalb des gültigen Bereichs liegt) oder zum Verlust von Informationen führt (z. B. durch Verwerfen von höherwertigen Bits), dann sollte diese Konvertierung als explizite Konvertierung definiert werden.

In dem Beispiel

using System;

public struct Digit  
{  
 byte value;

public Digit(byte value) {  
 if (value < 0 || value > 9) throw new ArgumentException();  
 this.value = value;  
 }

public static implicit operator byte(Digit d) {  
 return d.value;  
 }

public static explicit operator Digit(byte b) {  
 return new Digit(b);  
 }  
}

ist die Konvertierung aus Digit in byte implizit, weil nie Ausnahmen ausgelöst werden oder Information verloren gehen. Die Konvertierung aus byte in Digit ist jedoch explizit, da Digit nur eine Teilmenge der möglichen Werte von einem byte darstellen kann.

## Instanzkonstruktoren

Ein Instanzkonstruktor ist ein Member, der die Aktionen implementiert, die zum Initialisieren der Instanz einer Klasse erforderlich sind. Instanzkonstruktoren werden mithilfe von constructor-declarations deklariert:

constructor-declaration:  
attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:  
constructor-modifier  
constructor-modifiers constructor-modifier

constructor-modifier:  
public  
protected  
internal  
private  
extern

constructor-declarator:  
identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:  
: base ( argument-listopt )  
: this ( argument-listopt )

constructor-body:  
block  
;

Eine constructor-declaration kann eine Gruppe von attributes (§17), eine gültige Kombination der vier Zugriffsmodifizierer (§10.3.5) und einen extern-Modifizierer (§10.6.7) enthalten. In einer Konstruktordeklaration dürfen keine Modifizierer mehrfach enthalten sein.

Der identifier eines constructor-declarator muss die Klasse benennen, in der der Instanzkonstruktor deklariert wird. Bei Angabe eines anderen Namens tritt ein Fehler während der Kompilierung auf.

Die optionale formal-parameter-list eines Instanzkonstruktors unterliegt denselben Regeln wie die formal-parameter-list einer Methode (§10.6). Mit der Liste formaler Parameter wird die Signatur (§3.6) eines Instanzkonstruktors definiert und der Prozess bestimmt, mit dem die Überladungsauflösung (§7.5.2)) einen bestimmten Instanzkonstruktor in einem Aufruf auswählt.

Alle Typen, auf die in der formal-parameter-list eines Instanzkonstruktors verwiesen wird, müssen mindestens genauso verfügbar sein wie der Konstruktor selbst (§3.5.4).

Der optionale constructor-initializer gibt einen weiteren Instanzkonstruktor an, der aufgerufen wird, bevor die im constructor-body dieses Instanzkonstruktors angegebenen Anweisungen ausgeführt werden. Dies wird in §10.11.1 näher erläutert.

Wenn eine Konstruktordeklaration einen extern-Modifizierer enthält, wird der Konstruktor als externer Konstruktor bezeichnet. Da eine externe Konstruktordeklaration keine tatsächliche Implementierung zur Verfügung stellt, besteht ihr constructor-body aus einem Semikolon. Bei allen anderen Konstruktoren besteht der constructor-body aus einem block, der die Anweisungen für die Initialisierung einer neuen Instanz der Klasse angibt. Dies entspricht dem block einer Instanzmethode mit einem void-Rückgabetyp (§10.6.10).

Instanzkonstruktoren sind nicht geerbt. Demnach verfügt eine Klasse nur über die Instanzkonstruktoren, die in der Klasse tatsächlich deklariert wurden. Wenn eine Klasse keine Instanzkonstruktordeklarationen enthält, wird automatisch ein Standardinstanzkonstruktor zur Verfügung gestellt (§10.11.4).

Instanzkonstruktoren werden mithilfe von object-creation-expressions (§7.6.10.1) und über constructor-initializer aufgerufen.

### Konstruktorinitialisierungen

Alle Instanzkonstruktoren (außer die der Klasse object) enthalten implizit unmittelbar vor dem constructor-body einen Aufruf eines anderen Instanzkonstruktors. Die Festlegung des Konstruktors, der implizit aufgerufen wird, erfolgt durch den constructor-initializer:

* Ein Instanzkonstruktorinitialisierer der Form base(argument-listopt) führt dazu, dass ein Instanzkonstruktor aus der direkten Basisklasse aufgerufen wird. Dieser Konstruktor wird anhand der argument-list und der Überladungsauflösungsregeln in §7.5.3 ausgewählt. Die Gruppe der Instanzkonstruktoren besteht aus allen zugreifbaren Instanzkonstruktoren, die in der direkten Basisklasse oder dem Standardkonstruktor (§10.11.4) enthalten sind, wenn in der direkten Basisklasse keine Instanzkonstruktoren deklariert werden. Wenn diese Gruppe leer ist oder kein einzelner optimaler Instanzkonstruktor identifiziert werden kann, tritt während der Kompilierung ein Fehler auf.
* Ein Instanzkonstruktorinitialisierer der Form this(argument-listopt) führt dazu, dass ein Instanzkonstruktor aus der Klasse selbst aufgerufen wird. Der Konstruktor wird anhand der argument-list und der Überladungsauflösungsregeln in §7.5.3 ausgewählt. Die Gruppe der potenziellen Instanzkonstruktoren besteht aus allen in der Klasse selbst deklarierten verfügbaren Instanzkonstruktoren. Wenn diese Gruppe leer ist oder kein einzelner optimaler Instanzkonstruktor identifiziert werden kann, tritt während der Kompilierung ein Fehler auf. Wenn eine Instanzkonstruktordeklaration eine Konstruktorinitialisierung enthält, die den Konstruktor selbst aufruft, tritt während der Kompilierung ein Fehler auf.

Wenn ein Instanzkonstruktor nicht über ein Konstruktorinitialisierer verfügt, wird implizit ein Konstruktorinitialisierer der Form base() zur Verfügung gestellt. Daher ist eine Instanzkonstruktordeklaration der Form

C(...) {...}

mit folgender Anweisung äquivalent

C(...): base() {...}

Der Gültigkeitsbereich der in der formal-parameter-list angegebenen Parameter einer Instanzkonstruktordeklaration enthält der Konstruktorinitialisierer dieser Deklaration. Somit darf eine Konstruktorinitialisierung auf die Parameter des Konstruktors zugreifen. Beispiel:

class A  
{  
 public A(int x, int y) {}  
}

class B: A  
{  
 public B(int x, int y): base(x + y, x - y) {}  
}

Eine Instanzkonstruktorinitialisierung kann nicht auf die Instanz zugreifen, die gerade erstellt wird. Daher gilt, dass in einem Argumentausdruck des Konstruktorinitialisierers nicht auf this verwiesen werden darf und dass ein Argumentausdruck nicht über einen simple-name auf Instanzmember verweisen darf, da andernfalls ein Kompilierungsfehler auftritt.

### Instanzvariableninitialisierungen

Wenn ein Instanzkonstruktor keinen Konstruktorinitialisierer oder einen Konstruktorinitialisierer der Form base(...) hat, führt er implizit die Initialisierer aus, die von den variable-initializers der in seiner Klasse deklarierten Instanzfelder angegeben werden. Dies entspricht einer Reihe von Zuweisungen, die unmittelbar nach Eintritt in den Konstruktor und vor dem impliziten Aufruf des Konstruktors der direkten Basisklasse ausgeführt werden. Die Variableninitialisierungen werden in der Textreihenfolge ausgeführt, in der sie in der Klassendeklaration aufgeführt werden.

### Konstruktorausführung

Variableninitialisierungen werden in Zuweisungsanweisungen umgewandelt, die dann vor dem Aufrufen des Basisklasseninstanzkonstruktors ausgeführt werden. Aufgrund dieser Reihenfolge ist sichergestellt, dass Instanzfelder von ihrer Variableninitialisierung initialisiert werden, bevor Anweisungen ausgeführt werden, die Zugriff auf diese Instanz haben.

In diesem Beispiel

using System;

class A  
{  
 public A() {  
 PrintFields();  
 }

public virtual void PrintFields() {}

}

class B: A  
{  
 int x = 1;  
 int y;

public B() {  
 y = -1;  
 }

public override void PrintFields() {  
 Console.WriteLine("x = {0}, y = {1}", x, y);  
 }  
}

wird folgende Ausgabe erzeugt, wenn new B() zum Erstellen einer Instanz von B verwendet wird:

x = 1, y = 0

Der Wert von x lautet 1, weil der Variableninitialisierer vor dem Aufrufen des Basisklasseninstanzkonstruktors ausgeführt wird. Der Wert von y ist jedoch 0 (der Standardwert von int), weil y erst zugewiesen wird, nachdem der Basisklassenkonstruktor zurückgegeben wurde.

Instanzvariableninitialisierer und Konstruktorinitialisierer können sinnvollerweise auch als Anweisungen betrachtet werden, die automatisch vor dem constructor-body eingefügt werden. In dem Beispiel

using System;  
using System.Collections;

class A  
{  
 int x = 1, y = -1, count;

public A() {  
 count = 0;  
 }

public A(int n) {  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2 = Math.Sqrt(2.0);  
 ArrayList items = new ArrayList(100);  
 int max;

public B(): this(100) {  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 max = n;  
 }  
}

enthält mehrere Variableninitialisierer sowie Konstruktorinitialisierer beider Formen (base und this). Das Beispiel entspricht dem unten dargestellten Code, in dem jeder Kommentar eine automatisch eingefügte Anweisung angibt (diese Syntax für die automatisch eingefügten Konstruktoraufrufe ist nicht zulässig, sondern wird nur zur Veranschaulichung des Mechanismus verwendet).

using System.Collections;

class A  
{  
 int x, y, count;

public A() {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = 0;  
 }

public A(int n) {  
 x = 1; // Variable initializer  
 y = -1; // Variable initializer  
 object(); // Invoke object() constructor  
 count = n;  
 }  
}

class B: A  
{  
 double sqrt2;  
 ArrayList items;  
 int max;

public B(): this(100) {  
 B(100); // Invoke B(int) constructor  
 items.Add("default");  
 }

public B(int n): base(n – 1) {  
 sqrt2 = Math.Sqrt(2.0); // Variable initializer  
 items = new ArrayList(100); // Variable initializer  
 A(n – 1); // Invoke A(int) constructor  
 max = n;  
 }  
}

### Standardkonstruktoren

Wenn eine Klasse keine Instanzkonstruktordeklarationen enthält, wird automatisch ein Standardinstanzkonstruktor zur Verfügung gestellt. Dieser Standardkonstruktor ruft einfach den parameterlosen Konstruktor der direkten Basisklasse auf. Wenn die Klasse abstrakt ist, wird der deklarierte Zugriff für den Standardkonstruktor geschützt. Andernfalls ist der deklarierte Zugriff für den Standardkonstruktor öffentlich. Daher hat ein Standardkonstruktor die Form

protected C(): base() {}

oder

public C(): base() {}

wobei C der Name der Klasse ist. Wenn die Überladungsauflösung nicht in der Lage ist, einen eindeutigen besten Kandidaten für den Initialisierer des Basisklassenkonstruktors auszuwählen, tritt ein Kompilierungsfehler auf.

In dem Beispiel

class Message  
{  
 object sender;  
 string text;  
}

wird ein Standardkonstruktor zur Verfügung gestellt, weil die Klasse keine Instanzkonstruktordeklarationen enthält. Daher ist dieses Beispiel mit folgendem Beispiel identisch:

class Message  
{  
 object sender;  
 string text;

public Message(): base() {}  
}

### Private Konstruktoren

Wenn von einer Klasse T nur private Instanzkonstruktoren deklariert werden, können Klassen außerhalb des Programmtexts von T nicht von T ableiten bzw. keine Instanzen direkt von T erstellen. Wenn eine Klasse nur statische Member enthält und nicht instanziiert werden soll, wird die Instanziierung folglich durch Hinzufügen eines leeren privaten Instanzkonstruktors verhindert. Beispiel:

public class Trig  
{  
 private Trig() {} // Prevent instantiation

public const double PI = 3.14159265358979323846;

public static double Sin(double x) {...}  
 public static double Cos(double x) {...}  
 public static double Tan(double x) {...}  
}

Die Trig-Klasse fasst verknüpfte Methoden und Konstanten in einer Gruppe zusammen, wird jedoch nicht instanziiert. Aus diesem Grund deklariert sie einen leeren privaten Instanzkonstruktor. Um die automatische Erstellung eines Standardkonstruktors zu unterdrücken, muss mindestens ein Instanzkonstruktor deklariert werden.

### Parameter für optionale Instanzkonstruktoren

Die this(...)-Form eines Konstruktorinitialisierer wird in der Regel in Verbindung mit der Überladung verwendet, um Parameter für optionale Instanzkonstruktoren zu implementieren. In dem Beispiel

class Text  
{  
 public Text(): this(0, 0, null) {}

public Text(int x, int y): this(x, y, null) {}

public Text(int x, int y, string s) {  
 // Actual constructor implementation  
 }  
}

stellen die beiden ersten Instanzkonstruktoren lediglich die Standardwerte für die fehlenden Argumente zur Verfügung. Beide verwenden den this(...)-Konstruktorinitialisierer zum Aufrufen des dritten Instanzkonstruktors, der die neue Instanz tatsächlich initialisiert. Dies hat dieselbe Wirkung wie die Verwendung von Parametern für optionale Konstruktoren:

Text t1 = new Text(); // Same as Text(0, 0, null)  
Text t2 = new Text(5, 10); // Same as Text(5, 10, null)  
Text t3 = new Text(5, 20, "Hello");

## Statische Konstruktoren

Ein statischer Konstruktor ist ein Member, der die Aktionen implementiert, die zum Initialisieren eines geschlossenen Klassentyps erforderlich sind. Statische Konstruktoren werden mithilfe von static-constructor-declarations deklariert:

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

Eine static-constructor-declaration kann eine Gruppe von attributes (§17) und einen extern-Modifizierer (§10.6.7) enthalten.

Der identifier einer static-constructor-declaration muss die Klasse benennen, in der der statische Konstruktor deklariert wird. Bei Angabe eines anderen Namens tritt ein Fehler während der Kompilierung auf.

Wenn eine Deklaration für einen statischen Konstruktor einen extern-Modifizierer enthält, wird der statische Konstruktor als externer statischer Konstruktor bezeichnet. Da eine Deklaration für einen externen statischen Konstruktor keine tatsächliche Implementierung zur Verfügung stellt, besteht ihr static-constructor-body aus einem Semikolon. Bei allen anderen Deklarationen für statische Konstruktoren besteht der static-constructor-body aus einem block, der die Anweisungen angibt, die zur Initialisierung der Klasse auszuführen sind. Dies entspricht dem method-body einer statischen Methode mit einem void-Rückgabetyp (§10.6.10).

Statische Konstruktoren werden nicht geerbt und können nicht direkt aufgerufen werden.

Der statische Konstruktor für einen geschlossenen Klassentyp wird in einer bestimmten Anwendungsdomäne höchstens einmal ausgeführt. Die Ausführung eines statischen Konstruktors wird durch das erste der folgenden Ereignisse ausgelöst, die in einer Anwendungsdomäne auftreten:

* Eine Instanz des Klassentyps wird erstellt.
* Es wird auf beliebige statische Member des Klassentyps verwiesen.

Wenn eine Klasse die Methode Main (§3.1) enthält, in der die Ausführung gestartet wird, wird der statische Konstruktor dieser Klasse vor dem Aufruf der Main-Methode ausgeführt.

Zum Initialisieren eines neuen geschlossen Klassentyps wird zunächst ein neuer Satz von statischen Feldern (§10.5.1) für den speziellen geschlossen Typ erstellt. Jedes der statischen Felder wird mit seinem Standardwert (§5.2) initialisiert. Anschließend werden die Initialisierungen für statische Felder (§10.4.5.1) für diese statischen Felder ausgeführt. Abschließend wird der statische Konstruktor ausgeführt.

In dem Beispiel

using System;

class Test  
{  
 static void Main() {  
 A.F();  
 B.F();  
 }  
}

class A  
{  
 static A() {  
 Console.WriteLine("Init A");  
 }  
 public static void F() {  
 Console.WriteLine("A.F");  
 }  
}

class B  
{  
 static B() {  
 Console.WriteLine("Init B");  
 }  
 public static void F() {  
 Console.WriteLine("B.F");  
 }  
}

sollte die folgende Ausgabe ergeben

Init A  
A.F  
Init B  
B.F

da die Ausführung des statischen Konstruktors von A durch den Aufruf von A.F und die Ausführung des statischen Konstruktors von B durch den Aufruf von B.F ausgelöst wird.

Es besteht die Möglichkeit, zirkuläre Abhängigkeiten zu erstellen, durch die statische Felder mit Variableninitialisierungen bei Angabe des Standardwertes beobachtet werden können.

In dem Beispiel

using System;

class A  
{  
 public static int X;

static A() {  
 X = B.Y + 1;  
 }  
}

class B  
{  
 public static int Y = A.X + 1;

static B() {}

static void Main() {  
 Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);  
 }  
}

erzeugt die Ausgabe

X = 1, Y = 2

Um die Main-Methode auszuführen, führt das System vor dem statischen Konstruktor von Klasse B zuerst den Initialisierer für B.Y aus. Der Initialisierer von Y’ bewirkt, dass der statische Konstruktor von A ausgeführt wird, da auf den Wert von A.X verwiesen wird. Der statische Konstruktor von A fährt wiederum mit der Berechnung des Werts von X fort. Dabei ruft er den Standardwert von Y (= 0) ab. A.X wird folglich mit 1 initialisiert. Die Ausführung der Initialisierer für statische Felder von A und die des statischen Konstruktors wird daraufhin abgeschlossen, wobei zur Berechnung des Anfangswerts von Y zurückgekehrt wird, die den Wert 2 ergibt.

Da der statische Konstruktor genau einmal für jeden geschlossen konstruierten Klassentyp ausgeführt wird, können an dieser Stelle Laufzeitüberprüfungen für Typparameter eingefügt werden, die während der Kompilierung nicht über Einschränkungen (§10.1.5) durchgeführt werden können. Der folgende Typ verwendet beispielsweise einen statischen Konstruktor, um zu erzwingen, dass das Typargument eine Enumeration ist:

class Gen<T> where T: struct  
{  
 static Gen() {  
 if (!typeof(T).IsEnum) {  
 throw new ArgumentException("T must be an enum");  
 }  
 }  
}

## Destruktoren

Destruktoren sind Member, die die Aktionen implementieren, die zum Zerstören einer Klasseninstanz erforderlich sind. Ein Destruktor wird mithilfe einer destructor-declaration deklariert:

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

Eine destructor-declaration kann eine Gruppe von attributes (§17) enthalten.

Der identifier eines destructor-declarator muss die Klasse benennen, in der der Destruktor deklariert wird. Bei Angabe eines anderen Namens tritt ein Fehler während der Kompilierung auf.

Wenn eine Destruktordeklaration einen extern-Modifizierer enthält, wird der Destruktor als externer Destruktor bezeichnet. Da eine externe Destruktordeklaration keine tatsächliche Implementierung zur Verfügung stellt, besteht ihr destructor-body aus einem Semikolon. Bei allen anderen Destruktoren besteht der destructor-body aus einem block, der die Anweisungen für die Zerstörung einer Instanz der Klasse angibt. Ein destructor-body entspricht dem method-body einer Instanzmethode mit einem void-Rückgabetyp (§10.6.10).

Destruktoren sind nicht geerbt. Demnach verfügt eine Klasse nur über den Destruktor, der in dieser Klasse ggf. deklariert wurde.

Da ein Destruktor keine Parameter enthalten darf, kann er nicht überladen werden. Eine Klasse kann also höchstens einen Destruktor enthalten.

Destruktoren werden automatisch aufgerufen und können nicht explizit aufgerufen werden. Eine Instanz wird für die Zerstörung freigegeben, wenn sie von keinem Code mehr verwendet werden kann. Nachdem die Instanz für die Zerstörung freigegeben wurde, kann der Destruktor jederzeit für die Instanz ausgeführt werden. Beim Zerstören einer Instanz werden die Destruktoren in ihrer Vererbungskette der Reihe nach aufgerufen, angefangen bei dem am meisten abgeleiteten Destruktor bis hin zum am wenigsten abgeleiteten Destruktor. Ein Destruktor kann für einen beliebigen Thread ausgeführt werden. In §3.9 werden die Richtlinien näher erläutert, durch die festgelegt wird, wann und wie ein Destruktor ausgeführt wird.

Die Ausgabe des Beispiels

using System;

class A  
{  
 ~A() {  
 Console.WriteLine("A's destructor");  
 }  
}

class B: A  
{  
 ~B() {  
 Console.WriteLine("B's destructor");  
 }  
}

class Test  
{  
 static void Main() {  
 B b = new B();  
 b = null;  
 GC.Collect();  
 GC.WaitForPendingFinalizers();  
 }  
}

ist

B’s destructor  
A’s destructor

weil die Destruktoren in der Vererbungskette der Reihe nach aufgerufen werden – angefangen bei dem am meisten abgeleiteten Destruktor bis hin zum am wenigsten abgeleiteten Destruktor.

Destruktoren werden implementiert, indem die virtuelle Methode Finalize für System.Object überschrieben wird. Für C#-Programme ist es nicht zulässig, diese Methode zu überschreiben oder sie (bzw. überschriebene Methoden davon) direkt aufzurufen. Beispiel: Das Programm

class A   
{  
 override protected void Finalize() {} // error

public void F() {  
 this.Finalize(); // error  
 }  
}

enthält zwei Fehler.

Der Compiler verhält sich, als ob diese Methode und ihre überschreibenden Methoden überhaupt nicht vorhanden wären. Somit ist das nachstehende Programm gültig:

class A   
{  
 void Finalize() {} // permitted  
}

Die aufgeführte Methode verdeckt die Finalize-Methode von System.Object.

In §16.3 wird das Verhalten näher erläutert, das sich ergibt, wenn eine Ausnahme von einem Destruktor ausgelöst wird.

## Iteratoren

Ein Funktionsmember (§7.5), der mit einem Iteratorblock (§8.2) implementiert wurde, wird als Iterator bezeichnet.

Ein Iteratorblock kann als Text eines Funktionsmembers verwendet werden, sofern der Rückgabetyp des entsprechenden Funktionsmembers eine der Enumeratorschnittstellen (§10.14.1) oder eine der aufzählbaren Schnittstellen (§10.14.2) ist. Er kann als method-body, operator-body oder accessor-body auftreten, während Ereignisse, Instanzkonstruktoren, statische Konstuktoren und Destruktoren nicht als Iteratoren implementiert werden können.

Wenn ein Funktionsmember mithilfe eines Iteratorblocks implementiert wird, führt die Angabe von ref-Parametern oder out-Parametern zu einem Kompilierungsfehler für die Liste formaler Parameter des Funktionsmembers.

### Enumeratorschnittstellen

Enumeratorschnittstellen sind nicht generische Schnittstellen vom Typ System.Collections.IEnumerator sowie alle Instanziierungen der generischen Schnittstelle System.Collections.Generic.IEnumerator<T>. Aus Effizienzgründen werden diese Schnittstellen in diesem Kapitel als IEnumerator bzw. IEnumerator<T> bezeichnet.

### Aufzählbare Schnittstellen

Aufzählbare Schnittstellen sind nicht generische Schnittstellen vom Typ System.Collections.IEnumerable sowie alle Instanziierungen der generischen Schnittstelle System.Collections.Generic.IEnumerable<T>. Aus Effizienzgründen werden diese Schnittstellen in diesem Kapitel als IEnumerable bzw. IEnumerable<T> bezeichnet.

### Ausgabetyp

Ein Iterator generiert eine Folge von Werten, die alle von demselben Typ sind. Dieser Typ wird als Ausgabetyp des Iterators bezeichnet.

* Der Ausgabetyp eines Iterators, der IEnumerator oder IEnumerable zurückgibt, ist object.
* Der Ausgabetyp eines Iterators, der IEnumerator<T> oder IEnumerable<T> zurückgibt, lautet T.

### Enumeratorobjekte

Wenn ein Funktionsmember, der eine Enumeratorschnittstelle zurückgibt, mithilfe eines Iteratorblocks implementiert wird, wird beim Aufruf des Funktionsmembers nicht sofort der Code im Iteratorblock ausgeführt. Stattdessen wird ein Enumeratorobjekt erstellt und zurückgegeben. Dieses Objekt kapselt den im Iteratorblock angegebenen Code. Der Code im Iteratorblock wird dann ausgeführt, wenn die MoveNext-Methode des Enumeratorobjekts aufgerufen wird. Ein Enumeratorobjekt weist die folgenden Merkmale auf:

* Es implementiert IEnumerator und IEnumerator<T>, wobei T der Ausgabetyp des Iterators ist.
* Er implementiert System.IDisposable.
* Es wird mit einer Kopie des Argumentwerts (sofern vorhanden) und einem an den Funktionsmember übergebenen Instanzwert initialisiert.
* Es hat vier mögliche Zustände: vorher, ausgeführt, unterbrochen und nachher, wobei der Anfangszustand vorher lautet.

Ein Enumeratorobjekt ist normalerweise eine Instanz einer vom Compiler generierten Enumeratorklasse, die den Code im Iteratorblock kapselt und die Enumeratorschnittstellen implementiert, wobei aber auch andere Arten der Implementierung möglich sind. Wenn eine Enumeratorklasse vom Compiler erstellt wird, wird diese direkt oder indirekt in der Klasse mit dem Funktionsmember geschachtelt. Sie verfügt dann über privaten Zugriff, und es wird ein Name für die Verwendung durch den Compiler reserviert (§2.4.2).

Ein Enumeratorobjekt kann neben den oben beschriebenen Schnittstellen weitere Schnittstellen implementieren.

In den folgenden Abschnitten wird das genaue Verhalten des MoveNext-Members, des Current-Members und des Dispose-Members der durch ein Enumeratorobjekt bereitgestellten IEnumerable-Schnittstelle und IEnumerable<T>-Schnittstelle beschrieben.

Beachten Sie, dass Enumeratorobjekte die IEnumerator.Reset-Methode nicht unterstützen. Wenn Sie diese Methode aufrufen, wird eine System.NotSupportedException ausgelöst.

#### Die MoveNext-Methode

Die MoveNext-Methode eines Enumeratorobjekts kapselt den Code eines Iteratorblocks. Durch Aufrufen der MoveNext-Methode wird der Code im Iteratorblock ausgeführt, und die Current-Eigenschaft des Enumeratorobjekts wird entsprechend festgelegt. Die tatsächlich von MoveNext ausgeführte Aktion ist vom Zustand des Enumeratorobjekts beim Aufruf von MoveNext abhängig:

* Wenn der Zustand des Enumeratorobjekts beim Aufruf von MoveNext before lautet:
* Der Zustand wird in ausgeführt geändert.
* Die Parameter (einschließlich this) des Iteratorblocks werden mit den beim Initialisieren des Enumeratorobjekts gespeicherten Argument- und Instanzwerten initialisiert.
* Der Iteratorblock wird vom Anfang bis zur Unterbrechung ausgeführt (siehe unten).
* Wenn der Zustand des Enumeratorobjekts ausgeführt lautet, ist das Ergebnis eines Aufrufs von MoveNext nicht festgelegt.
* Wenn der Zustand des Enumeratorobjekts beim Aufruf von MoveNext unterbrochen lautet:
* Der Zustand wird in ausgeführt geändert.
* Die Werte aller lokalen Variablen und Parameter (einschließlich this) werden auf die beim letzten Unterbrechen der Ausführung des Iteratorblocks gespeicherten Werte zurückgesetzt. Beachten Sie, dass der Inhalt aller Objekte, auf die diese Variablen verweisen, seit dem letzten Aufruf von MoveNext möglicherweise geändert wurde.
* Die Ausführung des Iteratorblocks wird direkt nach der yield return-Anweisung, die den Abbruch der Ausführung verursacht hat, bis zur Unterbrechung fortgesetzt (siehe unten).
* Wenn der Zustand des Enumeratorobjekts beim Aufruf von MoveNext nachher lautet, wird false zurückgegeben:

Wenn MoveNext den Iteratorblock ausführt, kann die Ausführung auf vier Arten unterbrochen werden: Durch eine yield return-Anweisung, durch eine yield break-Anweisung, durch das Ende des Iteratorblocks und durch eine ausgelöste Ausnahme, die aus dem Iteratorblock heraus weitergeleitet wird.

* Bei einer yield return-Anweisung (§8.14):
* Der in der Anweisung angegebene Ausdruck wird ausgewertet, implizit in den Ausgabetyp umgewandelt und der Current-Eigenschaft des Enumeratorobjekts zugewiesen.
* Die Ausführung des Iteratortexts wird unterbrochen. Die Werte aller lokalen Variablen und Parameter (einschließlich this) werden wie auch die Position dieser yield return-Anweisung gespeichert. Wenn sich die yield return-Anweisung innerhalb eines oder mehrerer try-Blöcke befindet, werden die zugehörigen finally-Blöcke zu diesem Zeitpunkt nicht ausgeführt.
* Der Zustand des Enumeratorobjekts wird in unterbrochen geändert.
* Die MoveNext-Methode gibt true an den Aufrufer zurück, wodurch angegeben wird, dass die Iteration erfolgreich auf den nächsten Wert erhöht wurde.
* Bei einer yield break-Anweisung (§8.14):
* Wenn sich die yield break-Anweisung innerhalb eines oder mehrerer try-Blöcke befindet, werden die zugehörigen finally-Blöcke ausgeführt.
* Der Zustand des Enumeratorobjekts wird in nachher geändert.
* Die MoveNext-Methode gibt false an den Aufrufer zurück, wodurch der Abschluss der Iteration angegeben wird.
* Wenn das Ende des Iteratortexts erreicht wird:
* Der Zustand des Enumeratorobjekts wird in nachher geändert.
* Die MoveNext-Methode gibt false an den Aufrufer zurück, wodurch der Abschluss der Iteration angegeben wird.
* Wenn eine Ausnahme ausgelöst und aus dem Iteratorblock heraus weitergeleitet wird:
* Während der Weiterleitung der Ausnahme wurden die entsprechenden finally-Blöcke im Iteratortext ausgeführt.
* Der Zustand des Enumeratorobjekts wird in nachher geändert.
* Die Ausnahme wird an den Aufrufer der MoveNext-Methode weitergeleitet.

#### Die Current-Eigenschaft

Die Current-Eigenschaft eines Enumeratorobjekts wird durch die yield return-Anweisungen im Iteratorblock beeinflusst.

Wenn ein Enumeratorobjekt den Zustand unterbrochen aufweist, ist der Wert von Current der vor dem Aufruf von MoveNext festgelegte Wert. Wenn ein Enumeratorobjekt den Zustand vorher, ausgeführt oder nachher aufweist, ist das Ergebnis eines Zugriffs auf Current nicht festgelegt.

Bei einem Iterator mit einem anderen Ausgabetyp als object entspricht das Ergebnis eines Zugriffs auf Current über die IEnumerable-Implementierung des Enumeratorobjekts dem beim Zugriff auf Current über die IEnumerator<T>-Implementierung des Enumeratorobjekts und dem Umwandeln des Ergebnisses in den Typ object.

#### Die Dispose-Methode

Die Dispose-Methode wird zum Bereinigen der Iteration verwendet, indem das Enumeratorobjekt in den Zustand nachher geändert wird.

* Wenn der Zustand des Enumeratorobjekts vorher lautet, wird dieser durch Aufruf von Dispose in nachher geändert.
* Wenn der Zustand des Enumeratorobjekts ausgeführt lautet, ist das Ergebnis eines Aufrufs von Dispose nicht festgelegt.
* Wenn der Zustand des Enumeratorobjekts beim Aufruf von Dispose unterbrochen lautet:
* Der Zustand wird in ausgeführt geändert.
* Es werden alle finally-Blöcke ausgeführt, sofern die zuletzt ausgeführte yield return-Anweisung eine yield break-Anweisung war. Wenn dadurch eine Ausnahme ausgelöst wird, die aus dem Iteratortext heraus weitergeleitet wird, wird der Zustand des Enumeratorobjekts auf nachher festgelegt, und die Ausnahme wird an den Aufrufer der Dispose-Methode weitergeleitet.
* Der Zustand wird in nachher geändert.
* Wenn der Zustand des Enumeratorobjekts nachher lautet, hat ein Aufruf von Dispose keine Auswirkungen.

### Aufzählbare Objekte

Wenn ein Funktionsmember, der eine aufzählbare Schnittstelle zurückgibt, mithilfe eines Iteratorblocks implementiert wird, wird der Code im Iteratorblock nicht sofort beim Aufruf des Funktionsmembers ausgeführt. Stattdessen wird ein aufzählbares Objekt erstellt und zurückgegeben. Die GetEnumerator-Methode des aufzählbaren Objekts gibt ein Enumeratorobjekt zurück, das den im Iteratorblock angegebenen Code kapselt. Die Ausführung des Codes im Iteratorblock erfolgt beim Aufruf der MoveNext-Methode des Enumeratorobjekts. Ein aufzählbares Objekt weist die folgenden Merkmale auf:

* Es implementiert IEnumerable und IEnumerable<T>, wobei T der Ausgabetyp des Iterators ist.
* Es wird mit einer Kopie des Argumentwerts (sofern vorhanden) und einem an den Funktionsmember übergebenen Instanzwert initialisiert.

Ein aufzählbares Objekt ist normalerweise eine Instanz einer vom Compiler generierten aufzählbaren Klasse, die den Code im Iteratorblock kapselt und die aufzählbare Schnittstellen implementiert, wobei aber auch andere Arten der Implementierung möglich sind. Wenn eine aufzählbare Klasse vom Compiler erstellt wird, wird diese direkt oder indirekt in der Klasse mit dem Funktionsmember geschachtelt. Sie verfügt dann über privaten Zugriff, und es wird ein Name für die Verwendung durch den Compiler reserviert (§2.4.2).

Ein aufzählbares Objekt kann neben den oben beschriebenen Schnittstellen weitere Schnittstellen implementieren. Ein aufzählbares Objekt kann insbesondere auch IEnumerator und IEnumerator<T> implementieren, wodurch es aufzählbar und als Enumerator verwendet werden kann. Bei dieser Art der Implementierung wird beim ersten Aufruf der GetEnumerator-Methode eines aufzählbaren Objekts das aufzählbare Objekt selbst zurückgegeben. Bei nachfolgenden Aufrufen von GetEnumerator im aufzählbaren Objekt wird eine Kopie des aufzählbaren Objekts zurückgegeben. Daher verfügt jeder zurückgegebene Enumerator über einen eigenen Zustand, und Änderungen an einem Enumerator haben keinen Einfluss auf die anderen Enumeratoren.

#### Die GetEnumerator-Methode

Ein aufzählbares Objekt stellt eine Implementierung der GetEnumerator Methoden der IEnumerable-Schnittstelle und der IEnumerable<T>-Schnittstelle bereit. Die beiden GetEnumerator-Methoden verfügen über eine gemeinsame Implementierung, die ein verfügbares Enumeratorobjekt abfragt und zurückgibt. Das Enumeratorobjekt wird mit den beim Initialisieren des aufzählbaren Objekts gespeicherten Argument- und Instanzwerten initialisiert, wobei das Enumeratorobjekt sich ansonsten wie in §10.14.4 verhält.

### Implementierungsbeispiel

In diesem Abschnitt wird eine mögliche Implementierung von Iteratoren mithilfe von C#-Standardkonstrukten beschrieben. Die hier erläuterte Implementierung beruht auf denselben Grundsätzen wie den vom Microsoft C#-Compiler verwendeten, sie stellt jedoch unter keinen Umständen eine zwingend erforderliche oder die einzig mögliche Implementierung dar.

Die folgende Stack<T>-Klasse implementiert die eigene GetEnumerator-Methode mit einem Iterator. Der Iterator enumeriert die Elemente des Stapels von oben nach unten.

using System;  
using System.Collections;  
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>  
{  
 T[] items;  
 int count;

public void Push(T item) {  
 if (items == null) {  
 items = new T[4];  
 }  
 else if (items.Length == count) {  
 T[] newItems = new T[count \* 2];  
 Array.Copy(items, 0, newItems, 0, count);  
 items = newItems;  
 }  
 items[count++] = item;  
 }

public T Pop() {  
 T result = items[--count];  
 items[count] = default(T);  
 return result;  
 }

public IEnumerator<T> GetEnumerator() {  
 for (int i = count - 1; i >= 0; --i) yield return items[i];  
 }  
}

Die GetEnumerator-Methode kann in eine Instanziierung einer vom Compiler generierten Enumeratorklasse übersetzt werden, die den Code des Iteratorblocks kapselt. Dies wird im Folgenden veranschaulicht.

class Stack<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1: IEnumerator<T>, IEnumerator  
 {  
 int \_\_state;  
 T \_\_current;  
 Stack<T> \_\_this;  
 int i;

public \_\_Enumerator1(Stack<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1: goto \_\_state1;  
 case 2: goto \_\_state2;  
 }  
 i = \_\_this.count - 1;  
 \_\_loop:  
 if (i < 0) goto \_\_state2;  
 \_\_current = \_\_this.items[i];  
 \_\_state = 1;  
 return true;  
 \_\_state1:  
 --i;  
 goto \_\_loop;  
 \_\_state2:  
 \_\_state = 2;  
 return false;  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

In der vorherigen Übersetzung wurde der Code im Iteratorblock in einen Zustandsautomaten umgewandelt und in die MoveNext-Methode der Enumeratorklasse eingefügt. Außerdem wurde die lokale Variable i in ein Feld des Enumeratorobjekts umgewandelt, damit sie über mehrere Aufrufe von MoveNextt beibehalten werden kann.

Im folgenden Beispiel wird eine einfache Multiplikationstabelle der ganzen Zahlen von 1 bis 10 ausgegeben. Die FromTo-Methode im Beispiel gibt ein aufzählbares Objekt zurück und wird mithilfe eines Iterators implementiert.

using System;  
using System.Collections.Generic;

class Test  
{  
 static IEnumerable<int> FromTo(int from, int to) {  
 while (from <= to) yield return from++;  
 }

static void Main() {  
 IEnumerable<int> e = FromTo(1, 10);  
 foreach (int x in e) {  
 foreach (int y in e) {  
 Console.Write("{0,3} ", x \* y);  
 }  
 Console.WriteLine();  
 }  
 }  
}

Die FromTo-Methode kann in eine Instanziierung einer vom Compiler generierten aufzählbaren Klasse übersetzt werden, die den Code des Iteratorblocks kapselt. Dies wird im Folgenden veranschaulicht.

using System;  
using System.Threading;  
using System.Collections;  
using System.Collections.Generic;

class Test  
{  
 ...

static IEnumerable<int> FromTo(int from, int to) {  
 return new \_\_Enumerable1(from, to);  
 }

class \_\_Enumerable1:  
 IEnumerable<int>, IEnumerable,  
 IEnumerator<int>, IEnumerator  
 {  
 int \_\_state;  
 int \_\_current;  
 int \_\_from;  
 int from;  
 int to;  
 int i;

public \_\_Enumerable1(int \_\_from, int to) {  
 this.\_\_from = \_\_from;  
 this.to = to;  
 }

public IEnumerator<int> GetEnumerator() {  
 \_\_Enumerable1 result = this;  
 if (Interlocked.CompareExchange(ref \_\_state, 1, 0) != 0) {  
 result = new \_\_Enumerable1(\_\_from, to);  
 result.\_\_state = 1;  
 }  
 result.from = result.\_\_from;  
 return result;  
 }

IEnumerator IEnumerable.GetEnumerator() {  
 return (IEnumerator)GetEnumerator();  
 }

public int Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 switch (\_\_state) {  
 case 1:  
 if (from > to) goto case 2;  
 \_\_current = from++;  
 \_\_state = 1;  
 return true;  
 case 2:  
 \_\_state = 2;  
 return false;  
 default:  
 throw new InvalidOperationException();  
 }  
 }

public void Dispose() {  
 \_\_state = 2;  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

Die aufzählbare Klasse implementiert sowohl die aufzählbare Schnittstelle als auch die Enumeratorschnittstelle, wodurch sie aufzählbar und als Enumerator verwendet werden kann. Beim ersten Aufruf der GetEnumerator-Methode wird das aufzählbare Objekt selbst zurückgegeben. Bei nachfolgenden Aufrufen von GetEnumerator im aufzählbaren Objekt wird eine Kopie des aufzählbaren Objekts zurückgegeben. Daher verfügt jeder zurückgegebene Enumerator über einen eigenen Zustand, und Änderungen an einem Enumerator haben keinen Einfluss auf die anderen Enumeratoren. Mit der Interlocked.CompareExchange-Methode wird die threadsichere Ausführung sichergestellt.

Der from-Parameter und der to-Parameter werden in der aufzählbaren Klasse in Felder umgewandelt. Da from im Iteratorblock geändert wird, wird ein zusätzliches \_\_from-Feld eingefügt, das den Anfangswert von from in jedem Enumerator enthält.

Die MoveNext-Methode löst eine InvalidOperationException aus, wenn sie aufgerufen wird und \_\_state den Wert 0 hat. Dies verhindert die Verwendung des aufzählbaren Objekts als Enumeratorobjekt ohne vorherigen Aufruf von GetEnumerator.

Im folgenden Beispiel wird eine einfache Strukturklasse veranschaulicht. Die Tree<T>-Klasse implementiert die eigene GetEnumerator-Methode mit einem Iterator. Der Iterator listet die Elemente der Struktur in Infix-Reihenfolge auf.

using System;  
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>  
{  
 T value;  
 Tree<T> left;  
 Tree<T> right;

public Tree(T value, Tree<T> left, Tree<T> right) {  
 this.value = value;  
 this.left = left;  
 this.right = right;  
 }

public IEnumerator<T> GetEnumerator() {  
 if (left != null) foreach (T x in left) yield x;  
 yield value;  
 if (right != null) foreach (T x in right) yield x;  
 }  
}

class Program  
{  
 static Tree<T> MakeTree<T>(T[] items, int left, int right) {  
 if (left > right) return null;  
 int i = (left + right) / 2;  
 return new Tree<T>(items[i],   
 MakeTree(items, left, i - 1),  
 MakeTree(items, i + 1, right));  
 }

static Tree<T> MakeTree<T>(params T[] items) {  
 return MakeTree(items, 0, items.Length - 1);  
 }

// The output of the program is:  
 // 1 2 3 4 5 6 7 8 9  
 // Mon Tue Wed Thu Fri Sat Sun

static void Main() {  
 Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);  
 foreach (int i in ints) Console.Write("{0} ", i);  
 Console.WriteLine();

Tree<string> strings = MakeTree(  
 "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");  
 foreach (string s in strings) Console.Write("{0} ", s);  
 Console.WriteLine();  
 }  
}

Die GetEnumerator-Methode kann in eine Instanziierung einer vom Compiler generierten Enumeratorklasse übersetzt werden, die den Code des Iteratorblocks kapselt. Dies wird im Folgenden veranschaulicht.

class Tree<T>: IEnumerable<T>  
{  
 ...

public IEnumerator<T> GetEnumerator() {  
 return new \_\_Enumerator1(this);  
 }

class \_\_Enumerator1 : IEnumerator<T>, IEnumerator  
 {  
 Node<T> \_\_this;  
 IEnumerator<T> \_\_left, \_\_right;  
 int \_\_state;  
 T \_\_current;

public \_\_Enumerator1(Node<T> \_\_this) {  
 this.\_\_this = \_\_this;  
 }

public T Current {  
 get { return \_\_current; }  
 }

object IEnumerator.Current {  
 get { return \_\_current; }  
 }

public bool MoveNext() {  
 try {  
 switch (\_\_state) {

case 0:  
 \_\_state = -1;  
 if (\_\_this.left == null) goto \_\_yield\_value;  
 \_\_left = \_\_this.left.GetEnumerator();  
 goto case 1;

case 1:  
 \_\_state = -2;  
 if (!\_\_left.MoveNext()) goto \_\_left\_dispose;  
 \_\_current = \_\_left.Current;  
 \_\_state = 1;  
 return true;

\_\_left\_dispose:  
 \_\_state = -1;  
 \_\_left.Dispose();

\_\_yield\_value:  
 \_\_current = \_\_this.value;  
 \_\_state = 2;  
 return true;

case 2:  
 \_\_state = -1;  
 if (\_\_this.right == null) goto \_\_end;  
 \_\_right = \_\_this.right.GetEnumerator();  
 goto case 3;

case 3:  
 \_\_state = -3;  
 if (!\_\_right.MoveNext()) goto \_\_right\_dispose;  
 \_\_current = \_\_right.Current;  
 \_\_state = 3;  
 return true;

\_\_right\_dispose:  
 \_\_state = -1;  
 \_\_right.Dispose();

\_\_end:  
 \_\_state = 4;  
 break;

}  
 }  
 finally {  
 if (\_\_state < 0) Dispose();  
 }  
 return false;  
 }

public void Dispose() {  
 try {  
 switch (\_\_state) {

case 1:  
 case -2:  
 \_\_left.Dispose();  
 break;

case 3:  
 case -3:  
 \_\_right.Dispose();  
 break;

}  
 }  
 finally {  
 \_\_state = 4;  
 }  
 }

void IEnumerator.Reset() {  
 throw new NotSupportedException();  
 }  
 }  
}

Die vom Compiler generierten temporären Werte, die in den foreach-Anweisungen verwendet werden, werden in das \_\_left-Feld und das \_\_right-Feld des Enumeratorobjekts geschrieben. Das \_\_state-Feld des Enumeratorobjekts wird sorgfältig aktualisiert, sodass beim Auslösen einer Ausnahme die richtige Dispose()-Methode ordnungsgemäß aufgerufen wird. Beachten Sie, dass es nicht möglich ist, den übersetzten Code mit einfachen foreach-Anweisungen zu schreiben.

## Async-Funktionen

Eine Methode (§10.6) oder anonyme Funktion (§7.15) mit dem async-Modifizierer wird als Async-Funktion bezeichnet. Im Allgemeinen wird mit dem Begriff async jede Art von Funktion beschrieben, die den async-Modifizierer aufweist.

Wenn in der Liste formaler Parameter einer Async-Funktion ref- oder out-Parameter angegeben werden, tritt ein Kompilierungsfehler auf.

Der return-type einer Async-Methode muss entweder void oder ein Aufgabentyp sein. Die Aufgabentypen lauten System.Threading.Tasks.Task und Typen, die aus System.Threading.Tasks.Task<T> konstruiert wurden. Aus Effizienzgründen werden diese Typen in diesem Kapitel als Task bzw. Task<T> bezeichnet. Eine Async-Methode, die einen Aufgabentyp zurückgibt, wird als "Aufgabe zurückgebend" bezeichnet.

Die genaue Definition der Aufgabentypen wird durch die Implementierung definiert. Aus Sicht der Sprache befindet sich ein Aufgabentyp jedoch in einem der folgenden Zustände: unvollständig, erfolgreich oder fehlerhaft. Eine fehlerhafte Aufgabe zeichnet eine entsprechende Ausnahme auf. Eine erfolgreiche Task<*T*> zeichnet ein Ergebnis vom Typ *T* auf. Aufgabentypen sind "awaitable" und können somit Operanden von Await-Ausdrücken sein (§7.7.7).

Eine Async-Funktion weist die Fähigkeit auf, eine Auswertung mittels Await-Ausdrücken (§7.7.7) im Text anzuhalten. Die Auswertung kann später am Punkt des entsprechenden Await-Ausdrucks mittels eines Wiederaufnahmedelegaten fortgesetzt werden. Der Wiederaufnahmedelegat ist vom Typ System.Action, und bei seinem Aufruf wird die Auswertung des Async-Funktionsaufrufs ab dem Await-Ausdruck fortgesetzt, wo sie angehalten wurde. Der aktuelle Aufrufer eines Async-Funktionsaufrufs ist der ursprüngliche Aufrufer, wenn der Funktionsaufruf niemals angehalten wurde, oder andernfalls der letzte Aufrufer des Wiederaufnahmedelegaten.

### Auswertung einer Async-Funktion, die eine Aufgabe zurückgibt

Beim Aufruf einer Async-Funktion, die eine Aufgabe zurückgibt, wird eine Instanz des zurückgegebenen Aufgabentyps generiert. Diese wird als Rückgabeaufgabe der Async-Funktion bezeichnet. Die Aufgabe befindet sich zu Beginn im Zustand unvollständig.

Der Async-Funktionstext wird dann ausgewertet, bis er angehalten (durch Erreichen eines Await-Ausdrucks) oder beendet wird. An diesem Punkt wird die Steuerung an den Aufrufer zusammen mit der Rückgabeaufgabe zurückgegeben.

Beim Beenden des Texts der Async-Funktion wird der unvollständige Zustand der Rückgabeaufgabe aufgehoben:

* Wenn der Funktionstext durch Erreichen einer Rückgabeanweisung oder des Textendes beendet wird, werden alle Ergebniswerte in der Rückgabeaufgabe aufgezeichnet, die in den Zustand erfolgreich versetzt wird.
* Wenn der Funktionstext durch eine nicht abgefangene Ausnahme (§8.9.5) beendet wird, wird die Ausnahme in der Rückgabeaufgabe aufgezeichnet, die in den Zustand fehlerhaft versetzt wird.

### Auswertung einer Async-Funktion, die "void" zurückgibt

Wenn der Rückgabetyp der Async-Funktion void ist, weicht die Auswertung folgendermaßen ab: Da keine Aufgabe zurückgegeben wird, sendet die Funktion den Abschluss und die Ausnahmen stattdessen an den Synchronisierungskontext des aktuellen Threads. Die genaue Definition des Synchronisierungskontexts ist von der Implementierung abhängig, es handelt sich jedoch um eine Darstellung davon, "wo" der aktuelle Thread ausgeführt wird. Der Synchronisierungskontext wird benachrichtigt, wenn die Auswertung einer Async-Funktion, die "void" zurückgibt, beginnt, erfolgreich abgeschlossen wird oder eine nicht abgefangene Ausnahme auslöst.

Dadurch kann der Kontext nachverfolgen, wie viele Async-Funktionen, die "void" zurückgeben, darunter ausgeführt werden und entscheiden, wie die daher stammenden Ausnahmen weitergegeben werden.

# Strukturen

Strukturen stellen ähnlich wie Klassen Datenstrukturen dar, die Datenmember und Funktionsmember enthalten können. Im Gegensatz zu Klassen handelt es sich bei Strukturen jedoch um Werttypen, die keine Heapzuordnung benötigen. Eine Variable eines Strukturtyps enthält direkt die Daten der Struktur, während eine Variable eines Klassentyps einen Verweis auf die Daten enthält. Dieser Verweis wird als Objekt bezeichnet.

Strukturen eignen sich besonders für kleine Datenstrukturen, die über Wertsemantik verfügen. Geeignete Beispiele für Strukturen sind komplexe Zahlen, Punkte in einem Koordinatensystem und Schlüsselwertpaare in einem Wörterbuch. Das Entscheidende bei diesen Datenstrukturen ist, dass sie über eine geringe Anzahl von Datenmembern verfügen, dass die Verwendung von Vererbung und referenzieller Identität nicht erforderlich ist und dass sie mithilfe der Wertsemantik einfach durch Zuweisen des Wertes (anstelle des Verweises) implementiert werden können.

Bei den einfachen Typen in C# (z. B. int, double und bool) handelt es sich um Strukturtypen (siehe §4.1.4). Da es sich bei diesen vordefinierten Typen um Strukturen handelt, können mithilfe von Strukturen und Operatorüberladungen neue „primitive“ Typen in der Sprache C# implementiert werden. Am Ende dieses Kapitels werden zwei Beispiele für diese Typen beschrieben (§11.4).

## Strukturdeklarationen

Eine struct-declaration ist eine type-declaration (§9.6), die eine neue Struktur deklariert:

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

Eine struct-declaration besteht aus einer optionalen Gruppe von attributes (§17) gefolgt von einer optionalen Gruppe von struct-modifiers (§11.1.1), einem optionalen partial-Modifizierer, dem Schlüsselwort struct und einem identifier, der die Struktur benennt. Darauf folgt eine optionale type-parameter-list-Spezifikation (§10.1.3), eine optionale struct-interfaces-Spezifikation (§11.1.2), eine optionale type-parameters-constraints-clauses-Spezifikation (§10.1.5) und ein struct-body (§11.1.4), an den sich optional ein Semikolon anschließt.

### Strukturmodifizierer

Eine struct-declaration kann optional eine Reihe von Strukturmodifizierern enthalten:

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

Wenn derselbe Modifizierer mehrmals in einer Strukturdeklaration aufgeführt ist, wird ein Kompilierungsfehler verursacht.

Die Modifizierer einer Strukturdeklaration haben dieselbe Bedeutung wie die einer Klassendeklaration (§10.1).

### Partial-Modifizierer

Der partial -Modifizierer gibt an, dass diese struct-declaration eine partielle Typdeklaration ist. Mehrere partielle Strukturdeklarationen mit demselben Namen in einem einschließenden Namespace oder einer einschließenden Typdeklaration bilden zusammen eine Strukturdeklaration. Dabei gelten die in §10.2 beschriebenen Regeln.

### Strukturschnittstellen

Unter bestimmten Umständen kann eine Strukturdeklaration eine struct-interfaces-Spezifikation enthalten. Wenn dies der Fall ist, werden die angegebenen Schnittstellentypen von der Struktur direkt implementiert.

struct-interfaces:  
: interface-type-list

Schnittstellenimplementierungen werden in §13.4 ausführlicher beschrieben.

### Strukturkörper

Im struct-body einer Struktur werden die Member der Struktur definiert.

struct-body:  
{ struct-member-declarationsopt }

## Strukturmember

Die Member einer Struktur bestehen einerseits aus den Membern, die von ihren struct-member-declarations eingeführt wurden, und andererseits aus den Membern, die vom Typ System.ValueType geerbt wurden.

struct-member-declarations:  
struct-member-declaration  
struct-member-declarations struct-member-declaration

struct-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
static-constructor-declaration  
type-declaration

Abgesehen von den in §11.3 beschriebenen Unterschieden treffen die in §10.3 bis §10.14 aufgeführten Beschreibungen zu Klassenmembern auch auf Strukturmember zu.

## Unterschiede zwischen Klassen und Strukturen

Zwischen Strukturen und Klassen bestehen einige wichtige Unterschiede:

* Strukturen sind Werttypen (§11.3.1).
* Alle Strukturtypen erben implizit von der System.ValueType-Klasse (§11.3.2).
* Durch die Zuweisung zu einer Variablen eines Strukturtyps wird eine Kopie des zugewiesenen Wertes erstellt (§11.3.3).
* Der Standardwert für eine Struktur wird erzeugt, indem für alle Werttypfelder der Standardwert und für alle Verweistypfelder null festgelegt wird (§11.3.4).
* Zur Konvertierung zwischen einem Strukturtyp und object werden Boxing- und Unboxingoperationen verwendet (§11.3.5).
* Im Kontext von Strukturen hat this eine andere Bedeutung (§7.6.7).
* Instanzfelddeklarationen für eine Struktur dürfen keine Variableninitialisierungen enthalten (§11.3.7).
* Eine Struktur darf keine parameterlosen Instanzkonstruktoren deklarieren (§11.3.8).
* Eine Struktur darf keinen Destruktor deklarieren (§11.3.9).

### Wertsemantik

Strukturen sind Werttypen (§4.1), die der Wertsemantik folgen. Klassen hingegen sind Verweistypen (§4.2), die der Verweissemantik folgen.

Eine Variable eines Strukturtyps enthält direkt die Daten der Struktur, während eine Variable eines Klassentyps einen Verweis auf die Daten enthält. Dieser Verweis wird als Objekt bezeichnet. Wenn die Struktur B ein Instanzfeld vom Typ A enthält und A ein Strukturtyp ist, wird ein Kompilierungsfehler ausgelöst, wenn A von B oder von einem aus B konstruierten Typ abhängig ist. Die Struktur X ist direkt abhängig von der Struktur Y, wenn X ein Instanzfeld vom Typ Y enthält. Aufgrund dieser Definition ist die vollständige Reihe von Strukturen, von der eine Struktur abhängt, der transitive Schluss der Beziehung der direkten Abhängigkeit. Beispiel:

struct Node  
{  
 int data;

Node next; // error, Node directly depends on itself

}

führt zu einem Fehler, weil Node ein Instanzfeld seines eigenen Typs enthält. Auch das Beispiel

struct A { B b; }

struct B { C c; }

struct C { A a; }

führt zu einem Fehler, weil die Typen A, B und C voneinander abhängig sind.

Bei Klassen besteht die Möglichkeit, dass zwei Variablen auf dasselbe Objekt verweisen. Somit ist es möglich, dass die eine Variable betreffenden Operationen auch Auswirkungen auf das von der anderen Variablen referenzierte Objekt haben. Bei Strukturen besitzt jede Variable ihre eigene Kopie der Daten (ausgenommen Variablen mit dem Parameter ref und out), sodass gegenseitige Beeinflussungen ausgeschlossen sind. Da Strukturen keine Verweistypen sind, können Werte eines Strukturtyps außerdem nicht null sein.

Bei der Deklaration

struct Point  
{  
 public int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

gibt das Codefragment

Point a = new Point(10, 10);  
Point b = a;  
a.x = 100;  
System.Console.WriteLine(b.x);

den Wert 10 aus. Anhand der Zuweisung von a zu b wird eine Kopie des Werts erstellt, sodass b von der Zuweisung zu a.x nicht betroffen ist. Wenn Point hingegen eine Klasse deklariert, wird der Wert 100 ausgegeben, weil a und b auf dasselbe Objekt verweisen.

### Vererbung

Alle Strukturtypen erben implizit von der System.ValueType-Klasse, die ihrerseits von der object-Klasse erbt. In einer Strukturdeklaration kann eine Liste der implementierten Schnittstellen, aber keine Basisklasse angegeben werden.

Strukturtypen sind nie abstrakt und immer implizit versiegelt. Der abstract-Modifizierer und der sealed-Modifizierer sind daher in Strukturdeklarationen nicht zulässig.

Da die Vererbung für Strukturen nicht unterstützt wird, kann der deklarierte Zugriff eines Strukturmembers weder protected noch protected internal sein.

In einer Struktur enthaltene Funktionsmember dürfen weder abstract noch virtual sein. Der override-Modifizierer darf nur Methoden überschreiben, die von System.ValueType geerbt wurden.

### Zuweisung

Durch die Zuweisung zu einer Variablen eines Strukturtyps wird eine Kopie des zugewiesenen Wertes erstellt. Bei der Zuweisung zu einer Variablen eines Klassentyps wird dagegen nur der Verweis und nicht das durch den Verweis gekennzeichnete Objekt kopiert.

Ähnlich wie bei einer Zuweisung wird beim Übergeben einer Struktur als Werteparameter oder bei der Rückgabe als Ergebnis eines Funktionsmembers eine Kopie der Struktur erstellt. Eine Struktur kann mithilfe eines ref-Parameters oder eines out-Parameters durch Verweis auf einen Funktionsmember übergeben werden.

Wenn eine Eigenschaft oder ein Indexer einer Struktur das Ziel einer Zuweisung ist, muss der Instanzausdruck, der mit der Eigenschaft oder dem Indexer verknüpft ist, als Variable klassifiziert werden. Wenn der Instanzausdruck als Wert klassifiziert ist, tritt während der Kompilierung ein Fehler auf. Dies wird in §7.17.1 näher erläutert.

### Standardwerte

Wie in §5.2 beschrieben, werden mehrere Variablenarten beim Erstellen automatisch mit ihren Standardwerten initialisiert. Bei Variablen von Klassentypen und anderen Verweistypen lautet dieser Standardwert null. Da es sich bei Strukturen um Werttypen handelt, die nicht null sein können, ist der Standardwert einer Struktur der Wert, der erzeugt wird, wenn für alle Werttypfelder der Standardwert und für alle Verweistypfelder null festgelegt wird.

Bezüglich der weiter oben deklarierten Point-Struktur wird im Beispiel

Point[] a = new Point[100];

jeder Point in dem Array mit dem Wert initialisiert, der sich ergibt, wenn das x-Feld und das y-Feld auf den Wert null gesetzt werden.

Der Standardwert einer Struktur entspricht dem Wert, der von dem Standardkonstruktor der Struktur zurückgegeben wird (§4.1.2). Eine Struktur darf im Gegensatz zu einer Klasse keinen parameterlosen Instanzkonstruktor deklarieren. Allerdings verfügen alle Strukturen implizit über einen parameterlosen Instanzkonstruktor, der immer den Wert zurückgibt, der erzeugt wird, wenn für alle Werttypfelder der Standardwert und für alle Verweistypfelder der Wert null festgelegt wird.

Strukturen sollten so entworfen werden, dass der Standardinitialisierungsstatus als gültiger Status betrachtet wird. In dem Beispiel

using System;

struct KeyValuePair  
{  
 string key;  
 string value;

public KeyValuePair(string key, string value) {  
 if (key == null || value == null) throw new ArgumentException();  
 this.key = key;  
 this.value = value;  
 }  
}

schützt der benutzerdefinierte Instanzkonstruktor nur gegen Nullwerte, wenn er explizit aufgerufen wird. Wenn die KeyValuePair-Variable der Standardwertinitialisierung unterliegt, sind das key-Feld und das value-Feld gleich null, sodass die Struktur für die Verarbeitung dieses Zustands eingerichtet werden muss.

### Boxing und Unboxing

Der Wert eines Klassentyps kann in den Typ object oder in einen Schnittstellentyp konvertiert werden, der von der Klasse implementiert wird, indem der Verweis einfach während der Kompilierung wie ein anderer Typ behandelt wird. Ebenso kann ein Wert des Typs object oder ein Wert eines Schnittstellentyps ohne Änderung des Verweises in einen Klassentyp zurückkonvertiert werden (in diesem Fall ist natürlich eine Laufzeittypüberprüfung erforderlich).

Da es sich bei Strukturen nicht um Verweistypen handelt, werden diese Operationen für Strukturtypen anders implementiert. Wenn der Wert eines Strukturtyps in den Typ object oder in einen von der Struktur implementierten Schnittstellentyp konvertiert wird, erfolgt eine Boxingoperation. Wenn ein Wert vom Typ object oder ein Wert eines Schnittstellentyps in den Strukturtyp zurückkonvertiert wird, erfolgt hingegen eine Unboxingoperation. Bei Klassentypen unterscheiden sich dieselben Operationen im Wesentlichen dadurch, dass beim Boxing und Unboxing der Strukturwert entweder in die geschachtelte Instanz oder aus der geschachtelten Instanz kopiert wird. Nach einer Boxing- oder einer Unboxingoperation werden daher die in der nicht geschachtelten Struktur vorgenommenen Änderungen nicht in der geschachtelten Struktur wiedergegeben.

Wenn ein Strukturtyp eine von System.Object vererbte virtuelle Methode (z. B. Equals, GetHashCode oder ToString) überschreibt, wird bei einem Aufruf der virtuellen Methode über eine Instanz des Strukturtyps kein Boxing ausgeführt. Dies ist auch dann der Fall, wenn die Struktur als Typparameter verwendet wird und der Aufruf über eine Instanz des Typparametertyps erfolgt. Beispiel:

using System;

struct Counter  
{  
 int value;

public override string ToString() {  
 value++;  
 return value.ToString();  
 }  
}

class Program  
{  
 static void Test<T>() where T: new() {  
 T x = new T();  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 Console.WriteLine(x.ToString());  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

Die Ausgabe des Programms sieht folgendermaßen aus:

1  
2  
3

Obwohl es kein guter Programmierungsstil ist, wenn ToString Nebeneffekte hat, wird im Beispiel veranschaulicht, dass bei den drei Aufrufen von x.ToString() kein Boxing erfolgt ist.

Entsprechend erfolgt beim Zugriff auf einen Member für einen eingeschränkten Typparameter kein implizites Boxing. Wenn beispielsweise die ICounter-Schnittstelle eine Increment-Methode enthält, die zum Ändern eines Werts verwendet werden kann, und ICounter als Einschränkung verwendet, wird die Implementierung der Increment-Methode mit einem Verweis auf die Variable aufgerufen, für die Increment aufgerufen wurde, und niemals auf eine durch Boxing generierte Kopie.

using System;

interface ICounter  
{  
 void Increment();  
}

struct Counter: ICounter  
{  
 int value;

public override string ToString() {  
 return value.ToString();  
 }

void ICounter.Increment() {  
 value++;  
 }  
}

class Program  
{  
 static void Test<T>() where T: ICounter, new() {  
 T x = new T();  
 Console.WriteLine(x);  
 x.Increment(); // Modify x  
 Console.WriteLine(x);  
 ((ICounter)x).Increment(); // Modify boxed copy of x  
 Console.WriteLine(x);  
 }

static void Main() {  
 Test<Counter>();  
 }  
}

Durch den ersten Aufruf von Increment wird der Wert in der Variablen x geändert. Dies ist nicht gleichbedeutend mit dem zweiten Aufruf von Increment, der den Wert in einer Boxingkopie von x ändert. Die Ausgabe des Programms lautet daher wie folgt:

0  
1  
1

Weitere Informationen zum Boxing und Unboxing finden Sie in §4.3.

### Bedeutung von "this"

Innerhalb eines Instanzkonstruktors oder eines Instanzfunktionsmembers einer Klasse wird this als Wert klassifiziert. Während mit this auf die Instanz verwiesen werden kann, für die der Funktionsmember aufgerufen wurde, kann in einem Funktionsmember einer Klasse keine Zuweisung zu this erfolgen.

Innerhalb eines Instanzkonstruktors einer Struktur entspricht this einem out-Parameter des Strukturtyps. Innerhalb eines Instanzfunktionsmembers einer Struktur entspricht this hingegen einem ref-Parameter des Strukturtyps. In beiden Fällen wird this als Variable klassifiziert. Die gesamte Struktur, für die der Funktionsmember aufgerufen wurde, kann geändert werden, indem eine Zuweisung zu this erfolgt oder this als ref-Parameter oder out-Parameter übergeben wird.

### Feldinitialisierungen

Wie in §11.3.4 beschrieben, besteht der Standardwert einer Struktur aus dem Wert, der erzeugt wird, wenn für alle Werttypfelder der Standardwert und für alle Verweistypfelder null festgelegt wird. Aus diesem Grund unterstützen Strukturen keine Variableninitialisierungen in Instanzfelddeklarationen. Diese Einschränkung trifft nur auf Instanzfelder zu. Statische Felder einer Struktur dürfen Variableninitialisierungen enthalten.

In dem Beispiel

struct Point  
{  
 public int x = 1; // Error, initializer not permitted  
 public int y = 1; // Error, initializer not permitted  
}

ist fehlerhaft, da in den Instanzfelddeklarationen Variableninitialisierungen enthalten sind.

### Konstruktoren

Eine Struktur darf im Gegensatz zu einer Klasse keinen parameterlosen Instanzkonstruktor deklarieren. Allerdings verfügen alle Strukturen implizit über einen parameterlosen Instanzkonstruktor, der immer den Wert zurückgibt, der erzeugt wird, wenn für alle Werttypfelder der Standardwert und für alle Verweistypfelder der Wert NULL festgelegt wird (§4.1.2). Eine Struktur kann Instanzkonstruktoren deklarieren, die über Parameter verfügen. Beispiel:

struct Point  
{  
 int x, y;

public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
}

Entsprechend der oben angegebenen Deklaration erstellen die beiden Anweisungen

Point p1 = new Point();

Point p2 = new Point(0, 0);

jeweils Point mit x und y, die mit 0 initialisiert werden.

Ein Strukturinstanzkonstruktor darf keinen Konstruktorinitialisierer der Form base(...) enthalten.

Die this-Variable entspricht einem out-Parameter des Strukturtyps, wenn der Strukturinstanzkonstruktor keinen Konstruktorinitialisierer festlegt. Ähnlich wie ein out-Parameter muss this an allen Positionen definitiv zugewiesen sein (§5.3), an die der Konstruktor zurückgegeben wird. Falls im Strukturinstanzkonstruktor ein Konstruktorinitialisierer angegeben wird, entspricht die this-Variable einem ref-Parameter des Strukturtyps. Ähnlich wie ein ref-Parameter wird this bei Eintritt in den Konstruktortext als definitiv zugewiesen betrachtet. Beachten Sie die folgende Implementierung eines Instanzkonstruktors:

struct Point  
{  
 int x, y;

public int X {  
 set { x = value; }  
 }

public int Y {  
 set { y = value; }  
 }

public Point(int x, int y) {  
 X = x; // error, this is not yet definitely assigned  
 Y = y; // error, this is not yet definitely assigned  
 }  
}

Bevor nicht alle Felder der zu erstellenden Struktur definitiv zugewiesen wurden, kann keine Instanzmemberfunktion (einschließlich set-Accessoren für die Eigenschaften X und Y) aufgerufen werden. Beachten Sie jedoch, dass die Implementierung des Instanzkonstruktors zulässig wäre, wenn Point eine Klasse und keine Struktur wäre.

### Destruktoren

Eine Struktur darf keinen Destruktor deklarieren.

### Statische Konstruktoren

Statische Konstruktoren für Strukturen unterliegen den meisten Regeln, die auch für Klassen gelten. Die Ausführung eines statischen Konstruktors für einen Strukturtyp wird durch das erste der folgenden Ereignisse ausgelöst, die in einer Anwendungsdomäne auftreten:

* Es wird auf einen statischen Member des Strukturtyps verwiesen.
* Es wird ein explizit deklarierter Konstruktor des Strukturtyps aufgerufen.

Das Erstellen von Standardwerten (§11.3.4) von Strukturtypen löst den statischen Konstruktor nicht auf. (Beispiel hierfür ist der Anfangswert der Elemente in einem Array.)

## Strukturbeispiele

Im Folgenden sind zwei signifikante Beispiele für die Verwendung von struct-Typen zur Erstellung von Typen dargestellt, die ähnlich wie die in der Sprache vordefinierten Typen verwendet werden können, allerdings mit anderer Semantik.

### Datenbankganzzahltyp

Die unten beschriebene DBInt-Struktur implementiert einen Integertyp, der die vollständige Gruppe von Werten des Typs int darstellen kann, sowie einen zusätzlichen Zustand, der einen unbekannten Wert angibt. Typen mit diesen Eigenschaften werden in der Regel in Datenbanken verwendet.

using System;

public struct DBInt  
{  
 // The Null member represents an unknown DBInt value.

public static readonly DBInt Null = new DBInt();

// When the defined field is true, this DBInt represents a known value  
 // which is stored in the value field. When the defined field is false,  
 // this DBInt represents an unknown value, and the value field is 0.

int value;  
 bool defined;

// Private instance constructor. Creates a DBInt with a known value.

DBInt(int value) {  
 this.value = value;  
 this.defined = true;  
 }

// The IsNull property is true if this DBInt represents an unknown value.

public bool IsNull { get { return !defined; } }

// The Value property is the known value of this DBInt, or 0 if this  
 // DBInt represents an unknown value.

public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {  
 return new DBInt(x);  
 }

// Explicit conversion from DBInt to int. Throws an exception if the  
 // given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {  
 if (!x.defined) throw new InvalidOperationException();  
 return x.value;  
 }

public static DBInt operator +(DBInt x) {  
 return x;  
 }

public static DBInt operator -(DBInt x) {  
 return x.defined ? -x.value : Null;  
 }

public static DBInt operator +(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value + y.value: Null;  
 }

public static DBInt operator -(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value - y.value: Null;  
 }

public static DBInt operator \*(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value \* y.value: Null;  
 }

public static DBInt operator /(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value / y.value: Null;  
 }

public static DBInt operator %(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value % y.value: Null;  
 }

public static DBBool operator ==(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value == y.value: DBBool.Null;  
 }

public static DBBool operator !=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value != y.value: DBBool.Null;  
 }

public static DBBool operator >(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value > y.value: DBBool.Null;  
 }

public static DBBool operator <(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value < y.value: DBBool.Null;  
 }

public static DBBool operator >=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value >= y.value: DBBool.Null;  
 }

public static DBBool operator <=(DBInt x, DBInt y) {  
 return x.defined && y.defined? x.value <= y.value: DBBool.Null;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBInt)) return false;  
 DBInt x = (DBInt)obj;  
 return value == x.value && defined == x.defined;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 return defined? value.ToString(): “DBInt.Null”;  
 }  
}

### Boolescher Datenbanktyp

Die unten beschriebene DBBool-Struktur implementiert einen logischen dreiwertigen Typ. Die möglichen Werte dieses Typs sind DBBool.True, DBBool.False und DBBool.Null, wobei der Null-Member einen unbekannten Wert angibt. Diese logischen dreiwertigen Typen werden in der Regel in Datenbanken verwendet.

using System;

public struct DBBool  
{  
 // The three possible DBBool values.

public static readonly DBBool Null = new DBBool(0);  
 public static readonly DBBool False = new DBBool(-1);  
 public static readonly DBBool True = new DBBool(1);

// Private field that stores –1, 0, 1 for False, Null, True.

sbyte value;

// Private instance constructor. The value parameter must be –1, 0, or 1.

DBBool(int value) {  
 this.value = (sbyte)value;  
 }

// Properties to examine the value of a DBBool. Return true if this  
 // DBBool has the given value, false otherwise.

public bool IsNull { get { return value == 0; } }

public bool IsFalse { get { return value < 0; } }

public bool IsTrue { get { return value > 0; } }

// Implicit conversion from bool to DBBool. Maps true to DBBool.True and  
 // false to DBBool.False.

public static implicit operator DBBool(bool x) {  
 return x? True: False;  
 }

// Explicit conversion from DBBool to bool. Throws an exception if the  
 // given DBBool is Null, otherwise returns true or false.

public static explicit operator bool(DBBool x) {  
 if (x.value == 0) throw new InvalidOperationException();  
 return x.value > 0;  
 }

// Equality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator ==(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value == y.value? True: False;  
 }

// Inequality operator. Returns Null if either operand is Null, otherwise  
 // returns True or False.

public static DBBool operator !=(DBBool x, DBBool y) {  
 if (x.value == 0 || y.value == 0) return Null;  
 return x.value != y.value? True: False;  
 }

// Logical negation operator. Returns True if the operand is False, Null  
 // if the operand is Null, or False if the operand is True.

public static DBBool operator !(DBBool x) {  
 return new DBBool(-x.value);  
 }

// Logical AND operator. Returns False if either operand is False,  
 // otherwise Null if either operand is Null, otherwise True.

public static DBBool operator &(DBBool x, DBBool y) {  
 return new DBBool(x.value < y.value? x.value: y.value);  
 }

// Logical OR operator. Returns True if either operand is True, otherwise  
 // Null if either operand is Null, otherwise False.

public static DBBool operator |(DBBool x, DBBool y) {  
 return new DBBool(x.value > y.value? x.value: y.value);  
 }

// Definitely true operator. Returns true if the operand is True, false  
 // otherwise.

public static bool operator true(DBBool x) {  
 return x.value > 0;  
 }

// Definitely false operator. Returns true if the operand is False, false  
 // otherwise.

public static bool operator false(DBBool x) {  
 return x.value < 0;  
 }

public override bool Equals(object obj) {  
 if (!(obj is DBBool)) return false;  
 return value == ((DBBool)obj).value;  
 }

public override int GetHashCode() {  
 return value;  
 }

public override string ToString() {  
 if (value > 0) return "DBBool.True";  
 if (value < 0) return "DBBool.False";  
 return "DBBool.Null";  
 }  
}

# Arrays

Ein Array ist eine Datenstruktur mit mehreren Variablen, auf die mithilfe von berechneten Indizes zugegriffen wird. Die in einem Array enthaltenen Variablen, die auch als Arrayelemente bezeichnet werden, sind alle vom selben Typ. Dieser Typ wird als Elementtyp des Arrays bezeichnet.

Ein Array hat einen Rang, der die Anzahl der Indizes angibt, die jedem Arrayelement zugeordnet sind. Der Rang eines Arrays wird auch als „Dimensionen des Arrays“ beschrieben. Ein Array mit dem Rang 1 wird als eindimensionales Array bezeichnet. Ein Array mit einem Rang größer als 1 wird als mehrdimensionales Array bezeichnet. Mehrdimensionale Arrays von einer bestimmten Größe werden häufig auch als zweidimensionale Arrays, dreidimensionale Arrays usw. bezeichnet.

Alle Dimensionen eines Arrays haben eine zugewiesene Länge, die durch eine ganze Zahl größer oder gleich 0 ausgedrückt wird. Die Dimensionslängen sind nicht Teil des Arraytyps, werden aber eingerichtet, wenn eine Instanz des Arraytyps zur Laufzeit erstellt wird. Die Länge einer Dimension bestimmt den gültigen Bereich der Indizes für diese Dimension: Bei einer Dimension mit der Länge N können die Indizes einen Wert von 0 bis einschließlich N – 1 annehmen. Die Gesamtzahl der Elemente in einem Array ist das Produkt aus den Längen der einzelnen Dimensionen des Arrays. Wenn eine oder mehrere Dimensionen eines Arrays die Länge 0 aufweisen, wird das Array als leer bezeichnet.

Der Elementtyp eines Arrays kann ein beliebiger Typ sein, z. B. auch einen Arraytyp.

## Arraytypen

Ein Arraytyp wird wie ein non-array-type geschrieben, gefolgt von einem oder mehreren rank-specifiers:

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

Ein non-array-type ist jeder Typ, der selbst kein array-type ist.

Der Rang eines Arraytyps wird durch den am weitesten links stehenden rank-specifier im array-type bestimmt: Ein rank-specifier zeigt an, dass es sich um ein Array mit dem Rang 1 plus der Anzahl der ","-Token im rank-specifier handelt.

Der Elementtyp eines Arraytyps ist der Typ, der entsteht, wenn der am weitesten links stehende rank-specifier entfernt wird:

* Ein Arraytyp der Form T[R] ist ein Array mit dem Rang R und einem Nicht-Array-Elementtyp T.
* Ein Arraytyp der Form T[R][R1]...[RN] ist ein Array mit dem Rang R und einem Elementtyp T[R1]...[RN].

Die rank-specifier werden im Endeffekt vor dem letzten Nicht-Array-Elementtyp von links nach rechts gelesen. Der Typ int[][,,][,] ist demnach als ein eindimensionales Array von dreidimensionalen Arrays zweidimensionaler Arrays deklariert, die den Typ int haben.

Zur Laufzeit kann der Wert eines Arraytyps null oder ein Verweis auf eine Instanz dieses Arraytyps sein.

### Der System.Array-Typ

Der Typ System.Array ist der abstrakte Basistyp aller Arraytypen. Es gibt eine implizite Verweiskonvertierung (§6.1.6) von jedem beliebigen Arraytypen in System.Array und eine explizite Verweiskonvertierung (§6.2.4) von System.Array in jeden beliebigen Arraytyp. Beachten Sie, dass System.Array selbst kein array-type ist. Vielmehr handelt es sich um einen class-type, von dem alle array-types abgeleitet werden.

Zur Laufzeit kann der Wert des Typs System.Array null sein oder ein Verweis auf eine Instanz eines beliebigen Arraytyps.

### Arrays und die generische IList-Schnittstelle

Ein eindimensionales Array T[] implementiert die Schnittstelle System.Collections.Generic.IList<T> (Kurzform IList<T>) und deren Basisschnittstellen. Entsprechend gibt es eine implizite Konvertierung von T[] in IList<T> und deren Basisschnittstellen. Wenn es darüber hinaus eine implizite Verweiskonvertierung von S in T gibt, dann implementiert S[] IList<T>, und es erfolgt eine implizite Verweiskonvertierung von S[] in IList<T> und deren Basisschnittstellen (§6.1.6). Bei einer expliziten Verweiskonvertierung von S in T erfolgt eine explizite Verweiskonvertierung von S[] in IList<T> und deren Basisschnittstellen (§6.2.4). Beispiel:

using System.Collections.Generic;

class Test  
{  
 static void Main() {  
 string[] sa = new string[5];  
 object[] oa1 = new object[5];  
 object[] oa2 = sa;

IList<string> lst1 = sa; // Ok  
 IList<string> lst2 = oa1; // Error, cast needed  
 IList<object> lst3 = sa; // Ok  
 IList<object> lst4 = oa1; // Ok

IList<string> lst5 = (IList<string>)oa1; // Exception  
 IList<string> lst6 = (IList<string>)oa2; // Ok  
 }  
}

Die Zuweisung lst2 = oa1 generiert einen Kompilierungsfehler, da die Konvertierung von object[] in IList<string> eine explizite und keine implizite Konvertierung ist. Das Umwandeln von (IList<string>)oa1 löst zur Laufzeit eine Ausnahme aus, da oa1 auf ein object[] und nicht eine string[] verweist. Das Umwandeln von (IList<string>)oa2 löst zur Laufzeit jedoch keine Ausnahme aus, da oa2 auf eine string[] verweist.

Bei einer impliziten oder expliziten Verweiskonvertierung von S[] in IList<T> erfolgt auch eine explizite Verweiskonvertierung von IList<T> und deren Basisschnittstellen in S[] (§6.2.4).

Wenn IList<T> vom Arraytyp S[] implementiert wird, lösen einige der Member der implementierten Schnittstelle möglicherweise Ausnahmen aus. Eine Beschreibung des genauen Verhaltens der Implementierung der Schnittstelle übersteigt den Rahmen dieser Spezifikation.

## Arrayerstellung

Arrayinstanzen werden mithilfe von array-creation-expressions (§7.6.10.4) oder Deklarationen von Feldern oder lokalen Variablen erstellt, die einen array-initializer (§12.6) enthalten.

Wenn eine Arrayinstanz erstellt wird, werden Rang und Länge jeder Dimension festgelegt. Diese bleiben dann für die gesamte Lebensdauer der Instanz konstant. Es ist also weder möglich, den Rang einer vorhandenen Arrayinstanz noch die Größe ihrer Dimensionen zu verändern.

Eine Arrayinstanz bezieht sich immer auf einen Arraytypen. Der Typ System.Array ist ein abstrakter Typ, der nicht mit Instanzen versehen werden kann.

Arrayelemente, die mithilfe von array-creation-expressions erstellt wurden, werden immer mit ihren Standardwerten (§5.2) initialisiert.

## Arrayelementzugriff

Auf Arrayelemente wird mithilfe von element-access-Ausdrücken (§7.6.6.1) der Form A[I1, I2, ..., IN] zugegriffen. Dabei ist A ein Ausdruck eines Arraytyps und IX ein Ausdruck des Typs int, uint, long, ulong oder kann implizit in einen oder mehrere dieser Typen konvertiert werden. Das Ergebnis eines Zugriffs auf eine Arrayelement ist eine Variable, nämlich das Arrayelement, das mithilfe der Indizes gewählt wurde.

Die Elemente eines Arrays können mithilfe einer foreach-Anweisung (§8.8.4) aufgelistet werden.

## Arraymember

Jeder Arraytyp erbt die vom Typ System.Array deklarierten Member.

## Arraykovarianz

Wenn für zwei reference-types A und B eine implizite Verweiskonvertierung (§6.1.6) oder eine explizite Verweiskonvertierung (§6.2.4) von A in B vorhanden ist, ist dieselbe Verweiskonvertierung von Arraytyp A[R] in Arraytyp B[R] vorhanden, wobei R ein beliebig angegebener (allerdings für beide Arraytypen identischer) rank-specifier ist. Diese Beziehung wird als Arraykovarianz bezeichnet. Arraykovarianz bedeutet im Einzelnen, dass ein Wert eines Arraytyps A[R] tatsächlich ein Verweis auf eine Instanz eines Arraytyps B[R] sein kann, sofern eine implizite Verweiskonvertierung von B in A vorhanden ist.

Aufgrund der Arraykovarianz umfassen Zuweisungen zu Elementen von Verweistypen Laufzeitprüfungen, die sicherstellen, dass der Wert, der dem Arrayelement zugewiesen wird, tatsächlich den zulässigen Typ aufweist (§7.17.1). Beispiel:

class Test  
{  
 static void Fill(object[] array, int index, int count, object value) {  
 for (int i = index; i < index + count; i++) array[i] = value;  
 }

static void Main() {  
 string[] strings = new string[100];  
 Fill(strings, 0, 100, "Undefined");  
 Fill(strings, 0, 10, null);  
 Fill(strings, 90, 10, 0);  
 }  
}

Die Zuweisung zu array[i] in der Fill-Methode enthält eine implizite Laufzeitprüfung, die sicherstellt, dass das Objekt, auf das value verweist, entweder null oder eine Instanz ist, die mit dem tatsächlichen Elementtyp von array kompatibel ist. In Main sind die ersten beiden Aufrufe von Fill erfolgreich, beim dritten Aufruf wird jedoch eine System.ArrayTypeMismatchException bei der Ausführung der ersten Zuweisung zu array[i] ausgelöst. Die Ausnahme tritt auf, weil ein geschachtelter int nicht in einem string-Array gespeichert werden kann.

Die Arraykovarianz wird insbesondere nicht auf value-type-Arrays ausgeweitet. So ist z. B. keine Konvertierung vorhanden, die es zulässt, ein int[] wie ein object[] zu behandeln.

## Arrayinitialisierungen

Arrayinitialisierungen können in Deklarationen von Feldern (§10.5), Deklarationen von lokalen Variablen (§8.5.1) und Ausdrücken für die Arrayerstellung (§7.6.10.4) festgelegt werden:

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

Ein Arrayinitialisierer besteht aus einer Reihe von Variableninitialisierern, die von den Token "{" und "}" eingeschlossen und mit dem Token "," getrennt werden. Jede Variableninitialisierung ist ein Ausdruck oder im Falle eines mehrdimensionalen Arrays eine geschachtelte Arrayinitialisierung.

Der Kontext, in dem eine Arrayinitialisierung verwendet wird, bestimmt den Typ des initialisierten Arrays. In einem Arrayerstellungsausdruck steht der Arraytyp direkt vor dem Initialisierer, oder es wird aus den Ausdrücken im Arrayinitialisierer auf diesen geschlossen. Bei einer Feld- oder Variablendeklaration entspricht der Arraytyp dem Typ des deklarierten Feldes bzw. der deklarierten Variablen. Wenn eine Arrayinitialisierung in einer Feld- oder Variablendeklaration verwendet wird, z. B.:

int[] a = {0, 2, 4, 6, 8};

dann ist das nur die Kurzform für einen äquivalenten Arrayerstellungsausdruck:

int[] a = new int[] {0, 2, 4, 6, 8};

Bei einem eindimensionalen Array muss die Arrayinitialisierung aus einer Reihe von Ausdrücken bestehen, die zuweisungskompatibel mit dem Elementtyp des Arrays sind. Die Ausdrücke initialisieren die Arrayelemente in aufsteigender Reihenfolge und beginnen beim Element mit dem Index 0. Die Anzahl der Ausdrücke in der Arrayinitialisierung bestimmt die Länge der erstellten Arrayinstanz. Der Arrayinitialisierer im vorherigen Beispiel erstellt z. B. eine int[]-Instanz mit der Länge 5 und initialisiert dann die Instanz mit folgenden Werten:

a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;

Bei einem mehrdimensionalen Array muss die Anzahl der geschachtelten Ebenen in der Arrayinitialisierung der Anzahl der Dimensionen des Arrays entsprechen. Die äußerste Schachtelungsebene entspricht der am weitesten links stehenden Dimension und die innerste Schachtelungsebene der am weitesten rechts stehenden Dimension. Die Länge jeder Dimension des Arrays wird durch die Anzahl der Elemente der entsprechenden Schachtelungsebene in der Arrayinitialisierung bestimmt. Bei jeder geschachtelten Arrayinitialisierung muss die Anzahl der Elemente mit der Elementanzahl der anderen Arrayinitialisierungen auf derselben Ebene übereinstimmen. Beispiel:

int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};

Hier wird ein zweidimensionales Array mit einer Länge von fünf für die am weitesten links stehende Dimension und mit einer Länge von zwei für die am weitesten rechts stehende Dimension erstellt:

int[,] b = new int[5, 2];

Die Arrayinstanz wird dann mit folgenden Werten initialisiert:

b[0, 0] = 0; b[0, 1] = 1;  
b[1, 0] = 2; b[1, 1] = 3;  
b[2, 0] = 4; b[2, 1] = 5;  
b[3, 0] = 6; b[3, 1] = 7;  
b[4, 0] = 8; b[4, 1] = 9;

Wenn eine andere Dimension als die äußerste rechte Dimension mit einer Länge gleich 0 (null) angegeben wird, wird davon ausgegangen, dass die darauf folgenden Dimensionen ebenfalls eine Länge gleich 0 (null) aufweisen. Beispiel:

int[,] c = {};

erstellt ein zweidimensionales Array mit einer Länge von 0 (null) für die jeweils am weitesten links und die am weitesten rechts stehende Dimension:

int[,] c = new int[0, 0];

Wenn ein Arrayerstellungsausdruck sowohl explizite Dimensionslängen als auch eine Arrayinitialisierung enthält, müssen die Längen konstante Ausdrücke sein und die Anzahl der Elemente auf jeder Schachtelungsebene mit der entsprechenden Dimensionslänge übereinstimmen. Hierzu einige Beispiele:

int i = 3;  
int[] x = new int[3] {0, 1, 2}; // OK  
int[] y = new int[i] {0, 1, 2}; // Error, i not a constant  
int[] z = new int[3] {0, 1, 2, 3}; // Error, length/initializer mismatch

Der Initialisierer für y verursacht einen Kompilierungsfehler, weil der Ausdruck für die Dimensionslänge keine Konstante ist, und der Initialisierer für z erzeugt einen Kompilierungsfehler, weil die Länge mit der Anzahl der Elemente im Initialisierer nicht übereinstimmt.

# Schnittstellen

Eine Schnittstelle definiert einen Vertrag. Eine Klasse oder Struktur, die eine Schnittstelle implementiert, muss ihren Vertrag einhalten. Eine Schnittstelle kann von mehreren Basisschnittstellen erben. Klassen oder Strukturen können jeweils mehrere Schnittstellen implementieren.

Schnittstellen können Methoden, Eigenschaften, Ereignisse und Indexer enthalten. Die Schnittstelle selbst stellt keine Implementierungen für die durch sie definierten Member bereit. Die Schnittstelle legt lediglich die Member fest, die von Klassen oder Strukturen, die die Schnittstelle implementieren, zur Verfügung gestellt werden müssen.

## Schnittstellendeklarationen

Eine interface-declaration ist eine type-declaration (§9.6), die einen neuen Schnittstellentyp deklariert.

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

Eine interface-declaration besteht aus einer optionalen Gruppe von attributes (§17), gefolgt von einer optionalen Gruppe von interface-modifiers (§13.1.1), gefolgt von einem optionalen partial-Modifizierer, gefolgt von einem interface-Schlüsselwort und einem identifier, der die Schnittstelle benennt, gefolgt von einer optionalen variant-*type-parameter-list*-Spezifikation (§13.1.3), gefolgt von einer optionalen interface-base-Spezifikation (§13.1.4), gefolgt von einer optionalen type-parameter-constraints-clauses-Spezifikation (§10.1.5), gefolgt von einem interface-body (§13.1.5), optional gefolgt von einem Semikolon.

### Schnittstellenmodifizierer

Eine interface-declaration kann optional eine Reihe von Schnittstellenmodifizierern enthalten:

interface-modifiers:  
interface-modifier  
interface-modifiers interface-modifier

interface-modifier:  
new  
public  
protected  
internal  
private

Es tritt ein Kompilierungsfehler auf, wenn derselbe Modifizierer mehrmals in einer Schnittstellendeklaration vorkommt.

Der new-Modifizierer ist nur bei Schnittstellen zulässig, die innerhalb einer Klasse definiert werden. Er gibt an, dass die Schnittstelle einen vererbten Member mit demselben Namen verdeckt, wie in §10.3.4 beschrieben.

Der public-, protected-, internal- und private-Modifizierer steuern den Zugriff auf die Schnittstelle. Abhängig vom Kontext, in dem die Schnittstellendeklaration steht, sind möglicherweise nur einige der Modifizierer zulässig (§3.5.1).

### Partial-Modifizierer

Der partial-Modifizierer gibt an, dass diese interface-declaration eine partielle Typdeklaration ist. Mehrere partielle Schnittstellendeklarationen mit demselben Namen in einem einschließenden Namespace oder einer einschließenden Typdeklaration bilden zusammen eine Schnittstellendeklaration. Dabei gelten die in §10.2 beschriebenen Regeln.

### Listen der varianten Typparameter

Listen varianter Typparameter können nur in Schnittstellen- und Delegattypen auftreten. Der Unterschied zu normalen type-parameter-lists besteht in der optionalen variance-annotation für jeden Typparameter.

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:  
in  
out

Ist die Varianzanmerkung out, gilt der Typparameter als kovariant. Ist die Varianzanmerkung in, gilt der Typparameter als kontravariant. Gibt es keine Varianzanmerkung, so gilt der Typparameter als invariant.

In dem Beispiel

interface C<out X, in Y, Z>   
{  
 X M(Y y);

Z P { get; set; }  
}

ist X kovariant, während Y kontravariant und Z invariant ist.

#### Varianzsicherheit

Das Auftreten von Varianzanmerkungen in der Liste der Typparameter eines Typs schränkt die Stellen ein, an denen Typen innerhalb der Typdeklaration vorkommen können.

Ein Typ T ist output-unsafe, wenn eine der folgenden Bedingungen zutrifft:

* T ist ein kontravarianter Typparameter.
* T ist ein Arraytyp mit einem ausgabeunsicheren Elementtyp.
* T ist ein Schnittstellen- oder Delegattyp S<A1,… AK>, der aus einem generischen Typ S<X1, .. XK> konstruiert wurde, wobei für mindestens ein Ai eine der folgenden Bedingungen zutrifft:
* Xi ist kovariant oder invariant, und Ai ist ausgabeunsicher.
* Xi ist kontravariant oder invariant, und Ai ist eingabesicher.

Ein Typ T ist input-unsafe, wenn eine der folgenden Bedingungen zutrifft:

* T ist ein kovarianter Typparameter.
* T ist ein Arraytyp mit einem eingabeunsicheren Elementtyp.
* T ist ein Schnittstellen- oder Delegattyp S<A1,… AK>, der aus einem generischen Typ S<X1, .. XK> konstruiert wurde, wobei für mindestens ein Ai eine der folgenden Bedingungen zutrifft:
* Xi ist kovariant oder invariant, und Ai ist eingabeunsicher.
* Xi ist kontravariant oder invariant, und Ai ist ausgabeunsicher.

Erwartungsgemäß ist ein ausgabeunsicherer Typ an einer Ausgabeposition verboten, während ein eingabeunsicherer Typ an einer Eingabeposition verboten ist.

Ein Typ ist output-safe, wenn er nicht ausgabeunsicher ist, und input-safe, wenn er nicht eingabeunsicher ist.

#### Varianzkonvertierung

Der Zweck von Varianzanmerkungen besteht darin, Schnittstellen- und Delegattypen flexiblere (aber weiterhin typsichere) Konvertierungen bereitzustellen. Daher nutzen die Definitionen impliziter (§6.1) und expliziter Konvertierungen (§6.2) das Konzept der Varianzkonvertierbarkeit, die folgendermaßen definiert wird:

Ein Typ T<A1, …, An> ist varianzkonvertierbar in einen Typ T<B1, …, Bn>, wenn T ein mit den varianten Typparametern T<X1, …, Xn> deklarierter Schnittstellen- oder Delegattyp ist und für jeden varianten Typparameter Xi eine der folgenden Bedingungen erfüllt ist:

* Xi ist kovariant, und es ist ein impliziter Verweis oder eine implizite Identitätskonvertierung von Ai in Bi vorhanden.
* Xi ist kontravariant, und es ist ein impliziter Verweis oder eine implizite Identitätskonvertierung von Bi in Ai vorhanden.
* Xi ist invariant, und es ist eine Identitätskonvertierung von Ai in Bi vorhanden.

### Basisschnittstellen

Eine Schnittstelle kann von keiner oder mehreren Schnittstellentypen erben. Diese werden als explizite Basisschnittstellen der Schnittstelle bezeichnet. Wenn eine Schnittstelle mindestens eine explizite Basisschnittstelle hat, folgt in der Deklaration dieser Schnittstelle auf den Schnittstellenbezeichner ein Doppelpunkt und eine durch Trennzeichen getrennte Liste der Basisschnittstellentypen.

interface-base:  
: interface-type-list

Bei einem konstruierten Schnittstellentyp werden die expliziten Basisschnittstellen durch Übernehmen der expliziten Deklarationen für die Basisschnittstellen aus der generischen Typdeklaration gebildet, indem anschließend jeder type-parameter in der Deklaration der Basisschnittstelle durch das entsprechende type-argument des konstruierten Typs ersetzt wird.

Auf die expliziten Basisschnittstellen einer Schnittstelle muss mindestens ebenso zugegriffen werden können, wie auf die Schnittstelle selbst (§3.5.4). Es tritt beispielsweise ein Kompilierungsfehler auf, wenn eine Schnittstelle vom Typ private oder internal in der interface-base einer public-Schnittstelle angegeben wird.

Außerdem wird ein Kompilierungsfehler verursacht, wenn eine Schnittstelle direkt oder indirekt von sich selbst erbt.

Die Basisschnittstellen einer Schnittstelle sind die expliziten Basisschnittstellen und deren Basisschnittstellen. Mit anderen Worten ist die Gruppe der Basisschnittstellen der vollständig transitive Schluss der expliziten Basisschnittstellen, ihrer expliziten Basisschnittstellen usw. Eine Schnittstelle erbt alle Member ihrer Basisschnittstellen. In dem Beispiel

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

interface IComboBox: ITextBox, IListBox {}

die Basisschnittstellen von IComboBox sind IControl, ITextBox und IListBox.

Das bedeutet, dass die Schnittstelle IComboBox den SetText-Member und den SetItems-Member genauso erbt wie den Paint-Member.

Jede Basisschnittstelle einer Schnittstelle muss ausgabesicher sein (§13.1.3.1). Eine Klasse oder Struktur, die eine Schnittstelle implementiert, implementiert demnach implizit alle dazugehörigen Basisschnittstellen.

### Schnittstellendefinition

Die interface-body einer Schnittstelle definiert die Member der Schnittstelle.

interface-body:  
{ interface-member-declarationsopt }

## Schnittstellenmember

Die Member einer Schnittstelle sind die von den Basisschnittstellen vererbten Member und die in der Schnittstelle selbst deklarierten Member.

interface-member-declarations:  
interface-member-declaration  
interface-member-declarations interface-member-declaration

interface-member-declaration:  
interface-method-declaration  
interface-property-declaration  
interface-event-declaration  
interface-indexer-declaration

Eine Schnittstellendeklaration kann 0 oder mehr Member deklarieren. Die Member einer Schnittstelle müssen Methoden, Eigenschaften, Ereignisse oder Indexer sein. Eine Schnittstelle kann weder Konstanten, Felder, Operatoren, Instanzkonstruktoren, Destruktoren oder Typen noch statische Member jeglicher Art enthalten.

Alle Schnittstellenmember haben implizit öffentlichen Zugriff. Es tritt ein Kompilierungsfehler auf, wenn Schnittstellen-Memberdeklarationen Modifizierer enthalten. Insbesondere können Schnittstellenmember nicht mit den Modifizierern abstract, public, protected, internal, private, virtual, override oder static deklariert werden.

In dem Beispiel

public delegate void StringListEvent(IStringList sender);

public interface IStringList  
{  
 void Add(string s);

int Count { get; }

event StringListEvent Changed;

string this[int index] { get; set; }  
}

deklariert eine Schnittstelle, die von allen möglichen Arten von Membern jeweils einen enthält: Eine Methode, eine Eigenschaft, ein Ereignis und einen Indexer.

Die interface-declaration erstellt einen neuen Deklarationsabschnitt (§3.3), und die interface-member-declarations, die unmittelbar in der interface-declaration enthalten sind, führen neue Member in diesen Deklarationsabschnitt ein. Für die interface-member-declarations gelten folgende Regeln:

* Der Name einer Methode muss sich von den Namen aller Eigenschaften und Ereignisse unterscheiden, die in derselben Schnittstelle deklariert sind. Zusätzlich muss sich die Signatur (§3.6) einer Methode von den Signaturen aller anderen Methoden unterscheiden, die in der gleichen Schnittstelle deklariert werden. Die Signaturen zweier Methoden in der gleichen Schnittstelle dürfen sich nicht nur in ref und out unterscheiden.
* Der Name einer Eigenschaft oder eines Ereignisses muss sich von den Namen aller anderen Member unterscheiden, die in derselben Schnittstelle deklariert sind.
* Die Signatur eines Indexers muss sich von den Signaturen aller anderen Indexer unterscheiden, die in derselben Schnittstelle deklariert sind.

Die vererbten Member einer Schnittstelle sind nicht ausdrücklich Teil des Deklarationsabschnitts der Schnittstelle. Folglich kann in einer Schnittstelle ein Member mit dem gleichen Namen oder der gleichen Signatur eines vererbten Members deklariert werden. Wenn dieser Fall eintritt, wird der Basisschnittstellenmember vom abgeleiteten Schnittstellenmember verdeckt. Das Verdecken eines geerbten Members ist kein Fehler, aber es führt zu einer Warnung seitens des Compilers. Um die Warnung zu unterdrücken, muss die Deklaration des abgeleiteten Schnittstellenmembers den new-Modifizierer enthalten, um anzuzeigen, dass der abgeleitete Member den Basismember verdecken soll. Dieses Thema wird in §3.7.1.2 ausführlicher behandelt.

Wenn der new-Modifizierer in einer Deklaration enthalten ist, die keinen vererbten Member verdeckt, wird eine diesbezügliche Warnung ausgegeben. Diese Warnung wird durch Entfernen des new-Modifizierers unterdrückt.

Beachten Sie, dass die Member in der Klasse object, genau genommen, keine Member einer Schnittstelle (§13.2) sind. Die Member in der object-Klasse stehen jedoch über die Membersuche in jedem beliebigen Schnittstellentyp (§7.4) zur Verfügung.

### Schnittstellenmethoden

Schnittstellenmethoden werden mithilfe von interface-method-declarations deklariert:

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

Die attributes, return-type, identifier und formal-parameter-list haben bei einer Schnittstellenmethodendeklaration dieselbe Bedeutung wie bei einer Methodendeklaration in einer Klasse (§10.6). Bei einer Schnittstellen-Methodendeklaration ist die Angabe eines Methodentextes nicht zulässig, daher endet sie immer mit einem Semikolon.

Jeder formale Parametertyp einer Schnittstellenmethode muss eingabesicher sein (§13.1.3.1), und der Rückgabetyp muss entweder void oder ausgabesicher sein. Außerdem muss jede Klassentypeinschränkung, jede Schnittstellentypeinschränkung und jede Typparametereinschränkung für jeden Typparameter der Methode eingabesicher sein.

Durch diese Regeln wird sichergestellt, dass jede kovariante oder kontravariante Verwendung der Schnittstelle typsicher bleibt. Beispiel:

interface I<out T> { void M<U>() where U : T; }

ist ungültig, da die Verwendung von T als Typparametereinschränkung für U nicht eingabesicher ist.

Ohne diese Einschränkung wäre es möglich, auf folgende Art und Weise gegen die Typsicherheit zu verstoßen:

class B {}  
class D : B {}  
class E : B {}  
class C : I<D> { public void M<U>() {…} }  
…  
I<B> b = new C();  
b.M<E>();

Hierbei handelt es sich um einen Aufruf von C.M<E>. . Für den Aufruf muss jedoch E von D abgeleitet werden, die Typsicherheit wäre also hier verletzt.

### Schnittstelleneigenschaften

Schnittstelleneigenschaften werden mithilfe von interface-property-declarations deklariert:

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

Die attributes, type und identifier haben bei einer Schnittstelleneigenschaftendeklaration dieselbe Bedeutung wie bei einer Eigenschaftendeklaration in einer Klasse (§10.7).

Die Accessoren einer Schnittstelleneigenschaftendeklaration entsprechen den Accessoren einer Klasseneigenschaftendeklaration (§10.7.2), allerdings darf der Accessorkörper nur aus einem Semikolon bestehen. Mithilfe eines Accessors wird angegeben, ob die Eigenschaft Lese- und Schreibzugriffe gleichzeitig bzw. jeweils nur Schreib- oder Lesezugriffe unterstützt.

Der Typ einer Schnittstelleneigenschaft muss ausgabesicher sein, wenn ein get-Accessor vorhanden ist. Wenn ein set-Accessor vorhanden ist, muss der Typ eingabesicher sein.

### Schnittstellenereignisse

Schnittstellenereignisse werden mithilfe von interface-event-declarations deklariert:

interface-event-declaration:  
attributesopt newopt event type identifier ;

Die attributes, type und identifier haben bei einer Schnittstellenereignisdeklaration dieselbe Bedeutung wie bei einer Ereignisdeklaration in einer Klasse (§10.8).

Der Typ eines Schnittstellenereignisses muss eingabesicher sein.

### Schnittstellenindexer

Schnittstellenindexer werden mithilfe von interface-indexer-declarations deklariert:

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

Die attributes, type und formal-parameter-list haben bei einer Schnittstellenindexerdeklaration dieselbe Bedeutung wie bei einer Indexerdeklaration in einer Klasse (§10.9).

Die Accessoren einer Schnittstellenindexerdeklaration entsprechen den Accessoren einer Klassenindexerdeklaration (§10.9), allerdings darf der Accessorkörper nur aus einem Semikolon bestehen. Mithilfe eines Accessors wird angegeben, ob der Indexer Lese- und Schreibzugriffe gleichzeitig bzw. jeweils nur Schreib- oder Lesezugriffe unterstützt.

Alle formalen Parametertypen eines Schnittstellenindexers müssen eingabesicher sein. Außerdem müssen alle formalen out- oder ref-Parametertypen ebenfalls ausgabesicher sein. Beachten Sie, dass sogar out-Parameter aufgrund einer Einschränkung der zugrunde liegenden Ausführungsplattform eingabesicher sein müssen.

Der Typ eines Schnittstellenindexers muss ausgabesicher sein, wenn ein get-Accessor vorhanden ist. Wenn ein set-Accessor vorhanden ist, muss der Typ eingabesicher sein.

### Schnittstellenmemberzugriff

Der Zugriff auf Schnittstellenmember erfolgt über Ausdrücke für den Memberzugriff (§7.6.4) und den Indexerzugriff (§7.6.6.2) im Format I.M und I[A]. Dabei ist I ein Schnittstellentyp, M eine Methode, Eigenschaft oder ein Ereignis dieses Schnittstellentyps und A eine Argumentliste des Indexers.

Bei Schnittstellen, die ausschließlich einfach vererbt sind (jede Schnittstelle in der Vererbungskette hat genau eine oder keine Basisschnittstelle), wirken sich die Regeln für Membersuche (§7.4), Methodenaufruf (§7.6.5.1) und Indexerzugriff (§7.6.6.2) genauso aus wie bei Klassen und Strukturen: Bei gleichem Namen oder gleicher Signatur verdecken weiter abgeleitete Member die weniger abgeleiteten Member. Bei mehrfach vererbten Schnittstellen können Mehrdeutigkeiten auftreten, wenn zwei oder mehrere nicht verknüpfte Basisschnittstellen Member mit demselben Namen oder derselben Signatur deklarieren. Dieser Abschnitt zeigt verschiedene Beispiele solcher Situationen. In allen Fällen können explizite Umwandlungen dazu verwendet werden, Mehrdeutigkeiten zu beseitigen.

In dem Beispiel

interface IList  
{  
 int Count { get; set; }  
}

interface ICounter  
{  
 void Count(int i);  
}

interface IListCounter: IList, ICounter {}

class C  
{  
 void Test(IListCounter x) {  
 x.Count(1); // Error  
 x.Count = 1; // Error  
 ((IList)x).Count = 1; // Ok, invokes IList.Count.set  
 ((ICounter)x).Count(1); // Ok, invokes ICounter.Count  
 }  
}

erzeugen die ersten beiden Anweisungen Fehler während der Kompilierung, da die Membersuche (§7.4) von Count in IListCounter mehrdeutig ist. Wie in diesem Beispiel dargestellt, wird die Mehrdeutigkeit durch Umwandlung von x und die Zuordnung zu den entsprechenden Basisschnittstellentypen aufgehoben. Solche Umwandlungen haben keine Auswirkung auf die Laufzeit. Sie stellen nur eine andere Sichtweise dar, bei der die Instanz bei der Kompilierung als untergeordneter Typ angesehen wird.

In dem Beispiel

interface IInteger  
{  
 void Add(int i);  
}

interface IDouble  
{  
 void Add(double d);  
}

interface INumber: IInteger, IDouble {}

class C  
{  
 void Test(INumber n) {  
 n.Add(1); // Invokes IInteger.Add  
 n.Add(1.0); // Only IDouble.Add is applicable  
 ((IInteger)n).Add(1); // Only IInteger.Add is a candidate  
 ((IDouble)n).Add(1); // Only IDouble.Add is a candidate  
 }  
}

wird IInteger.Add durch den Aufruf von n.Add(1) ausgewählt, indem die Überladungsauflösungsregeln entsprechend §7.5.3 angewendet werden. Entsprechend wird IDouble.Add durch den Aufruf von n.Add(1.0) ausgewählt. Wenn explizite Umwandlungen eingefügt werden, gibt es nur eine infrage kommende Methode und daher keine Mehrdeutigkeit.

In dem Beispiel

interface IBase  
{  
 void F(int i);  
}

interface ILeft: IBase  
{  
 new void F(int i);  
}

interface IRight: IBase  
{  
 void G();  
}

interface IDerived: ILeft, IRight {}

class A  
{  
 void Test(IDerived d) {  
 d.F(1); // Invokes ILeft.F  
 ((IBase)d).F(1); // Invokes IBase.F  
 ((ILeft)d).F(1); // Invokes ILeft.F  
 ((IRight)d).F(1); // Invokes IBase.F  
 }  
}

wird der IBase.F-Member vom ILeft.F-Member verdeckt. Der d.F(1)-Aufruf wählt daher ILeft.F aus, obwohl IBase.F im Zugriffspfad über IRight nicht verdeckt zu sein scheint.

Die Faustregel zur Ausblendung bei mehrfach vererbten Schnittstellen ist folgende: Wenn ein Member in einem Zugriffspfad verdeckt ist, so ist er in allen Zugriffspfaden ausgeblendet. Da IBase.F im Zugriffspfad von IDerived zu ILeft zu IBase verdeckt wird, ist dieser Member auch im Zugriffspfad von IDerived zu IRight zu IBase verdeckt.

## Vollqualifizierte Schnittstellenmembernamen

Auf einen Schnittstellenmember wird in einigen Fällen mit seinem vollqualifizierten Namen verwiesen. Der voll gekennzeichnete Name eines Schnittstellenmembers besteht aus dem Namen der Schnittstelle, in der der Member deklariert ist, einem darauf folgenden Punkt und dem Namen des Members. Der voll gekennzeichnete Name verweist auf die Schnittstelle, in der der Member deklariert ist. Bei den Deklarationen

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

ist der vollqualifizierte Name von Paint beispielsweise IControl.Paint, der vollqualifizierte Name von SetText ist ITextBox.SetText.

Beim obigen Beispiel ist es nicht möglich, auf Paint mit der Bezeichnung ITextBox.Paint zu verweisen.

Ist eine Schnittstelle Teil eines Namespaces, enthält der voll gekennzeichnete Name eines Schnittstellenmembers auch den Namespacenamen. Beispiel:

namespace System  
{  
 public interface ICloneable  
 {  
 object Clone();  
 }  
}

Der vollqualifizierte Name der Clone-Methode lautet hier System.ICloneable.Clone.

## Schnittstellenimplementierungen

Schnittstellen können von Klassen und Strukturen implementiert werden. Um anzuzeigen, dass eine Klasse oder Struktur eine Schnittstelle direkt implementiert, wird der Schnittstellenbezeichner in die Basisklassenliste der Klasse oder Struktur eingebunden. Beispiel:

interface ICloneable  
{  
 object Clone();  
}

interface IComparable  
{  
 int CompareTo(object other);  
}

class ListEntry: ICloneable, IComparable  
{  
 public object Clone() {...}

public int CompareTo(object other) {...}  
}

Eine Klasse oder Struktur, die eine Schnittstelle direkt implementiert, implementiert demnach ebenfalls implizit alle Basisschnittstellen der Schnittstelle direkt. Dies trifft auch zu, wenn in der Basisklassenliste der Klasse oder Struktur nicht alle Basisschnittstellen explizit aufgeführt sind. Beispiel:

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

class TextBox: ITextBox  
{  
 public void Paint() {...}

public void SetText(string text) {...}  
}

In diesem Beispiel implementiert die TextBox- Klasse sowohl IControl als auch ITextBox.

Wenn eine C-Klasse eine Schnittstelle direkt implementiert, implementieren alle von C abgeleiteten Klasse die Schnittstelle ebenfalls implizit. Die in einer Klassendeklaration angegebenen Basisschnittstellen können konstruierte Schnittstellentypen (§4.4) sein. Eine Basisschnittstelle kann nicht selbst Typparameter sein, sie kann aber die Typparameter im Bereich enthalten. Mit dem folgenden Code wird veranschaulicht, wie eine Klasse konstruierte Typen implementieren und erweitern kann:

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

Die Basisschnittstellen einer generischen Klassendeklaration müssen den in §13.4.2 beschriebenen Eindeutigkeitsregeln entsprechen.

### Explizite Implementierungen von Schnittstellenmembern

Zur Implementierung einer Schnittstelle kann in einer Klasse oder Struktur eine explizite Implementierung eines Schnittstellenmembers deklariert werden. Eine explizite Implementierung eines Schnittstellenmembers ist eine Deklaration einer Methode, einer Eigenschaft, eines Ereignisses oder eines Indexers, die auf einen voll gekennzeichneten Namen eines Schnittstellenmembers verweist. Beispiel:

interface IList<T>  
{  
 T[] GetElements();  
}

interface IDictionary<K,V>  
{  
 V this[K key];

void Add(K key, V value);  
}

class List<T>: IList<T>, IDictionary<int,T>  
{  
 T[] IList<T>.GetElements() {...}

T IDictionary<int,T>.this[int index] {...}

void IDictionary<int,T>.Add(int index, T value) {...}  
}

In diesem Beispiel sind IDictionary<int,T>.this und IDictionary<int,T>.Add explizite Implementierungen von Schnittstellenmembern.

Wenn in einigen Fällen der Name eines Schnittstellenmembers nicht angemessen ist, kann es sinnvoll sein, diesen Schnittstellenmember mithilfe einer expliziten Implementierung zu implementieren. Eine Klasse, die beispielsweise eine Dateibehandlung umfasst, würde demnach eine Close-Memberfunktion implementieren, die die Datenquelle freigibt, und die Dispose-Methode der IDisposable-Schnittstelle mithilfe der expliziten Implementierung eines Schnittstellenmembers implementieren.

interface IDisposable  
{  
 void Dispose();  
}

class MyFile: IDisposable  
{  
 void IDisposable.Dispose() {  
 Close();  
 }

public void Close() {  
 // Do what's necessary to close the file  
 System.GC.SuppressFinalize(this);  
 }  
}

Es ist nicht möglich, mithilfe eines Methodenaufrufs, eines Eigenschaftenzugriffs oder eines Indexerzugriffs über den voll gekennzeichneten Namen auf eine explizite Implementierung eines Schnittstellenmembers zuzugreifen. Auf eine explizite Implementierung eines Schnittstellenmembers kann nur über eine Schnittstelleninstanz zugegriffen werden, wobei in diesem Fall einfach auf den Membernamen verwiesen wird.

Bei der expliziten Implementierung eines Schnittstellenmembers wird ein Kompilierungsfehler angezeigt, wenn Zugriffsmodifizierer oder die Modifizierer abstract, virtual, override oder static mit eingebunden wurden.

Explizit implementierte Schnittstellenmember weisen andere Zugriffseigenschaften als andere Member auf. Da auf explizite Implementierungen von Schnittstellenmembern mithilfe eines Eigenschaftenzugriffs oder Methodenaufrufs niemals über den voll gekennzeichneten Namen zugegriffen werden kann, haben sie in gewisser Weise einen privaten Status. Da auf sie allerdings über eine Schnittstelleninstanz zugegriffen werden kann, haben sie auf der anderen Seite auch einen öffentlichen Status.

Explizite Implementierungen von Schnittstellenmembern haben in erster Linie zwei Funktionen:

* Da über Klassen- oder Strukturinstanzen nicht auf explizite Implementierungen von Schnittstellenmembern zugegriffen werden kann, ermöglichen sie den Ausschluss von Schnittstellenimplementierungen von der public-Schnittstelle einer Klasse oder Struktur. Dies ist vor allem dann nützlich, wenn eine Klasse oder Struktur eine interne Schnittstelle implementiert, die für die Benutzer der Klasse oder Struktur nicht von Interesse ist.
* Explizite Implementierungen von Schnittstellenmembern ermöglichen die Aufhebung von Mehrdeutigkeiten bei Schnittstellenmembern mit derselben Signatur. Ohne explizite Implementierungen von Schnittstellenmembern könnte eine Klasse oder Struktur keine verschiedenen Implementierungen von Schnittstellenmembern mit derselben Signatur und demselben Rückgabetyp aufweisen. Es gäbe auch generell nicht die Möglichkeit, Schnittstellenmember mit derselben Signatur und verschiedenen Rückgabetypen in einer Klasse oder Struktur zu implementieren.

Damit eine explizite Implementierung eines Schnittstellenmembers gültig ist, muss in der Basisklassenliste der Klasse oder Struktur eine Schnittstelle aufgeführt sein, die einen Member enthält, dessen voll gekennzeichneter Name, Typ und Parametertypen genau mit den Angaben der expliziten Implementierung des Schnittstellenmembers übereinstimmen. Bei der Klasse

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}

int IComparable.CompareTo(object other) {...} // invalid  
}

verursacht die Deklaration von IComparable.CompareTo einen Kompilierungsfehler, da IComparable nicht in der Basisklassenliste von Shape aufgeführt und keine Basisschnittstelle von ICloneable ist. Auch bei den folgenden Deklarationen

class Shape: ICloneable  
{  
 object ICloneable.Clone() {...}  
}

class Ellipse: Shape  
{  
 object ICloneable.Clone() {...} // invalid  
}

verursacht die Deklaration von ICloneable.Clone in Ellipse einen Kompilierungsfehler, da ICloneable nicht explizit in der Basisklassenliste von Ellipse aufgeführt ist.

Der voll gekennzeichnete Name muss auf die Schnittstelle verweisen, in der der Member deklariert ist. Daher muss in den Deklarationen

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

class TextBox: ITextBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}  
}

die explizite Implementierung des Paint-Schnittstellenmembers in der Form IControl.Paint angegeben werden.

### Eindeutigkeit von implementierten Schnittstellen

Die in einer generischen Typdeklaration implementierten Schnittstellen müssen für alle möglichen konstruierten Typen eindeutig sein. Ohne diese Regel wäre es unmöglich, die richtige aufzurufende Methode für bestimmte konstruierte Typen zu bestimmen. Angenommen, eine generische Klassendeklaration dürfte folgendermaßen geschrieben werden:

interface I<T>  
{  
 void F();  
}

class X<U,V>: I<U>, I<V> // Error: I<U> and I<V> conflict  
{  
 void I<U>.F() {...}  
 void I<V>.F() {...}  
}

Wenn dies erlaubt wäre, wäre es unmöglich zu bestimmen, welcher Code im folgenden Fall auszuführen ist:

I<int> x = new X<int,int>();  
x.F();

Um zu bestimmen, ob die Schnittstellenliste einer generischen Typdeklaration gültig ist, werden die folgenden Schritte ausgeführt:

* L sei die Liste der direkt in einer generischen Klassen-, Struktur- oder Schnittstellendeklaration C angegebenen Schnittstellen.
* L werden alle Basisschnittstellen der bereits in L enthaltenen Schnittstellen hinzugefügt.
* Alle Duplikate werden aus L entfernt.
* Wenn alle möglichen aus C konstruierten Typen nach dem Ersetzen der Typargumente in L zwei identische Schnittstellen in L zur Folge hätten, ist die Deklaration von C ungültig. Einschränkungsdeklarationen werden beim Bestimmen aller möglichen konstruierten Typen nicht berücksichtigt.

In der oben stehenden Klassendeklaration X enthält die Schnittstellenliste L I<U> und I<V>. Die Deklaration ist ungültig, da bei jedem konstruierten Typ mit U und V von demselben Typ diese beiden Schnittstellen auch von demselben Typ sein würden.

Bei auf unterschiedlichen Vererbungsebenen angegebenen Schnittstellen ist eine Vereinheitlichung möglich:

interface I<T>  
{  
 void F();  
}

class Base<U>: I<U>  
{  
 void I<U>.F() {…}  
}

class Derived<U,V>: Base<U>, I<V> // Ok  
{  
 void I<V>.F() {…}  
}

Dieser Code ist gültig, obwohl Derived<U,V> sowohl I<U> als auch I<V> implementiert. Der Code

I<int> x = new Derived<int,int>();  
x.F();

ruft die Methode in Derived auf, da Derived<int,int> tatsächlich I<int> (§13.4.6) erneut implementiert.

### Implementierungen von generischen Methoden

Wenn eine generische Methode implizit eine Schnittstellenmethode implementiert, müssen die für jeden Methodentypparameter angegebenen Einschränkungen in beiden Deklarationen (nachdem alle Schnittstellentypparameter durch die entsprechenden Typargumente ersetzt wurden) gleich sein, wobei Methodentypparameter anhand ihrer ordinalen Reihenfolge von links nach rechts bezeichnet werden.

Wenn jedoch eine generische Methode eine Schnittstellenmethode explizit implementiert, sind in der implementierenden Methode keine Einschränkungen zulässig. Stattdessen werden die Einschränkungen von der Schnittstellenmethode geerbt.

interface I<A,B,C>  
{  
 void F<T>(T t) where T: A;  
 void G<T>(T t) where T: B;  
 void H<T>(T t) where T: C;  
}

class C: I<object,C,string>  
{  
 public void F<T>(T t) {...} // Ok  
 public void G<T>(T t) where T: C {...} // Ok  
 public void H<T>(T t) where T: string {...} // Error  
}

Die C.F<T>-Methode implementiert implizit I<object,C,string>.F<T>. In diesem Fall ist es nicht erforderlich (und nicht zulässig), dass C.F<T> die Einschränkung T: object angibt, da object eine implizite Einschränkung für alle Typparameter darstellt. Die C.G<T>-Methode implementiert implizit I<object,C,string>.G<T>, weil die Einschränkungen mit denen in der Schnittstelle übereinstimmen, nachdem die Schnittstellentypparameter durch die entsprechenden Typargumente ersetzt wurden. Die Einschränkung für die Methode C.H<T> stellt einen Fehler dar, weil versiegelte Typen (in diesem Fall string) nicht als Einschränkungen verwendet werden können. Es ergäbe ebenfalls einen Fehler, die Einschränkung wegzulassen, da die Einschränkungen impliziter Schnittstellenmethodenimplementierungen übereinstimmen müssen. Folglich ist es nicht möglich, I<object,C,string>.H<T> implizit zu implementieren. Diese Schnittstellenmethode kann nur mit einer expliziten Schnittstellenmemberimplementierung implementiert werden.

class C: I<object,C,string>  
{  
 ...

public void H<U>(U u) where U: class {...}

void I<object,C,string>.H<T>(T t) {  
 string s = t; // Ok  
 H<T>(t);  
 }  
}

In diesem Beispiel ruft die explizite Schnittstellenmemberimplementierung eine öffentliche Methode auf, die über deutlich geringere Einschränkungen verfügt. Beachten Sie, dass die Zuweisung von t zu s gültig ist, da T eine Einschränkung von T: string erbt, obwohl diese Einschränkung in Quellcode nicht dargestellt werden kann.

### Schnittstellenzuordnung

Eine Klasse oder Struktur muss Implementierungen aller Member bereitstellen, die in der Basisklassenliste der Klasse oder Struktur aufgeführt sind. Der Vorgang der Lokalisierung von Implementierungen von Schnittstellenmembern in einer Klasse oder Struktur wird als Schnittstellenzuordnung bezeichnet.

Die Schnittstellenzuordnung für eine Klasse oder Struktur C lokalisiert eine Implementierung für jeden Member aller Schnittstellen, die in der Basisklassenliste von C angegeben wurden. Die Implementierung eines speziellen I.M-Schnittstellenmembers wird durch die Überprüfung aller Klassen oder Strukturen S bestimmt. Dabei ist I die Schnittstelle, in welcher der Member M deklariert ist. Die Suche beginnt bei C und wird für jede folgende Basisklasse von C wiederholt, bis eine Übereinstimmung gefunden wird:

* Wenn S eine Deklaration einer expliziten Implementierung eines Schnittstellenmembers enthält, der mit I und M übereinstimmt, dann ist dieser Member die Implementierung von I.M.
* Wenn S andererseits eine Deklaration eines nicht statischen public-Members enthält, der mit M übereinstimmt, dann ist dieser Member die Implementierung von I.M. Wenn mehrere Member übereinstimmen, ist nicht festgelegt, welcher Member die Implementierung von I.M ist. Dies kann nur dann auftreten, wenn S ein konstruierter Typ ist und die beiden im generischen Typ deklarierten Member über unterschiedliche Signaturen verfügen, die jedoch durch die Typargument identisch werden.

Wenn nicht alle Implementierungen für alle Member jeder Schnittstelle in der Basisklassenliste von C gefunden werden können, wird ein Kompilierungsfehler angezeigt. Beachten Sie, dass die Member einer Schnittstelle jene mit einschließen, die von den Basisschnittstellen vererbt werden.

Bei der Schnittstellenzuordnung stimmt ein Klassenmember A mit einem Schnittstellenmember B in folgenden Fällen überein:

* A und B sind Methoden, und Name, Typ sowie Liste der formalen Parameter von A und B sind identisch.
* A und B sind Eigenschaften, Name und Typ vonA und B sind identisch, und A weist dieselben Accessoren auf wie B (A kann zusätzliche Accessoren haben, wenn es sich nicht um eine explizite Implementierung eines Schnittstellenmembers handelt).
* A und B sind Ereignisse, und Name sowie Typ von A und B sind identisch.
* A und B sind Indexer, Typ und Liste der formalen Parameter von A und B sind identisch, und A weist dieselben Accessoren auf wie B (A kann zusätzliche Accessoren haben, wenn es sich nicht um eine explizite Implementierung eines Schnittstellenmembers handelt).

Die wichtigsten Aspekte des Algorithmus für die Schnittstellenzuordnung sind folgende:

* Explizite Implementierungen von Schnittstellenmembern haben bei der Bestimmung von Klassen- oder Strukturmembern, die einen Schnittstellenmember implementieren, Vorrang vor anderen Membern in derselben Klasse oder Struktur.
* Weder nicht öffentliche noch statische Member werden bei der Schnittstellenzuordnung beachtet.

In dem Beispiel

interface ICloneable  
{  
 object Clone();  
}

class C: ICloneable  
{  
 object ICloneable.Clone() {...}

public object Clone() {...}  
}

wird der ICloneable.Clone-Member von C zur Implementierung von Clone in ICloneable, da explizite Implementierungen von Schnittstellenmembern Vorrang vor anderen Membern haben.

Wenn eine Klasse bzw. Struktur zwei oder mehr Schnittstellen implementiert, die Member mit identischen Namen, Typen und Parametertypen enthalten, ist es möglich, jeden dieser Schnittstellenmember einer einzelnen Klasse oder Struktur zuzuordnen. Beispiel:

interface IControl  
{  
 void Paint();  
}

interface IForm  
{  
 void Paint();  
}

class Page: IControl, IForm  
{  
 public void Paint() {...}  
}

Hier werden die Paint-Methoden von IControl und IForm der Paint-Methode in Page zugeordnet. Natürlich ist es auch möglich, separate explizite Implementierungen von Schnittstellenmembern für die beiden Methoden zu erhalten.

Wenn eine Klasse oder Struktur eine Schnittstelle implementiert, die verdeckte Member enthält, müssen einige Member notwendigerweise unter Zuhilfenahme expliziter Implementierungen von Schnittstellenmembern implementiert werden. Beispiel:

interface IBase  
{  
 int P { get; }  
}

interface IDerived: IBase  
{  
 new int P();  
}

Eine Implementierung dieser Schnittstelle würde mindestens eine explizite Implementierung eines Schnittstellenmembers erfordern und dabei eine der folgenden Formen annehmen:

class C: IDerived  
{  
 int IBase.P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 public int P { get {...} }

int IDerived.P() {...}  
}

class C: IDerived  
{  
 int IBase.P { get {...} }

public int P() {...}  
}

Wenn eine Klasse mehrere Schnittstellen implementiert, die dieselbe Basisschnittstelle haben, kann nur eine Implementierung der Basisschnittstelle erfolgen. In dem Beispiel

interface IControl  
{  
 void Paint();  
}

interface ITextBox: IControl  
{  
 void SetText(string text);  
}

interface IListBox: IControl  
{  
 void SetItems(string[] items);  
}

class ComboBox: IControl, ITextBox, IListBox  
{  
 void IControl.Paint() {...}

void ITextBox.SetText(string text) {...}

void IListBox.SetItems(string[] items) {...}  
}

ist es nicht möglich, verschiedene Implementierungen für die IControl-Schnittstelle in der Basisklassenliste und für die von ITextBox und IListBox geerbten IControl-Schnittstellen zu erstellen. Natürlich gelten diese Schnittstellen nicht als separate Identitäten. Stattdessen verwenden die Implementierungen von ITextBox und IListBox dieselbe Implementierung von IControl, und ComboBox implementiert drei Schnittstellen: IControl, ITextBox und IListBox.

Die Member einer Basisklasse werden in die Schnittstellenzuordnung mit einbezogen. In dem Beispiel

interface Interface1  
{  
 void F();  
}

class Class1  
{  
 public void F() {}

public void G() {}  
}

class Class2: Class1, Interface1  
{  
 new public void G() {}  
}

wird die F-Methode in Class1 in der Implementierung von Interface1 in Class2 verwendet.

### Vererbung der Schnittstellenimplementierung

Eine Klasse erbt alle Schnittstellenimplementierungen aus ihrer Basisklasse.

Ohne explizite erneute Implementierung einer Schnittstelle können in einer abgeleiteten Klasse die von den Basisklassen geerbten Schnittstellenzuordnungen in keiner Weise verändert werden. Bei folgenden Deklarationen

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public void Paint() {...}  
}

class TextBox: Control  
{  
 new public void Paint() {...}  
}

verdeckt die Paint-Methode in TextBox die Paint-Methode in Control, die Zuordnung von Control.Paint zu IControl.Paint wird jedoch nicht verändert. Daher haben Aufrufe von Paint über Klassen- und Schnittstelleninstanzen folgende Auswirkungen:

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes Control.Paint();

Wenn aber eine Schnittstellenmethode einer virtuellen Methode in einer Klasse zugeordnet wird, können abgeleitete Klassen die virtuelle Methode überschreiben und die Implementierung der Schnittstelle verändern. Wenn das oben gezeigte Beispiel folgendermaßen umgeschrieben wird

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 public virtual void Paint() {...}  
}

class TextBox: Control  
{  
 public override void Paint() {...}  
}

dann treten folgende Effekte ein:

Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint(); // invokes Control.Paint();  
t.Paint(); // invokes TextBox.Paint();  
ic.Paint(); // invokes Control.Paint();  
it.Paint(); // invokes TextBox.Paint();

Da explizite Implementierungen von Schnittstellenmembern nicht virtuell deklariert werden können, ist es nicht möglich, eine solche Implementierung zu überschreiben. Es ist aber absolut zulässig, wenn eine explizite Implementierung eines Schnittstellenmembers eine andere Methode aufruft, die als virtuell deklariert und somit von abgeleiteten Klassen überschrieben werden kann. Beispiel:

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() { PaintControl(); }

protected virtual void PaintControl() {...}  
}

class TextBox: Control  
{  
 protected override void PaintControl() {...}  
}

Hier können die von Control abgeleiteten Klassen die Implementierung von IControl.Paint anpassen, indem die PaintControl-Methode überschrieben wird.

### Erneute Implementierung von Schnittstellen

Bei einer Klasse, die eine Schnittstellenimplementierung erbt, ist es zulässig, die Schnittstelle erneut zu implementieren, indem sie in die Basisklassenliste aufgenommen wird.

Eine erneute Implementierung einer Schnittstelle erfolgt nach denselben Zuordnungsregeln wie die erste Implementierung einer Schnittstelle. Die geerbte Schnittstellenzuordnung hat daher keinerlei Auswirkungen auf die für die erneute Implementierung der Schnittstelle erstellte Zuordnung. Bei folgenden Deklarationen

interface IControl  
{  
 void Paint();  
}

class Control: IControl  
{  
 void IControl.Paint() {...}  
}

class MyControl: Control, IControl  
{  
 public void Paint() {}  
}

wirkt sich die Tatsache, dass IControl.Paint durch Control zu Control.IControl.Paint zugeordnet wird, nicht auf die erneute Implementierung von MyControl aus, durch die IControl.Paint zu MyControl.Paint zugeordnet wird.

Vererbte public-Memberdeklarationen und vererbte explizite Implementierungen von Schnittstellenmembern werden in den Zuordnungsprozess für erneut implementierte Schnittstellen mit einbezogen. Beispiel:

interface IMethods  
{  
 void F();  
 void G();  
 void H();  
 void I();  
}

class Base: IMethods  
{  
 void IMethods.F() {}  
 void IMethods.G() {}  
 public void H() {}  
 public void I() {}  
}

class Derived: Base, IMethods  
{  
 public void F() {}  
 void IMethods.H() {}  
}

In diesem Beispiel ordnet die Implementierung von IMethods in Derived die Schnittstellenmethoden Derived.F, Base.IMethods.G, Derived.IMethods.H und Base.I zu.

Wenn eine Klasse eine Schnittstelle implementiert, implementiert sie auch implizit alle dazugehörigen Basisschnittstellen. Ebenso ist eine erneute Implementierung einer Schnittstelle implizit auch eine erneute Implementierung aller dazugehörigen Basisschnittstellen. Beispiel:

interface IBase  
{  
 void F();  
}

interface IDerived: IBase  
{  
 void G();  
}

class C: IDerived  
{  
 void IBase.F() {...}

void IDerived.G() {...}  
}

class D: C, IDerived  
{  
 public void F() {...}

public void G() {...}  
}

In diesem Fall umfasst die eine erneute Implementierung von IDerived auch die erneute Implementierung von IBase, wobei IBase.F zu D.F zugeordnet wird.

### Abstrakte Klassen und Schnittstellen

Wie eine nicht abstrakte Klasse muss eine abstrakte Klasse die Implementierungen aller Member der Schnittstellen bereitstellen, die in der Basisklassenliste der Klasse aufgeführt sind. Allerdings ist es bei abstrakten Klassen zulässig, Schnittstellenmethoden abstrakten Methoden zuzuordnen. Beispiel:

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 public abstract void F();  
 public abstract void G();  
}

In diesem Fall ordnet die Implementierung von IMethods F und G abstrakten Methoden zu, die in von C abgeleiteten, nicht abstrakten Klassen aufgehoben werden müssen.

Beachten Sie, dass explizite Implementierungen von Schnittstellenmembern nicht abstrakt sein können, der Aufruf von abstrakten Methoden jedoch absolut zulässig ist. Beispiel:

interface IMethods  
{  
 void F();  
 void G();  
}

abstract class C: IMethods  
{  
 void IMethods.F() { FF(); }

void IMethods.G() { GG(); }

protected abstract void FF();

protected abstract void GG();  
}

In diesem Beispiel müssen von C abgeleitete, nicht abstrakte Klassen FF und GG aufheben und die tatsächliche Implementierung von IMethods bereitstellen.

# Enumerationen

Ein Enumerationstyp ist ein spezieller Werttyp (§4.1), der eine Reihe von benannten Konstanten deklariert.

In dem Beispiel

enum Color  
{  
 Red,  
 Green,  
 Blue  
}

deklariert einen Enumerationstyp mit dem Namen Color und den Membern Red, Green und Blue.

## Enumerationsdeklarationen

Eine Enumerationsdeklaration deklariert einen neuen Enumerationstyp. Eine Enumerationsdeklaration beginnt mit dem Schlüsselwort enum und legt Namen, Zugriff, zugrunde liegenden Typ und Member der Enumeration fest.

enum-declaration:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

Jeder Enumerationstyp besitzt einen entsprechenden ganzzahligen Typ, der als zugrunde liegender Typ des Enumerationstyps bezeichnet wird. Dieser zugrunde liegende Typ muss alle Enumeratorwerte darstellen können, die in der Enumeration definiert wurden. Eine Enumerationsdeklaration kann den zugrunde liegenden Typ explizit als byte, sbyte, short, ushort, int, uint, long oder ulong deklarieren. Beachten Sie, dass char nicht als zugrunde liegender Typ verwendet werden kann. Bei einer Enumerationsdeklaration ohne explizite Angabe des zugrunde liegenden Typs ist der zugrunde liegende Typ int.

In dem Beispiel

enum Color: long  
{  
 Red,  
 Green,  
 Blue  
}

deklariert eine Enumeration mit dem zugrunde liegenden Typ long. Ein Entwickler könnte wie im Beispiel den zugrunde liegenden Typ long wählen, um die Verwendung aller Werte zu ermöglichen, die in den Bereich des Typs long, nicht aber in den Bereich des Typs int fallen, oder um diese Option für die Zukunft offen zu halten.

## Enumerationsmodifizierer

Eine enum-declaration kann optional eine Reihe von Enumerationsmodifizierern enthalten:

enum-modifiers:  
enum-modifier  
enum-modifiers enum-modifier

enum-modifier:  
new  
public  
protected  
internal  
private

Es tritt ein Kompilierungsfehler auf, wenn derselbe Modifizierer mehrmals in einer Enumerationsdeklaration vorkommt.

Die Modifizierer einer Enumerationsdeklaration haben dieselbe Bedeutung wie die einer Klassendeklaration (§10.1.1). Beachten Sie aber, dass die Modifizierer abstract und sealed in einer Enumerationsdeklaration nicht zulässig sind. Enumerationen können nicht abstrakt sein und lassen keine Ableitungen zu.

## Enumerationsmember

Im Text einer Enumerationstypdeklaration können mehrere Enumerationsmember definiert werden, die die benannten Konstanten des Enumerationstyps sind. Dabei dürfen Namen für Enumerationsmember nicht mehrmals vergeben werden.

enum-member-declarations:  
enum-member-declaration  
enum-member-declarations , enum-member-declaration

enum-member-declaration:  
attributesopt identifier  
attributesopt identifier = constant-expression

Jedem Enumerationsmember ist ein konstanter Wert zugeordnet. Der Typ dieses Wertes ist der zugrunde liegende Wert der entsprechenden Enumeration. Der konstante Wert jedes Enumerationsmembers muss im Zahlenbereich des zugrunde liegenden Typs der Enumeration liegen. In dem Beispiel

enum Color: uint  
{  
 Red = -1,  
 Green = -2,  
 Blue = -3  
}

ergibt einen Kompilierungsfehler, da die konstanten Werte -1, -2 und –3 nicht im Zahlenbereich des zugrunde liegenden ganzzahligen Typs uint liegen.

Mehrere Enumerationsmember können dieselbe Wertzuweisung erhalten. In dem Beispiel

enum Color   
{  
 Red,  
 Green,  
 Blue,

Max = Blue  
}

zeigt eine Enumeration mit zwei Enumerationsmembern (Blue und Max), denen derselbe Wert zugewiesen wird.

Der mit einem Enumerationsmember verknüpfte Wert kann entweder implizit oder explizit zugewiesen werden. Wenn bei der Deklaration eines Enumerationsmembers ein Initialisierer mit einer constant-expression vorhanden ist, wird der Wert dieses konstanten Ausdrucks implizit in den zugrunde liegenden Typ des Enumerationsmembers konvertiert und dem Enumerationsmember zugeordnet. Wenn in der Deklaration des Enumerationsmembers keine Initialisierung stattfindet, wird der verknüpfte Wert auf folgende Weise implizit festgelegt:

* Ist der Enumerationsmember der erste in der Deklaration, ist der zugeordnete Wert 0.
* Andernfalls wird der zugewiesene Wert erzeugt, indem der Wert des in der Deklaration vorangegangenen Enumerationsmembers um eins erhöht wird. Dieser erhöhte Wert muss innerhalb des Wertebereichs liegen, der durch den zugrunde liegenden Typ dargestellt werden kann. Andernfalls tritt ein Kompilierungsfehler auf.

In dem Beispiel

using System;

enum Color  
{  
 Red,  
 Green = 10,  
 Blue  
}

class Test  
{  
 static void Main() {  
 Console.WriteLine(StringFromColor(Color.Red));  
 Console.WriteLine(StringFromColor(Color.Green));  
 Console.WriteLine(StringFromColor(Color.Blue));  
 }

static string StringFromColor(Color c) {  
 switch (c) {  
 case Color.Red:   
 return String.Format("Red = {0}", (int) c);

case Color.Green:  
 return String.Format("Green = {0}", (int) c);

case Color.Blue:  
 return String.Format("Blue = {0}", (int) c);

default:  
 return "Invalid color";  
 }  
 }  
}

werden die Namen der Enumerationsmember und ihre zugewiesenen Werte ausgegeben. Die Ausgabe lautet:

Red = 0  
Green = 10  
Blue = 11

Das hat folgende Gründe:

* Dem Enumerationsmember Red wird automatisch der Wert 0 zugeordnet (da kein Initialisierer vorhanden ist und es sich um den ersten Enumerationsmember handelt).
* Dem Enumerationsmember Green wird explizit der Wert 10 zugewiesen.
* Dem Enumerationsmember Blue wird automatisch ein Wert zugewiesen, der um eins größer ist als der Wert des im Text vorangegangenen Members.

Der dem Enumerationsmember zugeordnete Wert darf weder direkt noch indirekt den Wert seines eigenen Enumerationsmembers verwenden. Bis auf diese Zirkularitätsbeschränkung kann bei Initialisierungen auf andere Initialisierungen von Membern verwiesen werden, unabhängig von ihrer Position im Text. Innerhalb der Initialisierung eines Enumerationsmembers wird bei den Werten anderer Enumerationsmember immer der zugrunde liegende Typ angenommen, sodass Umwandlungen bei dem Verweis auf andere Enumerationsmember nicht notwendig sind.

In dem Beispiel

enum Circular  
{  
 A = B,  
 B  
}

verursacht einen Kompilierungsfehler, da die Deklarationen von A und B zirkulär sind. A ist explizit von B abhängig, und B ist implizit von A abhängig.

Enumerationsmember werden analog zu Feldern innerhalb von Klassen benannt und bewertet. Der Gültigkeitsbereich eines Enumerationsmembers ist der Textkörper des enthaltenden Enumerationstyps. Innerhalb dieses Gültigkeitsbereichs kann auf die Enumerationsmember mit ihrem einfachen Namen verwiesen werden. Beim Zugriff durch anderen Code muss der Name eines Enumerationsmembers mit dem Namen des Enumerationstyps vervollständigt werden. Der Zugriff auf Enumerationsmember ist nicht deklariert. Auf einen Enumerationsmember kann zugegriffen werden, wenn der Zugriff auf den enthaltenden Enumerationstyp möglich ist.

## Der System.Enum-Typ

Der Typ System.Enum ist die abstrakte Basisklasse aller Enumerationstypen (unterscheidet sich von dem zugrunde liegenden Typ des Enumerationstyps). Die von System.Enum geerbten Member stehen in allen Enumerationstypen zur Verfügung. Eine Boxingkonvertierung (§4.3.1) von allen Enumerationstypen in System.Enum und eine Unboxingkonvertierung (§4.3.2) von System.Enum in alle Enumerationstypen sind vorhanden.

Beachten Sie, dass System.Enum selbst kein enum-type ist. Vielmehr handelt es sich um einen class-type, von dem alle enum-types abgeleitet werden. Der Typ System.Enum erbt vom Typ System.ValueType (§4.1.1), der selbst wiederum vom Typ object erbt. Zur Laufzeit kann der Wert des Typs System.Enum den Wert null annehmen oder ein Verweis auf einen geschachtelten Wert eines beliebigen Enumerationstyps sein.

## Enumerationswerte und -operationen

Jeder Enumerationstyp definiert einen eindeutigen Typ. Zur Umwandlung von Enumerationstypen in ganzzahlige Typen oder von einem Enumerationstyp in einen anderen ist eine explizite Enumerationskonvertierung (§6.2.2) erforderlich. Die Gruppe von Werten, die ein Enumerationstyp annehmen kann, ist nicht durch die Enumerationsmember begrenzt. Genau genommen kann jeder Wert des zugrunde liegenden Typs einer Enumeration in den Enumerationstyp umgewandelt werden und ist dann ein bestimmter gültiger Wert dieses Enumerationstyps.

Enumerationsmember haben denselben Typ wie der sie einschließende Enumerationstyp (mit Ausnahme von Membern innerhalb von Initialisierungen anderer Enumerationsmember: siehe §14.3). Der Wert eines im Enumerationstyp E deklarierten Enumerationsmembers mit dem zugewiesenen Wert v ist (E)v.

Die folgenden Operatoren können für Werte von Enumerationstypen verwendet werden: ==, !=, <, >, <=, >= (§7.10.5), binär + (§7.8.4), binär ‑ (§7.8.5), ^, &, | (§7.11.2), ~ (§7.7.4), ++ und -- (§7.6.9 und §7.7.5).

Jeder Enumerationstyp wird automatisch von der System.Enum -Klasse abgeleitet (die wiederum von System.ValueType und object abgeleitet wird). Folglich können von dieser Klasse vererbte Methoden und Eigenschaften für Werte eines Enumerationstyps verwendet werden.

# Delegaten

Delegaten ermöglichen Szenarien, für die in anderen Sprachen (z. B. C++, Pascal oder Modula) Funktionszeiger verwendet wurden. Delegaten sind im Gegensatz zu Funktionszeigern in C++ vollständig objektorientiert und schließen, anders als Memberfunktionszeiger in C++, sowohl eine Objektinstanz als auch eine Methode mit ein.

Eine Delegatdeklaration definiert eine Klasse, die von der System.Delegate-Klasse abgeleitet ist. Eine Delegatinstanz schließt eine Aufrufliste ein. Dies ist eine Liste aus einer oder mehreren Methoden, auf die als aufrufbare Entität verwiesen werden kann. Bei Instanzmethoden besteht eine aufrufbare Entität beispielsweise aus einer Instanz und einer Methode für diese Instanz. Bei statischen Methoden besteht eine aufrufbare Entität nur aus einer Methode. Beim Aufruf einer Delegatinstanz mit einem entsprechenden Satz von Argumenten werden alle aufrufbaren Entitäten des Delegaten mit dem vorgegebenen Satz von Argumenten aufgerufen.

Eine interessante und nützliche Eigenschaft einer Delegatinstanz ist die Tatsache, dass sie die Klassen der enthaltenen Methoden nicht kennt oder sich nicht dafür interessiert. Die einzige Voraussetzung besteht darin, dass die Methoden mit dem Typ des Delegaten kompatibel sind (§15.1). Dadurch sind Delegaten hervorragend für „anonyme“ Aufrufe geeignet.

## Delegatdeklarationen

Eine delegate-declaration ist eine type-declaration (§9.6), die einen neuen Delegattyp deklariert.

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type   
 identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:  
delegate-modifier  
delegate-modifiers delegate-modifier

delegate-modifier:  
new  
public  
protected  
internal  
private

Es tritt ein Kompilierungsfehler auf, wenn derselbe Modifizierer mehrmals in einer Delegatdeklaration vorkommt.

Der new-Modifizierer ist nur für Delegaten zulässig, die mit einem anderen Typ deklariert wurden. In diesem Fall gibt der Modifizierer an, dass ein solcher Delegat einen geerbten Member desselben Namens verdeckt, wie in §10.3.4 beschrieben.

Der public-, protected-, internal- und private-Modifizierer steuern den Zugriff auf den Delegattyp. Abhängig vom Kontext, in dem die Delegatdeklaration steht, sind möglicherweise nicht alle Modifizierer zulässig (§3.5.1).

Der Name des Delegattyps wird durch identifier angegeben.

Die optionale formal-parameter-list legt die Parameter des Delegaten fest, und return-type gibt den Rückgabetyp an.

Die optionale variant-type-parameter-list (§13.1.3) gibt die Typparameter für den Delegaten selbst an.

Der Rückgabetyp eines Delegattyps muss entweder void oder ausgabesicher sein (§13.1.3.1).

Alle formalen Parametertypen eines Delegattyps müssen eingabesicher sein. Außerdem müssen alle out- oder ref-Parametertypen ebenfalls ausgabesicher sein. Beachten Sie, dass sogar out-Parameter aufgrund einer Einschränkung der zugrunde liegenden Ausführungsplattform eingabesicher sein müssen.

Delegattypen in C# sind namentlich, nicht jedoch strukturell äquivalent. Zwei verschiedene Delegattypen, die dieselben Parameterlisten und Rückgabetypen aufweisen, werden als unterschiedliche Delegattypen angesehen. Beim Vergleich der Instanzen zweier unterschiedlicher, strukturell jedoch identischer Delegattypen könnte möglicherweise ausgegeben werden, dass diese Instanzen gleich sind (§7.9.8).

Beispiel:

delegate int D1(int i, double d);

class A  
{  
 public static int M1(int a, double b) {...}  
}

class B  
{  
 delegate int D2(int c, double d);

public static int M1(int f, double g) {...}

public static void M2(int k, double l) {...}

public static int M3(int g) {...}

public static void M4(int g) {...}  
}

Die Methoden A.M1 und B.M1 sind mit den beiden Delegattypen D1 und D2 kompatibel, da sie dieselben Rückgabetypen und Parameterlisten aufweisen. Dennoch sind diese Delegattypen zwei verschiedene Typen und daher nicht austauschbar. Die Methoden B.M2, B.M3 und B.M4 sind mit den Delegattypen D1 und D2 inkompatibel, da sie über verschiedene Rückgabetypen oder Parameterlisten verfügen.

Wie bei anderen generischen Typdeklarationen müssen Typargumente angegeben werden, um einen konstruierten Delegattyp zu erstellen. Die Parametertypen und der Rückgabetyp eines konstruierten Delegattyps werden durch Ersetzung jedes Typparameters in der Delegatdeklaration durch das entsprechende Typargument des konstruierten Delegattyps erstellt. Der resultierende Rückgabetyp und die Typparameter werden beim Bestimmen der mit einem konstruierten Delegattyp kompatiblen Methoden verwendet. Beispiel:

delegate bool Predicate<T>(T value);

class X  
{  
 static bool F(int i) {...}

static bool G(string s) {...}  
}

Die Methode X.F ist mit dem Delegattypen Predicate<int> und die Methode X.G mit dem Delegattypen Predicate<string> kompatibel.

Ein Delegattyp kann nur mithilfe einer delegate-declaration deklariert werden. Ein Delegattyp ist ein Klassentyp, der von der System.Delegate-Klasse abgeleitet ist. Delegattypen sind implizit versiegelt (sealed), daher ist es nicht zulässig, von einem Delegattyp einen anderen Typ abzuleiten. Ebenfalls nicht zulässig ist die Ableitung eines Klassentyps von System.Delegate, der kein Delegattyp ist. Beachten Sie, dass System.Delegate selbst kein Delegattyp ist. Es handelt sich hierbei um einen Klassentyp, von dem alle Delegattypen abgeleitet werden.

C# stellt eine spezielle Syntax für die Instanziierung und den Aufruf von Delegattypen bereit. Abgesehen von der Instanziierung kann jede Operation, die auf eine Klasse oder Klasseninstanz angewendet werden kann, auch entsprechend auf eine Delegatklasse bzw. -instanz angewendet werden. Es ist daher möglich, auf Member des Typs System.Delegate mithilfe der üblichen Syntax für Memberzugriffe zuzugreifen.

Die von einer Delegatinstanz eingeschlossene Gruppe von Methoden wird Aufrufliste genannt. Bei der Erstellung einer Delegatinstanz (§15.2) aus einer einzelnen Methode schließt diese Instanz die Methode ein, und die Aufrufliste enthält nur einen einzelnen Eintrag. Beim Kombinieren zweier Delegatinstanzen, die ungleich null sind, werden ihre Aufruflisten verkettet (in der Reihenfolge linker Operand, rechter Operand), um eine neue Aufrufliste zu erstellen, die zwei oder mehr Einträge aufweist.

Delegaten werden mit den binären Operatoren + (§7.8.4) und +== (§7.17.2) kombiniert. Mithilfe der binären Operatoren - (§7.8.5) und -= (§7.17.2) kann ein Delegat aus einer Kombination von Delegaten entfernt werden. Delegaten können auch auf Gleichheit überprüft werden (§7.10.8).

Das folgende Beispiel zeigt die Instanziierung mehrerer Delegaten sowie ihre entsprechenden Aufruflisten:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}

public static void M2(int i) {...}

}

class Test  
{  
 static void Main() {  
 D cd1 = new D(C.M1); // M1  
 D cd2 = new D(C.M2); // M2  
 D cd3 = cd1 + cd2; // M1 + M2  
 D cd4 = cd3 + cd1; // M1 + M2 + M1  
 D cd5 = cd4 + cd3; // M1 + M2 + M1 + M1 + M2  
 }

}

Bei der Instanziierung von cd1 und cd2 schließen diese jeweils eine Methode ein. Nach der Instanziierung von cd3 hat dieser Delegat eine Aufrufliste mit zwei Methoden in der Reihenfolge M1, M2. Die Aufrufliste von cd4 enthält die Methoden in der Reihenfolge M1, M2, M1. Die Aufrufliste von cd5 enthält die Methoden in der Reihenfolge M1, M2, M1, M1, M2. Weitere Beispiele zum Kombinieren und Entfernen von Delegaten finden Sie in §15.4.

## Kompatibilität von Delegaten

Eine Methode oder ein Delegat M ist mit einem Delegattyp D kompatibell, wenn die folgenden Bedingungen erfüllt sind:

* D und M verfügen über dieselbe Anzahl von Parametern, und jeder Parameter in D besitzt dieselben ref-Modifizierer oder out-Modifizierer wie der entsprechende Parameter in M.
* Für jeden Wertparameter (jeden Parameter ohne ref- bzw out out-Modifizierer) erfolgt eine Identitätskonvertierung (§6.1.1) oder implizite Verweiskonvertierung (§6.1.6) vom Parametertyp in D in den entsprechenden Parametertyp in M.
* Für jeden ref- bzw. out-Parameter ist der Parametertyp in D mit dem Parametertyp in M identisch.
* Eine Identitätskonvertierung bzw. implizite Verweiskonvertierung erfolgt vom Rückgabetyp von M in den Rückgabetyp von D.

## Delegatinstanziierung

Eine Instanz eines Delegaten wird mithilfe einer delegate-creation-expression (§7.6.10.5) oder einer Konvertierung in einen Delegattyp erstellt. Die neu erstellte Delegatinstanz verweist dann auf eines der folgenden Ziele:

* Die statische Methode, auf die in der delegate-creation-expression verwiesen wurde
* Das Zielobjekt (das nicht null sein kann) und die Instanzmethode, auf die in der delegate-creation-expression verwiesen wurde
* Einen anderen Delegaten

Beispiel:

delegate void D(int x);

class C  
{  
 public static void M1(int i) {...}  
 public void M2(int i) {...}  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1); // static method  
 C t = new C();  
 D cd2 = new D(t.M2); // instance method  
 D cd3 = new D(cd2); // another delegate  
 }  
}

Nach der Instanziierung verweisen die Delegatinstanzen immer auf dasselbe Zielobjekt und dieselbe Methode. Wenn zwei Delegaten kombiniert oder ein Delegat aus einem anderen entfernt wird, entsteht ein neuer Delegat mit einer eigenen Aufrufliste. Die Aufruflisten der kombinierten oder entfernten Delegaten bleiben unverändert.

## Delegataufruf

C# stellt eine spezielle Syntax für den Aufruf von Delegaten bereit. Beim Aufruf einer Delegatinstanz, die ungleich null ist und deren Aufrufliste nur einen Eintrag enthält, ruft die Instanz diese Methode mit den übergebenen Argumenten auf und gibt denselben Wert zurück wie die Methode, auf die verwiesen wurde. (Detaillierte Informationen zum Aufrufen von Delegaten finden Sie in §7.6.5.3.) Wenn während des Aufrufs eines solchen Delegaten eine Ausnahme auftritt, die nicht innerhalb der aufgerufenen Methode abgefangen werden kann, wird die Suche nach einer catch-Ausnahmeklausel in der aufrufenden Methode fortgesetzt, so als wäre die Methode, auf die der Delegat verweist, direkt aufgerufen worden.

Bei einem Aufruf einer Delegatinstanz, deren Aufrufliste mehrere Einträge enthält, werden alle Methoden der Aufrufliste synchron der Reihe nach aufgerufen. An jede Methode werden dieselben Argumente weitergereicht, die an die Delegatinstanz übergeben wurden. Wenn ein Delegataufruf dieser Art Verweisparameter enthält (§10.6.1.2), wird jeder Methodenaufruf mit einem Verweis auf dieselbe Variable durchgeführt. Verändert eine Methode in der Aufrufliste diese Variable, ist diese Änderung für die folgenden Methoden in der Aufrufliste sichtbar. Wenn der Delegataufruf Ausgabeparameter oder einen Rückgabewert enthält, wird der endgültige Wert vom Aufruf des letzten Delegaten in der Liste bestimmt.

Wenn während des Aufrufs eines solchen Delegaten eine Ausnahme auftritt, die nicht innerhalb der aufgerufenen Methode abgefangen werden kann, wird die Suche nach einer catch-Ausnahmeklausel in der aufrufenden Methode fortgesetzt, und die in der Aufrufliste folgenden Methoden werden nicht aufgerufen.

Der Versuch, eine Delegatinstanz mit dem Wert null aufzurufen, resultiert in einer Ausnahme des Typs System.NullReferenceException.

Das folgende Beispiel zeigt, wie Sie Delegaten instanziieren, kombinieren, entfernen und aufrufen:

using System;

delegate void D(int x);

class C  
{  
 public static void M1(int i) {  
 Console.WriteLine("C.M1: " + i);  
 }

public static void M2(int i) {  
 Console.WriteLine("C.M2: " + i);  
 }

public void M3(int i) {  
 Console.WriteLine("C.M3: " + i);  
 }  
}

class Test  
{  
 static void Main() {   
 D cd1 = new D(C.M1);  
 cd1(-1); // call M1

D cd2 = new D(C.M2);  
 cd2(-2); // call M2

D cd3 = cd1 + cd2;  
 cd3(10); // call M1 then M2

cd3 += cd1;  
 cd3(20); // call M1, M2, then M1

C c = new C();  
 D cd4 = new D(c.M3);  
 cd3 += cd4;  
 cd3(30); // call M1, M2, M1, then M3

cd3 -= cd1; // remove last M1  
 cd3(40); // call M1, M2, then M3

cd3 -= cd4;  
 cd3(50); // call M1 then M2

cd3 -= cd2;  
 cd3(60); // call M1

cd3 -= cd2; // impossible removal is benign  
 cd3(60); // call M1

cd3 -= cd1; // invocation list is empty so cd3 is null

// cd3(70); // System.NullReferenceException thrown

cd3 -= cd1; // impossible removal is benign  
 }  
}

Die Anweisung cd3 += cd1; zeigt, dass ein Delegat mehrmals in einer Aufrufliste vertreten sein kann. In diesem Fall wird er einfach einmal pro Eintrag aufgerufen. In einer Aufrufliste wie dieser wird im Falle des Entfernens der letzte Eintrag des betreffenden Delegaten entfernt.

Direkt vor der Ausführung der letzten Anweisung, cd3 -= cd1;, verweist der Delegat cd3 auf eine leere Aufrufliste. Der Versuch, einen Delegaten aus einer leeren Liste (oder einen nicht vorhandenen Delegaten aus einer nicht leeren Liste) zu entfernen, ist kein Fehler.

Die Ausgabe lautet:

C.M1: -1  
C.M2: -2  
C.M1: 10  
C.M2: 10  
C.M1: 20  
C.M2: 20  
C.M1: 20  
C.M1: 30  
C.M2: 30  
C.M1: 30  
C.M3: 30  
C.M1: 40  
C.M2: 40  
C.M3: 40  
C.M1: 50  
C.M2: 50  
C.M1: 60  
C.M1: 60

# Ausnahmen

Ausnahmen stellen in C# eine strukturierte, einheitliche und typensichere Art der Behandlung von Fehlerzuständen auf System- und Anwendungsebene bereit. Der Ausnahmemechanismus in C# ist dem von C++ sehr ähnlich, weist aber einige wichtige Unterschiede auf:

* In C# müssen alle Ausnahmen durch eine Instanz eines Klassentyps dargestellt werden, die von System.Exception abgeleitet ist. In C++ kann jeder Wert eines beliebigen Typs zur Darstellung einer Ausnahme verwendet werden.
* In C# kann ein finally-Block (§8.10) verwendet werden, um den Code zum Beenden zu schreiben, der bei normaler Ausführung und in Ausnahmezuständen ausgeführt wird. Es ist schwierig, diese Art von Code in C++ zu schreiben, ohne Code zu duplizieren.
* In C# haben Ausnahmen auf der Systemebene wie Überlauf, Division durch 0 oder Null-Dereferenzierung klar definierte Ausnahmeklassen und stehen auf einer Stufe mit Fehlerzuständen auf der Anwendungsebene.

## Ursachen von Ausnahmen

Ausnahmen können auf zwei verschiedene Arten ausgelöst werden.

* Eine throw-Anweisung (§8.9.5) löst eine Ausnahme sofort und ohne Bedingung aus. Die Steuerung erreicht niemals die unmittelbar auf throw folgende Anweisung.
* Bestimmte Ausnahmezustände, die bei der Verarbeitung von C#-Anweisungen und -Ausdrücken auftreten, verursachen unter bestimmten Umständen eine Ausnahme, wenn die Operation nicht mehr normal durchgeführt werden kann. Eine Operation zur Ganzzahldivision (§7.8.2) löst beispielsweise die Ausnahme System.DivideByZeroException aus, wenn der Nenner gleich 0 ist. Eine Liste der verschiedenen Ausnahmen, die auf diese Weise ausgelöst werden können, finden Sie in §16.4.

## Die System.Exception-Klasse

Die System.Exception-Klasse ist der Basistyp für alle Ausnahmen. Diese Klasse hat einige Eigenschaften, die alle Ausnahmen gemeinsam haben:

* Message ist eine schreibgeschützte Eigenschaft des Typs string, die eine Klartextbeschreibung der Gründe für die Ausnahme enthält.
* InnerException ist eine schreibgeschützte Eigenschaft des Typs Exception. Wenn der Wert ungleich NULL ist, verweist er auf die Ausnahme, die die aktuelle Ausnahme ausgelöst hat, d. h., dass die aktuelle Ausnahme in einem den Ausnahmetyp InnerException verarbeitenden catch-Block ausgelöst wurde. Andernfalls ist ihr Wert null und zeigt damit an, dass diese Ausnahme nicht durch eine andere Ausnahme ausgelöst wurde. Die Anzahl der auf diese Weise miteinander verketteten Ausnahmeobjekte kann beliebig groß sein.

Die Werte dieser Eigenschaften können in Aufrufen des Instanzkonstruktors für System.Exception festgelegt werden

## Behandeln von Ausnahmen

Ausnahmen werden mit einer try-Anweisung (§8.10) behandelt.

Wenn eine Ausnahme ausgelöst wird, sucht das System nach der nächstgelegenen catch-Klausel zur Behandlung der Ausnahme, wie es im Laufzeittyp der Ausnahme festgelegt ist. Zunächst werden die aktuelle Methode nach einer lexikalisch einschließenden try-Anweisung durchsucht und die zugeordneten catch-Klauseln nacheinander abgearbeitet. Ist dies nicht erfolgreich, wird die Methode, die die aktuelle Methode aufgerufen hat, nach einer lexikalisch einschließenden try-Anweisung durchsucht, die auch den Aufruf der aktuellen Methode einschließt. Diese Suche wird solange fortgesetzt, bis eine catch-Klausel gefunden ist, die die aktuelle Ausnahme durch die Benennung einer Ausnahmenklasse behandeln kann, die zur selben Klasse oder einer Basisklasse des Laufzeittyps der ausgelösten Ausnahme gehört. Eine catch-Klausel, die keine Ausnahmeklasse benennt, kann jede Ausnahme behandeln.

Wird eine passende catch-Klausel gefunden, übergibt das System die Steuerung an die erste Anweisung in der catch-Klausel. Vor dem Starten der catch-Klausel führt das System nacheinander zunächst alle mit solchen try-Anweisungen verknüpften finally-Klauseln aus, die einen höheren Schachtelungsgrad als die diejenige aufweisen, die die Ausnahme empfangen hat.

Wenn keine übereinstimmende catch-Klausel gefunden wird, tritt eines der zwei folgenden Ereignisse ein:

* Wenn bei der Suche nach einer übereinstimmenden catch-Klausel ein statischer Konstruktor (§10.12) oder ein statischer Feldinitialisierer gefunden wird, wird an dem Punkt, an dem der Aufruf des statischen Konstruktors ausgelöst wurde, eine Ausnahme des Typs System.TypeInitializationException ausgegeben. Die innere Ausnahme von System.TypeInitializationException enthält die Ausnahme, die ursprünglich ausgelöst wurde.
* Wenn die Suche nach übereinstimmenden catch-Klauseln den Code erreicht, der ursprünglich den Thread gestartet hat, wird die Ausführung des Threads beendet. Die Auswirkung dieses Abbruchs hängt von der Implementierung ab.

Ausnahmen, die während der Ausführung von Destruktoren auftreten, verlangen besondere Aufmerksamkeit. Wenn eine Ausnahme während der Destruktorausführung auftritt und nicht abgefangen wird, wird die Ausführung dieses Destruktors beendet und stattdessen der Destruktor der Basisklasse (falls vorhanden) aufgerufen. Wenn (wie beim Typ object) keine Basisklasse oder kein Basisklassendestruktor vorhanden ist, wird die Ausnahme ignoriert.

## Häufige Ausnahmeklassen

Die folgenden Ausnahmen werden von bestimmten C#-Operationen ausgelöst.

|  |  |
| --- | --- |
| System.ArithmeticException | Eine Basisklasse für Ausnahmen, die während arithmetischer Operationen ausgelöst werden, z. B. System.DivideByZeroException und System.OverflowException. |
| System.ArrayTypeMismatchException | Wird ausgelöst, wenn die Speicherung in einem Array fehlschlägt, weil der tatsächliche Typ des zu speichernden Elements nicht mit dem tatsächlichen Typ des Arrays kompatibel ist. |
| System.DivideByZeroException | Wird ausgelöst, wenn versucht wird, einen ganzzahligen Wert durch null zu teilen. |
| System.IndexOutOfRangeException | Wird ausgelöst, wenn versucht wird, ein Array mit einem Index zu indizieren, der kleiner als 0 ist oder außerhalb der Grenzen des Arrays liegt. |
| System.InvalidCastException | Wird ausgelöst, wenn eine explizite Konvertierung von einem Basistyp oder einer Schnittstelle in einen abgeleiteten Typ zur Laufzeit fehlschlägt. |
| System.NullReferenceException | Wird ausgegeben, wenn ein null-Verweis so verwendet wird, dass das Objekt, auf das verwiesen wurde, erforderlich ist. |
| System.OutOfMemoryException | Wird ausgelöst, wenn ein Versuch der Speicherbelegung (mithilfe von new) fehlschlägt. |
| System.OverflowException | Wird ausgelöst, wenn ein Überlauf einer arithmetischen Operation in einem checked-Kontext auftritt. |
| System.StackOverflowException | Wird ausgelöst, wenn der Ausführungsstapel durch zu viele anstehende Methodenaufrufe ausgelastet ist. Dies ist normalerweise ein Zeichen für zu tiefe oder ungebundene Rekursionen. |
| System.TypeInitializationException | Wird ausgelöst, wenn ein statischer Konstruktor eine Ausnahme auslöst und keine catch-Klausel vorhanden ist, um sie abzufangen. |

# Attribute

Ein großer Teil der Sprache C# ermöglicht dem Programmierer das Festlegen von Deklarationsinformationen zu den im Programm definierten Entitäten. So wird beispielsweise die Zugriffsmöglichkeit einer Methode in einer Klasse durch die Ergänzung von method-modifiers wie public, protected, internal und private festgelegt.

C# ermöglicht Programmierern auch die Einführung neuer Arten von Deklarationsinformationen, die als Attribute bezeichnet werden. Programmierer können dann diese Attribute mit verschiedenen Programmentitäten verbinden und in einer Laufzeitumgebung Attributinformationen abrufen. Ein Framework könnte beispielsweise ein HelpAttribute-Attribut definieren, das für bestimmte Programmelemente, z. B. Klassen und Methoden, verwendet werden kann und dadurch eine Zuordnung zwischen diesen Programmelementen und der Dokumentation ermöglicht.

Attribute werden über die Deklaration von attribute-Klassen (§17.1) definiert, die positionelle und benannte Parameter haben können (§17.1.2). In einem C#-Programm werden Attribute mithilfe von Attributspezifikationen (§17.2) mit Entitäten verbunden und können zur Laufzeit als Attributinstanzen (§17.3) abgerufen werden.

## Attributklassen

Eine Klasse, die direkt oder indirekt von der abstrakten Klasse System.Attribute abgeleitet wird, ist eine Attributklasse. Durch die Deklaration einer Attributklasse wird eine neue Art von Attribut definiert, das in einer Deklaration verwendet werden kann. Attributklassen werden immer mit dem Suffix Attribute benannt. Bei der Verwendung eines Attributs kann dieses Suffix enthalten sein oder ausgelassen werden.

### Attributverwendung

Mit dem AttributeUsage-Attribut (§17.4.1) wird beschrieben, wie eine Attributklasse verwendet werden kann.

AttributeUsage hat einen positionellen Parameter (§17.1.2), der es ermöglicht, die Arten von Deklarationen festzulegen, in denen eine Attributklasse verwendet werden kann. In dem Beispiel

using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]  
public class SimpleAttribute: Attribute   
{  
 ...  
}

definiert eine Attributklasse mit dem Namen SimpleAttribute, die nur in class-declarations und interface-declarations verwendet werden kann. In dem Beispiel

[Simple] class Class1 {...}

[Simple] interface Interface1 {...}

zeigt verschiedene Anwendungsmöglichkeiten des Simple-Attributs. Obwohl dieses Attribut mit dem Namen SimpleAttribute definiert wurde, kann bei der Verwendung das Suffix Attribute ausgelassen werden, wodurch der Kurzname Simple zustande kommt. Das oben aufgeführte Beispiel ist somit mit dem folgenden Code semantisch äquivalent:

[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}

AttributeUsage hat einen benannten Parameter (§17.1.2) mit dem Namen AllowMultiple, der angibt, ob das Attribut mehrmals für die entsprechende Entität angegeben werden kann. Wenn AllowMultiple für eine Attributklasse den Wert true aufweist, handelt es sich um eine Mehrfachattributklasse, die für eine Entität mehrfach angegeben werden kann. Wenn AllowMultiple für eine Attributklasse den Wert false aufweist oder nicht definiert ist, handelt es sich um eine Einfachattributklasse, die höchstens einmal in einer Entität spezifiziert werden kann.

In dem Beispiel

using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]  
public class AuthorAttribute: Attribute  
{  
 private string name;

public AuthorAttribute(string name) {  
 this.name = name;  
 }

public string Name {  
 get { return name; }  
 }  
}

definiert eine Mehrfachattributklasse mit dem Namen AuthorAttribute. In dem Beispiel

[Author("Brian Kernighan"), Author("Dennis Ritchie")]   
class Class1  
{  
 ...  
}

zeigt eine Klassendeklaration, in der das Author-Attribut zweimal verwendet wird.

AttributeUsage hat einen anderen benannten Parameter mit dem Namen Inherited, der anzeigt, ob ein auf einer Basisklasse beruhendes Attribut auch von Klassen geerbt wird, die von dieser Basisklasse abgeleitet werden. Wenn Inherited für eine Attributklasse true ist, dann ist dieses Attribut geerbt. Wenn Inherited für eine Attributklasse false ist, dann ist dieses Attribut nicht geerbt. Wenn keine Angabe gemacht wird, ist der Standardwert true.

Eine Attributklasse X ohne angefügtes AttributeUsage-Attribut, wie in

using System;

class X: Attribute {...}

ist äquivalent zu:

using System;

[AttributeUsage(  
 AttributeTargets.All,  
 AllowMultiple = false,  
 Inherited = true)  
]  
class X: Attribute {...}

### Positionelle und benannte Parameter

Attributklassen können positionelle Parameter und benannte Parameter haben. Jeder öffentliche Instanzkonstruktor für eine Attributklasse definiert eine gültige Reihe von positionellen Parametern für diese Attributklasse. Alle nicht statischen öffentlichen Felder und Eigenschaften mit Schreib- und Lesemöglichkeit, die zu einer Attributklasse gehören, definieren jeweils einen benannten Parameter für diese Attributklasse.

In dem Beispiel

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpAttribute: Attribute  
{  
 public HelpAttribute(string url) { // Positional parameter  
 ...  
 }

public string Topic { // Named parameter  
 get {...}  
 set {...}  
 }

public string Url {  
 get {...}  
 }  
}

definiert eine Attributklasse mit der Bezeichnung HelpAttribute, die einen positionellen Parameter (url) und einen benannten Parameter (Topic) besitzt. Obwohl die Eigenschaft Url öffentlich und nicht statisch ist, definiert sie keinen benannten Parameter, da sie keinen Schreib/-Lesezugriff ermöglicht.

Diese Attributklasse lässt sich wie folgt verwenden:

[Help("http://www.mycompany.com/.../Class1.htm")]  
class Class1  
{  
 ...  
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]  
class Class2  
{  
 ...  
}

### Attributparametertypen

Die Typen der positionellen und benannten Parameter für eine Attributklasse sind auf die Attributparametertypen beschränkt. Dies sind im Einzelnen:

* Einer der folgenden Typen: bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort.
* Der object-Typ.
* Der System.Type-Typ.
* Ein Enumerationstyp, sofern er und die Typen, in denen er geschachtelt ist, die Möglichkeit des öffentlichen Zugriffs bieten (§17.2).
* Eindimensionale Arrays der oben genannten Typen.

Ein Konstruktorargument oder ein öffentliches Feld, das keinen dieser Typen enthält, kann in einer Attributspezifikation nicht als positioneller oder benannter Parameter verwendet werden.

## Attributspezifikation

Die Attributspezifikation entspricht der Anwendung eines zuvor definierten Attributs auf eine Deklaration. Ein Attribut ist ein Teil einer zusätzlichen Information, die für eine Deklaration angegeben wird. Attribute können sowohl für einen globalen Gültigkeitsbereich festgelegt werden (um Attribute für die Assembly oder das Modul zu definieren) als auch für type-declarations (§9.6), class-member-declarations (§10.1.5), interface-member-declarations (§13.2), struct-member-declarations (§11.2), enum-member-declarations (§14.3), accessor-declarations (§10.7.2), event-accessor-declarations (§10.8.1) und formal-parameter-lists (§10.6.1).

Attribute werden in Attributabschnitten festgelegt. Ein Attributabschnitt besteht aus zwei eckigen Klammern, die eine durch Trennzeichen getrennte Liste einschließen, die ein oder mehrere Attribute enthalten kann. Die Reihenfolge, in der die Attribute in einer solchen Liste festgelegt und in der die mit derselben Programmentität verknüpften Abschnitte angeordnet sind, hat keinerlei Auswirkungen. Beispielsweise sind die Attributspezifikationen [A][B], [B][A], [A, B] und [B, A] äquivalent.

global-attributes:  
global-attribute-sections

global-attribute-sections:  
global-attribute-section  
global-attribute-sections global-attribute-section

global-attribute-section:  
[ global-attribute-target-specifier attribute-list ]  
[ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:  
global-attribute-target :

global-attribute-target:  
assembly  
module

attributes:  
attribute-sections

attribute-sections:  
attribute-section  
attribute-sections attribute-section

attribute-section:  
[ attribute-target-specifieropt attribute-list ]  
[ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:  
attribute-target :

attribute-target:  
field  
event  
method  
param  
property  
return  
type

attribute-list:  
attribute  
attribute-list , attribute

attribute:  
attribute-name attribute-argumentsopt

attribute-name:  
 type-name

attribute-arguments:  
( positional-argument-listopt )  
( positional-argument-list , named-argument-list )  
( named-argument-list )

positional-argument-list:  
positional-argument  
positional-argument-list , positional-argument

positional-argument:  
argument-nameopt attribute-argument-expression

named-argument-list:  
named-argument  
named-argument-list , named-argument

named-argument:  
identifier = attribute-argument-expression

attribute-argument-expression:  
expression

Ein Attribut besteht aus einem attribute-name und einer optionalen Liste positioneller und benannter Argumente. Die positionellen Argumente (falls vorhanden) stehen vor den benannten Argumenten. Ein positionelles Argument besteht aus einer attribute-argument-expression. Ein benanntes Argument besteht aus einem Namen, dem ein Gleichheitszeichen und eine attribute-argument-expression folgt, die alle denselben Regeln unterliegen wie einfache Zuweisungen. Die Reihenfolge der benannten Argumente ist irrelevant.

Durch den attribute-name wird die Attributklasse gekennzeichnet. Wenn die Form von attribute-name die von type-name aufweist, muss dieser Name auf eine Attributklasse verweisen. Andernfalls wird ein Kompilierungsfehler ausgelöst. In dem Beispiel

class Class1 {}

[Class1] class Class2 {} // Error

verursacht einen Kompilierungsfehler, da hier versucht wird, Class1 als Attributklasse zu verwenden, obwohl Class1 keine Attributklasse ist.

In bestimmten Kontexten ist die Spezifikation eines Attributs für mehrere Ziele zulässig. Ein Programm kann das Ziel durch Einbindung eines attribute-target-specifier explizit angeben. Wenn ein Attribut auf globaler Ebene positioniert wird, ist ein global-attribute-target-specifier erforderlich. In allen anderen Fällen wird ein sinnvoller Standardwert angewendet. Um den Standardwert in bestimmten, nicht eindeutigen Fällen zu bestätigen oder zu überschreiben (oder den Standardwert in eindeutigen Fällen einfach zu bestätigen), kann jedoch ein attribute-target-specifier verwendet werden. Daher können attribute-target-specifiers außer auf globaler Ebene meist ausgelassen werden. Die potenziell mehrdeutigen Kontexte werden auf folgende Weise behandelt:

* Ein im globalen Gültigkeitsbereich spezifiziertes Attribut kann entweder auf die Zielassembly oder das Zielmodul angewendet werden. Für diesen Kontext ist kein Standard vorhanden, daher ist hier immer ein attribute-target-specifier erforderlich. Das Vorhandensein des attribute-target-specifier assembly zeigt an, dass das Attribut auf die Zielassembly angewendet wird. Dementsprechend wird bei vorhandenem attribute-target-specifier module das Attribut auf das Zielmodul angewendet.
* Ein Attribut, das für eine Delegatdeklaration spezifiziert wurde, kann entweder auf den deklarierten Delegaten oder seinen Rückgabewert angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf den Delegaten angewendet. Das Vorhandensein des attribute-target-specifier type zeigt an, dass das Attribut auf den Delegaten angewendet wird. Dementsprechend wird bei vorhandenem attribute-target-specifier return das Attribut auf den Rückgabewert angewendet.
* Ein Attribut, das für eine Methodendeklaration spezifiziert wurde, kann entweder auf die deklarierte Methode oder ihren Rückgabewert angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf die Methode angewendet. Das Vorhandensein des attribute-target-specifier method zeigt an, dass das Attribut auf die Methode angewendet wird. Dementsprechend wird bei vorhandenem attribute-target-specifier return das Attribut auf den Rückgabewert angewendet.
* Ein Attribut, das für eine Operatordeklaration angegeben wurde, kann entweder auf den deklarierten Operator oder seinen Rückgabewert angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf den Operator angewendet. Das Vorhandensein des attribute-target-specifier method zeigt an, dass das Attribut auf den Operator angewendet wird. Dementsprechend wird bei vorhandenem attribute-target-specifier return das Attribut auf den Rückgabewert angewendet.
* Ein für eine Ereignisdeklaration spezifiziertes Attribut, bei dem Ereignisaccessoren ausgelassen werden, kann auf das deklarierte Ereignis, das zugeordnete Feld (bei einem nicht abstrakten Ereignis) oder auf die zugeordneten Methoden zum Hinzufügen sowie Entfernen angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf das Ereignis angewendet. Das Vorhandensein des attribute-target-specifier event zeigt an, dass das Attribut auf das Ereignis angewendet wird. Dementsprechend wird das Attribut bei vorhandenem attribute-target-specifier field auf das Feld und bei vorhandenem attribute-target-specifier method auf die Methoden angewendet.
* Ein Attribut, das für die Deklaration eines get-Accessors für eine Eigenschaft- oder eine Indexerdeklaration spezifiziert wird, kann entweder auf die zugehörige Methode oder ihren Rückgabewert angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf die Methode angewendet. Das Vorhandensein des attribute-target-specifier method zeigt an, dass das Attribut auf die Methode angewendet wird. Dementsprechend wird bei vorhandenem attribute-target-specifier return das Attribut auf den Rückgabewert angewendet.
* Ein Attribut, das für einen set-Accessor einer Eigenschaften- oder Indexerdeklaration spezifiziert wird, kann entweder auf die zugehörige Methode oder ihren allein stehenden impliziten Parameter angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf die Methode angewendet. Das Vorhandensein des attribute-target-specifier method zeigt an, dass das Attribut auf die Methode angewendet wird. Dementsprechend wird das Attribut bei vorhandenem attribute-target-specifier param auf den Parameter und bei vorhandenem attribute-target-specifier return auf den Rückgabewert angewendet.
* Ein Attribut, das für die Deklaration eines Accessors zum Hinzufügen oder Entfernen für eine Ereignisdeklaration spezifiziert wird, kann entweder auf die zugehörige Methode oder ihren Parameter angewendet werden. Bei nicht vorhandenem attribute-target-specifier wird das Attribut auf die Methode angewendet. Das Vorhandensein des attribute-target-specifier method zeigt an, dass das Attribut auf die Methode angewendet wird. Dementsprechend wird das Attribut bei vorhandenem attribute-target-specifier param auf den Parameter und bei vorhandenem attribute-target-specifier return auf den Rückgabewert angewendet.

In anderen Kontexten ist die Einbindung eines attribute-target-specifier zulässig, aber nicht erforderlich. Bei einer Klassendeklaration kann der type des Spezifizierers beispielsweise angegeben oder ausgelassen werden:

[type: Author("Brian Kernighan")]  
class Class1 {}

[Author("Dennis Ritchie")]  
class Class2 {}

Die Angabe eines ungültigen attribute-target-specifier verursacht einen Fehler. Der param-Spezifizierer kann beispielsweise nicht in einer Klassendeklaration verwendet werden:

[param: Author("Brian Kernighan")] // Error  
class Class1 {}

Attributklassen werden immer mit dem Suffix Attribute benannt. Bei einem attribute-name der Form type-name kann dieses Suffix enthalten sein oder ausgelassen werden. Falls eine Attributklasse mit und ohne dieses Suffix vorhanden ist, tritt eine Mehrdeutigkeit auf, die zu einem Kompilierungsfehler führt. Wenn es sich im attribute-name an der am weitesten rechts gelegenen Position um einen wörtlichen identifier (§2.4.2) handelt, dann kann nur ein Attribut ohne ein Suffix zugeordnet werden, wodurch diese Art der Mehrdeutigkeit aufgehoben wird. In dem Beispiel

using System;

[AttributeUsage(AttributeTargets.All)]  
public class X: Attribute  
{}

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Error: ambiguity  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Refers to X  
class Class3 {}

[@XAttribute] // Refers to XAttribute  
class Class4 {}

zeigt zwei Attributklassen mit den Namen X und XAttribute. Das [X]-Attribut ist mehrdeutig, da es sich auf X oder XAttribute beziehen kann. Mithilfe eines wörtlichen Bezeichners kann in diesem seltenen Fall der genaue Bezug angegeben werden. Das Attribut [XAttribute] ist somit nicht mehrdeutig (weil kein Attributklassenname XAttributeAttribute existiert). Wenn die Deklaration für die X-Klasse entfernt wird, verweisen beide Attribute auf die Klasse mit dem Namen XAttribute:

using System;

[AttributeUsage(AttributeTargets.All)]  
public class XAttribute: Attribute  
{}

[X] // Refers to XAttribute  
class Class1 {}

[XAttribute] // Refers to XAttribute  
class Class2 {}

[@X] // Error: no attribute named "X"  
class Class3 {}

Die mehrfache Verwendung einer Einfachattributklasse in derselben Entität verursacht einen Kompilierungsfehler. In dem Beispiel

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class HelpStringAttribute: Attribute  
{  
 string value;

public HelpStringAttribute(string value) {  
 this.value = value;  
 }

public string Value {  
 get {...}  
 }  
}

[HelpString("Description of Class1")]  
[HelpString("Another description of Class1")]  
public class Class1 {}

verursacht einen Kompilierungsfehler, da hier versucht wird, HelpString in der Deklaration von Class1 mehrfach zu verwenden.

Ein Ausdruck E ist eine attribute-argument-expression, wenn folgende Bedingungen erfüllt werden:

* Der Typ von E ist ein Attributparametertyp (§17.1.3).
* Bei der Kompilierung kann der Wert von E in einen der folgenden Ausdrücke aufgelöst werden:
* Einen konstanten Wert.
* Ein System.Type-Objekt.
* Ein eindimensionales Array des Typs attribute-argument-expression.

Beispiel:

using System;

[AttributeUsage(AttributeTargets.Class)]  
public class TestAttribute: Attribute  
{  
 public int P1 {  
 get {...}  
 set {...}  
 }

public Type P2 {  
 get {...}  
 set {...}  
 }

public object P3 {  
 get {...}  
 set {...}  
 }  
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]  
class MyClass {}

Eine typeof-expression (§7.6.11), die als Attributargumentausdruck verwendet wird, kann auf einen nicht generischen Typ, einen geschlossen konstruierten Typ oder einen ungebundenen generischen Typ, aber nicht auf einen offenen Typ verweisen. Damit wird sichergestellt, dass der Ausdruck während der Kompilierung aufgelöst werden kann.

class A: Attribute  
{  
 public A(Type t) {...}  
}

class G<T>  
{  
 [A(typeof(T))] T t; // Error, open type in attribute  
}

class X  
{  
 [A(typeof(List<int>))] int x; // Ok, closed constructed type  
 [A(typeof(List<>))] int y; // Ok, unbound generic type  
}

## Attributinstanzen

Eine Attributinstanz ist eine Instanz, die ein Attribut zur Laufzeit darstellt. Ein Attribut wird mit einer Attributklasse, positionellen Argumenten und benannten Argumenten definiert. Eine Attributinstanz ist eine Instanz einer Attributklasse, die mit den positionellen und benannten Argumenten initialisiert wird.

Der Abruf einer Attributinstanz betrifft sowohl die Kompilierungszeit- als auch die Laufzeitverarbeitung. Darauf wird in den folgenden Abschnitten eingegangen.

### Kompilieren eines Attributs

Die Kompilierung eines attribute mit der Attributklasse T, der positional-argument-list P und der named-argument-list N besteht aus folgenden Schritten:

* Befolgen Sie die Verarbeitungsschritte für die Kompilierung einer object-creation-expression der Form new T(P). Diese Schritte verursachen entweder einen Fehler während der Kompilierung oder bestimmen einen Instanzkonstruktor C für T, der zur Laufzeit aufgerufen werden kann.
* Wenn für C kein öffentlicher Zugriff möglich ist, wird ein Fehler während der Kompilierung ausgelöst.
* Für jedes named-argument Arg in N gilt:
* Name soll der identifier des named-argument Arg sein.
* Name muss ein nicht statisches öffentliches Feld mit Lese-/Schreibzugriff oder eine solche Eigenschaft für T bezeichnen. Wenn für T kein Feld und keine Eigenschaft dieser Art vorhanden ist, tritt ein Fehler während der Kompilierung auf.
* Übernehmen Sie die folgenden Informationen für die Laufzeitinstanziierung des Attributs: Die Attributklasse T, den Instanzkonstruktor C für T, die positional-argument-list P und die named-argument-list N.

### Abrufen einer Attributinstanz zur Laufzeit

Die Kompilierung eines attribute erzeugt eine Attributklasse T, einen Instanzkonstruktor C für T, eine positional-argument-list P und eine named-argument-list N. Wenn diese Informationen vorliegen, kann ein Attribut mithilfe folgender Schritte zur Laufzeit abgerufen werden:

* Befolgen Sie die Verarbeitungsschritte für die Ausführung einer object-creation-expression der Form new T(P), und verwenden Sie dabei den bei der Kompilierung bestimmten Instanzkonstruktor C. Diese Schritte führen entweder zu einer Ausnahme oder zur Erstellung einer Instanz O für T.
* Für jedes named-argument Arg in N gilt in der genannten Reihenfolge:
* Name soll der identifier des named-argument Arg sein. Wenn Name nicht ein nicht statisches öffentliches Feld mit Lese-/Schreibzugriff oder eine solche Eigenschaft für O bezeichnet, wird eine Ausnahme ausgelöst.
* Value soll das Ergebnis der Auswertung der attribute-argument-expression von Arg sein.
* Wenn Name ein Feld für O bezeichnet, legen Sie für dieses Feld den Wert Value fest.
* Andernfalls bezeichnet Name eine Eigenschaft für O. Legen Sie für diese Eigenschaft den Wert Value fest.
* Das Ergebnis ist O, eine Instanz der Attributklasse T, die mit der positional-argument-list P und der named-argument-list N initialisiert wurde.

## Reservierte Attribute

Einige wenige Attribute wirken sich in gewisser Weise auf die Programmiersprache aus. Dazu gehören:

* System.AttributeUsageAttribute (§17.4.1), das verwendet wird, um zu beschreiben, wie eine Attributklasse verwendet werden kann.
* System.Diagnostics.ConditionalAttribute (§17.4.2), das zur Definition bedingter Methoden verwendet wird.
* System.ObsoleteAttribute (§17.4.3), das verwendet wird, um einen Member als veraltet zu kennzeichnen.
* System.Runtime.CompilerServices.CallerLineNumberAttribute, System.Runtime.CompilerServices.CallerFilePathAttribute und System.Runtime.CompilerServices.CallerMemberNameAttribute (§17.4.4), die verwendet werden, um Informationen über den aufrufenden Kontext für optionale Parameter bereitzustellen.

### Das AttributeUsage-Attribut

Das AttributeUsage-Attribut wird verwendet, um zu beschreiben, auf welche Weise die Attributklasse verwendet werden kann.

Eine Klasse, die mit dem AttributeUsage-Attribut ergänzt ist, muss direkt oder indirekt von System.Attribute abgeleitet sein. Andernfalls wird ein Kompilierungsfehler ausgelöst.

namespace System  
{  
 [AttributeUsage(AttributeTargets.Class)]  
 public class AttributeUsageAttribute: Attribute  
 {  
 public AttributeUsageAttribute(AttributeTargets validOn) {...}

public virtual bool AllowMultiple { get {...} set {...} }

public virtual bool Inherited { get {...} set {...} }

public virtual AttributeTargets ValidOn { get {...} }  
 }

public enum AttributeTargets  
 {  
 Assembly = 0x0001,  
 Module = 0x0002,  
 Class = 0x0004,  
 Struct = 0x0008,  
 Enum = 0x0010,  
 Constructor = 0x0020,  
 Method = 0x0040,  
 Property = 0x0080,  
 Field = 0x0100,  
 Event = 0x0200,  
 Interface = 0x0400,  
 Parameter = 0x0800,  
 Delegate = 0x1000,  
 ReturnValue = 0x2000,

All = Assembly | Module | Class | Struct | Enum | Constructor |   
 Method | Property | Field | Event | Interface | Parameter |   
 Delegate | ReturnValue  
 }  
}

### Das Conditional-Attribut

Das Attribut Conditional ermöglicht die Definition von bedingten Methoden und bedingten Attributklassen.

namespace System.Diagnostics  
{  
 [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,  
 AllowMultiple = true)]  
 public class ConditionalAttribute: Attribute  
 {  
 public ConditionalAttribute(string conditionString) {...}

public string ConditionString { get {...} }  
 }  
}

#### Bedingte Methoden

Mit dem Conditional-Attribut deklarierte Methoden sind bedingte Methoden. Das Conditional-Attribut kennzeichnet eine Bedingung durch Prüfung eines bedingten Kompilierungssymbols. Aufrufe einer bedingten Methode werden je nachdem, ob dieses Symbol an der Stelle des Aufrufs definiert ist, einbezogen oder ausgelassen. Ist das Symbol definiert, wird der Aufruf berücksichtigt; andernfalls wird er (einschließlich der Auswertung des Empfängers sowie der Parameter des Aufrufs) nicht berücksichtigt.

Eine bedingte Methode unterliegt folgenden Einschränkungen:

* Die bedingte Methode muss eine Methode in einer class-declaration oder struct-declaration sein. Wenn das Conditional-Attribut für eine Methode in einer Schnittstellendeklaration festgelegt wird, tritt ein Kompilierungsfehler auf.
* Die bedingte Methode muss den Rückgabetyp void aufweisen.
* Die bedingte Methode darf nicht mit dem override-Modifizierer gekennzeichnet sein. Eine bedingte Methode kann jedoch mit dem virtual-Modifizierer gekennzeichnet sein. Überschreibungen einer solchen Methode sind implizit bedingt und dürfen nicht explizit mit einem Conditional-Attribut gekennzeichnet sein.
* Die bedingte Methode darf keine Implementierung einer Schnittstellenmethode sein. Andernfalls wird ein Kompilierungsfehler ausgelöst.

Darüber hinaus tritt während der Kompilierung ein Fehler auf, wenn eine bedingte Methode in einer delegate-creation-expression verwendet wird. In dem Beispiel

#define DEBUG

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void M() {  
 Console.WriteLine("Executed Class1.M");  
 }  
}

class Class2  
{  
 public static void Test() {  
 Class1.M();  
 }  
}

deklariert Class1.M als bedingte Methode. Die Test-Methode von Class2 ruft diese Methode auf. Da das bedingte Kompilierungssymbol DEBUG definiert ist, wird beim Aufruf von Class2.Test M aufgerufen. Wäre das Symbol DEBUG nicht definiert worden, würde Class2.Test nicht Class1.M aufrufen.

Beachten Sie, dass Einschluss oder Ausschluss eines Aufrufs einer bedingten Methode durch die bedingten Kompilierungssymbole an der Stelle des Aufrufs gesteuert werden. In dem Beispiel

Datei class1.cs:

using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public static void F() {  
 Console.WriteLine("Executed Class1.F");  
 }  
}

Datei class2.cs:

#define DEBUG

class Class2  
{  
 public static void G() {  
 Class1.F(); // F is called  
 }  
}

Datei class3.cs:

#undef DEBUG

class Class3  
{  
 public static void H() {  
 Class1.F(); // F is not called  
 }  
}

enthalten die Klassen Class2 und Class3 jeweils Aufrufe der bedingten Class1.F-Methode, die davon abhängig sind, ob DEBUG definiert ist. Da dieses Symbol nur im Kontext von Class2, nicht jedoch in dem von Class3 definiert ist, wird der Aufruf von F in Class2 berücksichtigt, während er in Class3 ausgelassen wird.

Die Verwendung von bedingten Methoden in einer Vererbungskette kann leicht unübersichtlich werden. Die Aufrufe von bedingten Methoden über base in der Form base.M unterliegen den Regeln für normale Aufrufe bedingter Methoden. In dem Beispiel

Datei class1.cs:

using System;  
using System.Diagnostics;

class Class1   
{  
 [Conditional("DEBUG")]  
 public virtual void M() {  
 Console.WriteLine("Class1.M executed");  
 }  
}

Datei class2.cs:

using System;

class Class2: Class1  
{  
 public override void M() {  
 Console.WriteLine("Class2.M executed");  
 base.M(); // base.M is not called!  
 }  
}

Datei class3.cs:

#define DEBUG

using System;

class Class3  
{  
 public static void Test() {  
 Class2 c = new Class2();  
 c.M(); // M is called  
 }  
}

enthält Class2 einen Aufruf von M, das in der zugehörigen Basisklasse definiert ist. Der Aufruf wird jedoch nicht berücksichtigt, da die Basismethode davon abhängt, ob das Symbol DEBUG vorhanden ist, das in diesem Fall nicht definiert ist. Aus diesem Grund schreibt die Methode ausschließlich "Class2.M executed" in die Konsole. Derartige Probleme lassen sich jedoch durch eine sachgerechte Verwendung von pp-declarations vermeiden.

#### Bedingte Attributklassen

Eine mit einem oder mehreren Conditional-Attributen versehene Attributklasse (§17.1) ist eine bedingte Attributklasse. Folglich ist eine bedingte Attributklasse den bedingten Kompilierungssymbol zugeordnet, die in deren Conditional-Attributen deklariert sind. In diesem Beispiel

using System;  
using System.Diagnostics;  
[Conditional("ALPHA")]  
[Conditional("BETA")]  
public class TestAttribute : Attribute {}

wird TestAttribute als bedingte Attributklasse deklariert, die den bedingten Kompilierungssymbolen ALPHA und BETA zugeordnet ist.

Attributspezifikationen (§17.2) eines bedingten Attributs werden eingeschlossen, wenn eines oder mehrere der zugeordneten bedingten Kompilierungssymbole an der Stelle der Spezifikation definiert sind, andernfalls wird die Attributspezifikation ausgelassen.

Beachten Sie, dass Einschluss oder Ausschluss einer Attributspezifikation einer bedingten Attributklasse durch die bedingten Kompilierungssymbole an der Stelle der Spezifikation gesteuert werden. In dem Beispiel

Datei test.cs:

using System;  
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}

Datei class1.cs:

#define DEBUG

[Test] // TestAttribute is specified

class Class1 {}

Datei class2.cs:

#undef DEBUG

[Test] // TestAttribute is not specified

class Class2 {}

sind die Klassen Class1 und Class2 jeweils mit dem Attribut Test versehen, das dadurch bedingt ist, ob DEBUG definiert ist. Da dieses Symbol nur im Kontext von Class1, nicht jedoch in dem von Class2 definiert ist, wird die Spezifikation des Test-Attributs in Class1 eingeschlossen, während die Spezifikation des Test-Attributs in Class2 ausgelassen wird.

### Das Obsolete-Attribut

Das Obsolete-Attribut wird zur Kennzeichnung von Typen und Typmembern verwendet, die nicht mehr verwendet werden sollen.

namespace System  
{  
 [AttributeUsage(  
 AttributeTargets.Class |   
 AttributeTargets.Struct |  
 AttributeTargets.Enum |   
 AttributeTargets.Interface |   
 AttributeTargets.Delegate |  
 AttributeTargets.Method |   
 AttributeTargets.Constructor |  
 AttributeTargets.Property |   
 AttributeTargets.Field |  
 AttributeTargets.Event,  
 Inherited = false)  
 ]  
 public class ObsoleteAttribute: Attribute  
 {  
 public ObsoleteAttribute() {...}

public ObsoleteAttribute(string message) {...}

public ObsoleteAttribute(string message, bool error) {...}

public string Message { get {...} }

public bool IsError { get {...} }  
 }  
}

Wenn ein Programm einen Typ oder Member verwendet, der mit dem Obsolete-Attribut versehen ist, gibt der Compiler eine Warnung oder einen Fehler aus. Insbesondere gibt der Compiler eine Warnung aus, wenn kein Fehlerparameter bereitgestellt wurde oder der bereitgestellte Fehlerparameter den Wert false hat. Wenn der Fehlerparameter angegeben wurde und den Wert true hat, gibt der Compiler einen Fehler aus.

In dem Beispiel

[Obsolete("This class is obsolete; use class B instead")]  
class A  
{  
 public void F() {}  
}

class B  
{  
 public void F() {}  
}

class Test  
{  
 static void Main() {  
 A a = new A(); // Warning  
 a.F();  
 }  
}

ist die Klasse A mit dem Obsolete-Attribut ergänzt. Die Verwendung von A in Main löst jedes Mal eine Warnung aus, die angibt, dass diese Klasse veraltet ist und stattdessen die Klasse B verwendet werden soll.

### Aufruferinformationsattribute

Für Zwecke wie Protokollierung und Berichterstellung ist es manchmal nützlich, dass ein Funktionsmember bestimmte Kompilierungsinformationen über den aufrufenden Code abruft. Die Aufruferinformationsattribute bieten eine Möglichkeit, solche Informationen transparent zu übergeben.

Wenn ein optionaler Parameter mit einem der Aufruferinformationsattribute versehen ist, führt das Auslassen des entsprechenden Arguments in einem Aufruf nicht unbedingt dazu, dass der Standardparameterwert ersetzt wird. Wenn stattdessen die angegebenen Informationen über den aufrufenden Kontext verfügbar sind, werden diese Informationen als Argumentwert übergeben.

Beispiel:

using System.Runtime.CompilerServices

…

public void Log(  
 [CallerLineNumber] int line = -1,  
 [CallerFilePath] string path = null,  
 [CallerMemberName] string name = null  
)  
{  
 Console.WriteLine((line < 0) ? "No line" : "Line "+ line);  
 Console.WriteLine((path == null) ? "No file path" : path);  
 Console.WriteLine((name == null) ? "No member name" : name);  
}

Ein Aufruf von Log() ohne Argumente würde die Zeilennummer und den Dateipfad des Aufrufs sowie den Namen des Members, in dem der Aufruf erfolgt ist, ausgeben.

Aufruferinformationsattribute können für optionale Parameter überall auftreten, einschließlich in Delegatdeklarationen. Die einzelnen Aufruferinformationsattribute haben jedoch Einschränkungen bezüglich der Typen von Parametern, die sie attributieren können, sodass immer eine implizite Konvertierung von einem ersetzten Wert in den Parametertyp vorhanden ist.

Wenn das gleiche Aufruferinformationsattribut für einen Parameter des definierenden und des implementierenden Teils der Deklaration einer partiellen Methode vorhanden ist, tritt ein Fehler auf. Es werden nur Aufruferinformationsattribute im definierenden Teil angewendet, während die Aufruferinformationsattribute, die nur im implementierenden Teil vorkommen, ignoriert werden.

Die Überladungsauflösung wird durch Aufruferinformationen nicht beeinflusst. Werden die attributierten optionalen Parameter weiterhin im Quellcode des Aufrufers ausgelassen, werden diese Paramenter von der Überladungsauflösung auf die gleiche Weise ignoriert wie andere ausgelassene Parameter (§7.5.3).

Aufruferinformationen werden nur ersetzt, wenn eine Funktion explizit im Quellcode aufgerufen wird. Implizite Aufrufe, z. B. eines übergeordneten Konstruktors, haben keine Quellposition und ersetzen keine Aufruferinformationen. Außerdem werden Aufruferinformationen nicht durch Aufrufe ersetzt, die dynamisch gebunden sind. Wird ein mit Aufruferinformationen attributierter Parameter in solchen Fällen ausgeslassen, wird stattdessen der angegebene Standardwert des Parameters verwendet.

Eine Ausnahme sind Abfrageausdrücke. Diese werden als syntaktische Erweiterungen angesehen, und wenn die Aufrufe, die sie erweitern, optionale Parameter mit Aufruferinformationsattributen auslassen, werden die Aufruferinformationen ersetzt. Der verwendete Ort entspricht dem Ort der Abfrageklausel, aus der der Aufruf generiert wurde.

Bei Angabe mehrerer Aufruferinformationsattribute für einen bestimmten Parameter werden diese in der folgenden Reihenfolge bevorzugt: CallerLineNumber, CallerFilePath, CallerMemberName.

#### Das CallerLineNumber-Attribut

Das System.Runtime.CompilerServices.CallerLineNumberAttribute ist für optionale Parameter zulässig, wenn eine implizite Standardkonvertierung (§6.3.1) vom konstanten Wert int.MaxValue in den Typ des Parameters vorhanden ist. Dadurch wird sichergestellt, dass alle nicht negativen Zeilennummern bis zu diesem Wert ohne Fehler übergeben werden können.

Wenn ein Funktionsaufruf von einer Position im Quellcode einen optionalen Parameter mit dem CallerLineNumberAttribute auslässt, wird ein numerisches Literal, das die Zeilennummer der Position darstellt, als Argument für den Aufruf anstelle des Standardparameterwerts verwendet.

Wenn der Aufruf mehrere Zeilen umfasst, ist die Auswahl der Zeile von der Implementierung abhängig.

Beachten Sie, dass die Zeilennummer durch #line-Direktiven beeinflusst werden kann (§2.5.7).

#### Das CallerFilePath-Attribut

Das System.Runtime.CompilerServices.CallerFilePathAttribute ist für optionale Parameter zulässig, wenn eine implizite Standardkonvertierung (§6.3.1) von string in den Typ des Parameters vorhanden ist.

Wenn ein Funktionsaufruf von einer Position im Quellcode einen optionalen Parameter mit dem CallerFilePathAttribute auslässt, wird ein Zeichenfolgenliteral, das den Dateipfad der Position darstellt, als Argument für den Aufruf anstelle des Standardparameterwerts verwendet.

Das Format des Dateipfads ist von der Implementierung abhängig.

Beachten Sie, dass der Dateipfad durch #line-Direktiven beeinflusst werden kann (§2.5.7).

#### Das CallerMemberName-Attribut

Das System.Runtime.CompilerServices.CallerMemberNameAttribute ist für optionale Parameter zulässig, wenn eine implizite Standardkonvertierung (§6.3.1) from string in denm Typ des Parameters vorhanden ist.

Wenn ein Funktionsaufruf von einer Position innerhalb des Texts eines Funktionsmembers oder innerhalb eines Attributs, das auf den Funktionsmember selbst oder seinen Rückgabetyp, seine Parameter oder Typparameter im Quellcode angewendet wird, einen optionalen Parameter mit dem CallerMemberNameAttribute auslässt, wird ein Zeichenfolgenliteral, das den Namen dieses Members darstellt, als Argument für den Aufruf anstelle des Standardparameterwerts verwendet.

Für Aufrufe, die innerhalb generischer Methoden erfolgen, wird nur der Methodenname selbst ohne die Typparameterliste verwendet.

Für Aufrufe, die innerhalb expliziter Schnittstellenmemberimplementierungen erfolgen, wird nur der Methodenname selbst ohne die vorangehende Schnittstellenqualifikation verwendet.

Für Aufrufe, die innerhalb von Eigenschaften- oder Ereignisaccessoren erfolgen, entspricht der verwendete Membername dem der Eigenschaft oder des Ereignisses selbst.

Für Aufrufe, die innerhalb von Indexeraccessoren erfolgen, wird der Membername verwendet, der von einem IndexerNameAttribute (§17.5.2.1) im Indexermember bereitgestellt wird, soweit vorhanden, oder andernfalls der Standardname Item.

Für Aufrufe, die innerhalb von Deklarationen von Instanzkonstruktoren, statischen Konstruktoren, Destruktoren und Operatoren erfolgen, ist der verwendete Membername von der Implementierung abhängig.

## Attribute für die Interoperabilität

Hinweis: Dieser Abschnitt bezieht sich nur auf die Microsoft .NET-Implementierung von C#.

### Interoperabilität mit COM- und Win32-Komponenten

Mit der .NET-Laufzeit wird eine Vielzahl von Attributen bereitgestellt, die die Interoperabilität von C#-Programmen mit Komponenten gewährleisten, die mithilfe von COM- und Win32-DLLs geschrieben wurden. Das DllImport-Attribut kann z. B. in einer static extern-Methode verwendet werden und gibt damit an, dass die Implementierung der Methode in einer Win32-DLL festgelegt wurde. Diese Attribute befinden sich im System.Runtime.InteropServices-Namespace, und eine detaillierte Dokumentation zu diesen Attributen finden Sie in der .NET-Laufzeitdokumentation.

### Interoperabilität mit anderen .NET-Sprachen

#### Das IndexerName-Attribut

Indexer werden in .NET mithilfe von indizierten Eigenschaften implementiert und verfügen in den .NET-Metadaten über einen Namen. Wenn für einen Indexer kein IndexerName-Attribut vorhanden ist, wird standardmäßig der Name Item verwendet. Das IndexerName-Attribut ermöglicht dem Entwickler das Überschreiben dieses Standards und das Festlegen eines anderen Namens.

namespace System.Runtime.CompilerServices.CSharp  
{  
 [AttributeUsage(AttributeTargets.Property)]  
 public class IndexerNameAttribute: Attribute  
 {  
 public IndexerNameAttribute(string indexerName) {...}

public string Value { get {...} }   
 }  
}

# Unsicherer Code

In der Kernsprache unterscheidet sich C#, wie in den vorhergehenden Abschnitten beschrieben, von C und C++ hauptsächlich durch den Wegfall von Zeigern als Datentypen. Stattdessen stellt C# Verweise bereit und ermöglicht die Erstellung von Objekten, die von einem Garbage Collector verwaltet werden. In Kombination mit weiteren Funktionen macht dieser Aufbau C# zu einer deutlich sichereren Sprache als C oder C++. In der Kernsprache von C# ist es nicht möglich, nicht initialisierte Variablen, „hängende“ Zeiger oder Ausdrücke zu verwenden, die ein Array außerhalb seiner Grenzen indizieren. Ganze Fehlerkategorien, die normalerweise in C und C++ zu Problemen führen, werden dadurch unterbunden.

Obwohl für praktisch jedes Zeigertypkonstrukt in C oder C++ ein Gegenstück in Form von Verweistypen in C# vorhanden ist, gibt es Situationen, in denen der Zugriff auf Zeigertypen notwendig wird. Kopplungen an das zugrunde liegende Betriebssystem, Zugriffe auf ein Gerät mit Speicherzuordnung oder Implementierungen von zeitkritischen Algorithmen könnten beispielsweise ohne Zugriff auf Zeiger nicht möglich oder nicht praktikabel sein. Für diese Anforderungen bietet C# die Möglichkeit, unsicheren Code zu schreiben.

Mit nicht gesichertem Code ist es möglich, Zeiger zu deklarieren und mit ihnen zu arbeiten, Konvertierungen zwischen Zeigern und Ganzzahltypen durchzuführen, um die Adressen von Variablen zu erhalten, usw. Auf gewisse Weise ähnelt das Schreiben von nicht gesichertem Code dem Schreiben von C-Code innerhalb eines C#-Programms.

Aus Sicht der Entwickler und Benutzer stellt unsicherer Code eigentlich eine „sichere“ Funktion dar. Unsicherer Code muss eindeutig mit dem unsafe-Modifizierer gekennzeichnet werden, um zu verhindern, dass Entwickler versehentlich unsichere Funktionen verwenden. Das Ausführungsmodul stellt dabei sicher, dass unsicherer Code nur in einer vertrauenswürdigen Umgebung ausgeführt werden kann.

## Unsichere Kontexte

Die nicht gesicherten Funktionen von C# stehen nur in unsicheren Kontexten zur Verfügung. Ein unsicherer Kontext wird durch die Einbindung eines unsafe-Modifizierers in die Deklaration eines Typs oder Members oder die Einbindung eines unsafe-statement eingeleitet:

* Eine Deklaration einer Klasse, Struktur, Schnittstelle oder eines Delegaten kann einen unsafe-Modifizierer enthalten, wodurch der gesamte folgende Text dieser Typdeklaration (einschließlich des Texts der Klasse, Struktur oder Schnittstelle) als unsicherer Kontext betrachtet wird.
* Deklarationen von Feldern, Methoden, Eigenschaften, Ereignissen, Indexern, Operatoren, Instanzkonstruktoren, Destruktoren oder statischen Konstruktoren können ebenfalls einen unsafe-Modifizierer enthalten, wodurch der gesamte Text dieser Memberdeklarationen als unsicherer Kontext angesehen wird.
* Eine unsafe-statement ermöglicht die Verwendung eines nicht gesicherten Kontextes innerhalb eines block. Der gesamte Text des zugeordneten block wird als unsicherer Kontext angesehen.

Die dazugehörigen Grammatikerweiterungen werden weiter unten gezeigt. Aus Platzgründen werden die in den vorangegangenen Kapiteln behandelten Ausdrücke weggelassen und durch Auslassungspunkte (...) ersetzt.

class-modifier:  
...  
unsafe

struct-modifier:  
...  
unsafe

interface-modifier:  
...  
unsafe

delegate-modifier:  
...  
unsafe

field-modifier:  
...  
unsafe

method-modifier:  
...  
unsafe

property-modifier:  
...  
unsafe

event-modifier:  
...  
unsafe

indexer-modifier:  
...  
unsafe

operator-modifier:  
...  
unsafe

constructor-modifier:  
...  
unsafe

destructor-declaration:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement

unsafe-statement:  
unsafe block

In dem Beispiel

public unsafe struct Node  
{  
 public int Value;  
 public Node\* Left;  
 public Node\* Right;  
}

führt der in der Strukturdeklaration spezifizierte unsafe-Modifizierer dazu, dass der gesamte weitere Text der Strukturdeklaration zu einem unsicheren Kontext wird. Dadurch wird es möglich, das Left-Feld und das Right-Feld als Zeigertyp zu deklarieren. Das obige Beispiel könnte auch so geschrieben werden:

public struct Node  
{  
 public int Value;  
 public unsafe Node\* Left;  
 public unsafe Node\* Right;  
}

In diesem Fall führen die unsafe-Modifizierer in den Felddeklarationen dazu, dass diese als unsichere Kontexte angesehen werden.

Im Gegensatz zur Einrichtung eines unsicheren Kontexts, der die Verwendung von Zeigern ermöglicht, hat der unsafe-Modifizierer keine Auswirkung auf einen Typ oder Member. In dem Beispiel

public class A  
{  
 public unsafe virtual void F() {  
 char\* p;  
 ...  
 }  
}

public class B: A  
{  
 public override void F() {  
 base.F();  
 ...  
 }  
}

führt der unsafe-Modifizierer der F-Methode in A dazu, dass der weitere Text von F zu einem unsicheren Kontext wird, in dem die unsicheren Funktionen der Sprache verwendet werden können. In der Überschreibung von F in B ist es nicht erforderlich, den unsafe-Modifizierer neu zu spezifizieren, es sei denn, die F-Methode in B benötigt selbst Zugriff auf unsichere Funktionen.

Dies gestaltet sich etwas anders, wenn ein Zeigertyp Teil der Signatur der Methode ist.

public unsafe class A  
{  
 public virtual void F(char\* p) {...}  
}

public class B: A  
{  
 public unsafe override void F(char\* p) {...}  
}

Da hier die Signatur von F einen Zeigertyp einschließt, kann dies nur in einem unsicheren Kontext geschrieben werden. Dieser unsichere Kontext kann definiert werden, indem die gesamte Klasse zu einem unsicheren Kontext erklärt (wie in A) oder ein unsafe-Modifizierer in die Methodendeklaration eingebunden wird (wie in B).

## Zeigertypen

In einem unsicheren Kontext kann ein type (§4) ein pointer-type, ein value-type oder ein reference-type sein. Ein *pointer-type* kann allerdings auch in einem typeof-Ausdruck (§7.6.10.6) außerhalb eines unsicheren Kontexts verwendet werden, da diese Art der Verwendung nicht unsicher ist.

type:  
...  
pointer-type

Ein pointer-type wird wie ein unmanaged-type oder mit dem Schlüsselwort void angegeben, gefolgt von einem \*-Token:

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

Der vor dem \* spezifizierte Typ in einem Zeigertyp wird als Verweistyp des Zeigertyps bezeichnet. Er steht für den Typ der Variable, auf den ein Wert des Zeigertyps verweist.

Anders als Verweise (Werte von Verweistypen) werden Zeiger nicht vom Garbage Collector nachverfolgt. Der Garbage Collector hat keine Kenntnis von Zeigern und den Daten, auf die sie zeigen. Aus diesem Grund ist es nicht zulässig, dass ein Zeiger auf einen Verweis zeigt oder auf eine Struktur, die Verweise enthält. Daher muss der Verweistyp eines Zeigers ein unmanaged-type sein.

Ein unmanaged-type kann jeder Typ sein, der kein reference-type oder konstruierter Typ ist und auf keiner Schachtelungsebene reference-type-Felder oder Felder konstruierter Typen aufweist. Ein unmanaged-type kann also einer der folgenden Typen sein:

* sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal oder bool.
* Beliebiger enum-type.
* Beliebiger pointer-type.
* Beliebiger benutzerdefinierter struct-type, der kein konstruierter Typ ist und nur Felder von unmanaged-types enthält.

Die Faustregel zur Mischung von Zeigern und Verweisen lautet: Referenten von Verweisen (Objekten) dürfen Zeiger enthalten, Referenten von Zeigern dürfen jedoch keine Verweise enthalten.

In der folgenden Tabelle werden einige Beispiele von Zeigertypen aufgeführt:

|  |  |
| --- | --- |
| **Beispiel** | **Beschreibung** |
| byte\* | Zeiger auf byte |
| char\* | Zeiger auf char |
| int\*\* | Zeiger auf Zeiger auf int |
| int\*[] | Eindimensionales Array von Zeigern auf int |
| void\* | Zeiger auf einen unbekannten Typ |

Bei einer vorhandenen Implementierung müssen alle Zeigertypen dieselbe Größe und Darstellung aufweisen.

Wenn mehrere Zeiger in derselben Deklaration deklariert werden, wird in C# im Unterschied zu C und C++ das Sternchen \* nur in Verbindung mit dem zugrunde liegenden Typ angegeben, nicht als Präfix vor jedem Zeigernamen. Beispiel:

int\* pi, pj; // NOT as int \*pi, \*pj;

Der Wert eines Zeigers vom Typ T\* stellt die Addresse einer Variablen vom Typ T dar. Der Zeigerdereferenzierungsoperator \* (§18.5.1) kann für den Zugriff auf diese Variable verwendet werden. Wenn wie im folgendem Beispiel

eine Variable P des Typs int\* vorliegt, kennzeichnet der Ausdruck \*P die int-Variable, die an der in P enthaltenen Adresse gefunden wurde.

Ein Zeiger kann wie ein Objektverweis null sein. Die Anwendung des Dereferenzierungsoperators auf einen null-Zeiger führt zu einem in der Implementierung definierten Verhalten. Bei einem Zeiger mit dem Wert null weisen alle Bits den Wert 0 auf.

Der void\*-Typ steht für einen Zeiger auf einen unbekannten Typ. Da der Referenztyp nicht bekannt ist, kann weder der Dereferenzierungsoperator auf einen Zeiger mit dem Typ void\* angewendet werden, noch kann mit einem solchen Zeiger eine arithmetische Operation ausgeführt werden. Allerdings kann ein Zeiger des Typs void\* in jeden anderen Zeigertyp umgewandelt werden (und umgekehrt).

Zeigertypen sind eine separate Typkategorie. Im Gegensatz zu Verweistypen und Werttypen erben Zeigertypen nicht von object, und es gibt keine Konvertierungen zwischen den Zeigertypen und object Das bedeutet, dass Boxing und Unboxing (§4.3) für Zeiger nicht unterstützt werden. Konvertierungen zwischen verschiedenen Zeigertypen sowie zwischen Zeigertypen und ganzzahligen Typen sind jedoch zulässig. Dies wird in §18.4 beschrieben.

Ein pointer-type kann nicht als Typargument verwendet werden (§4.4), und ein Typrückschluss (§7.5.2) löst bei generischen Methodenaufrufen, die ein Typargument als Zeigertyp abgeleitet hätten, einen Fehler aus.

Ein pointer-type kann als Typ eines flüchtigen Felds (§10.5.3) verwendet werden.

Zeiger können als ref-Parameter oder out-Parameter übergeben werden. Dies kann allerdings zu undefinierten Zuständen führen, da der Zeiger auf eine lokale Variable zeigen kann, die bei der Rückgabe der aufgerufenen Methode nicht mehr vorhanden ist, oder auf ein festes Objekt, das dann nicht mehr fest ist. Beispiel:

using System;

class Test  
{  
 static int value = 20;

unsafe static void F(out int\* pi1, ref int\* pi2) {  
 int i = 10;  
 pi1 = &i;

fixed (int\* pj = &value) {  
 // ...  
 pi2 = pj;  
 }  
 }

static void Main() {  
 int i = 10;  
 unsafe {  
 int\* px1;  
 int\* px2 = &i;

F(out px1, ref px2);

Console.WriteLine("\*px1 = {0}, \*px2 = {1}",  
 \*px1, \*px2); // undefined behavior  
 }  
 }  
}

Eine Methode kann einen Wert eines Typs zurückgeben, und dieser Typ kann ein Zeiger sein. Wenn beispielsweise ein Zeiger vorhanden ist, der auf eine fortlaufende Sequenz von int-Werten, auf die Anzahl der Elemente der Sequenz und auf einige andere int-Werte zeigt, gibt die folgende Methode bei einer Übereinstimmung die Adresse des in der Sequenz enthaltenen Werts zurück. Andernfalls ist der Rückgabewert null:

unsafe static int\* Find(int\* pi, int size, int value) {  
 for (int i = 0; i < size; ++i) {  
 if (\*pi == value)   
 return pi;  
 ++pi;  
 }  
 return null;  
}

In einem nicht gesicherten Kontext stehen mehrere Konstrukte für Operationen mit Zeigern zur Verfügung:

* Der \*-Operator kann zur Dereferenzierung von Zeigern verwendet werden (§18.5.1).
* Der ->-Operator kann über einen Zeiger zum Zugriff auf einen Member einer Struktur verwendet werden (§18.5.2).
* Der []-Operator kann zur Indizierung eines Zeigers verwendet werden (§18.5.3).
* Der &-Operator kann zur Ermittlung der Adresse einer Variablen verwendet werden (§18.5.4).
* Die Operatoren ++ und -- können zum Inkrementieren und Dekrementieren von Zeigern verwendet werden (§18.5.5).
* Die Operatoren + und - können zum Ausführen arithmetischer Operationen mit Zeigern verwendet werden (§18.5.6).
* Die Operatoren ==, !=, <, >, <= und => können zum Vergleichen von Zeigern verwendet werden (§18.5.7).
* Der stackalloc-Operator kann zum Belegen von Speicher aus der Aufrufliste verwendet werden (18.7).
* Die fixed-Anweisung kann verwendet werden, um eine Variable vorübergehend zu fixieren, damit es möglich ist, ihre Adresse zu ermitteln (§18.6).

## Feste und bewegliche Variablen

Der Adressoperator (§18.5.4) und die fixed-Anweisung (§18.6) unterteilen die Variablen in zwei Kategorien: feste Variablen und bewegliche Variablen.

Feste Variablen verbleiben in Speicherorten, die nicht durch die Operationen des Garbage Collectors beeinflusst werden. (Beispiele für feste Variablen sind lokale Variablen, Werteparameter und von dereferenzierenden Zeigern erstellte Variablen.) Andererseits befinden sich bewegliche Variablen an Speicherorten, die vom Garbage Collector verschoben oder verworfen werden können. (Beispiele für bewegliche Variablen sind Felder in Objekten und Elemente von Arrays.)

Der &-Operator (§18.5.4) ermöglicht das Ermitteln einer festen Variablen ohne Einschränkungen. Da eine bewegliche Variable vom Garbage Collector verschoben oder verworfen werden kann, ist ihre Adresse nur durch Verwendung einer fixed-Anweisung (§18.6) zu ermitteln, und diese Adresse bleibt nur für die Dauer dieser fixed-Anweisung gültig.

Eine feste Variable kann nur Folgendes sein:

* Eine Variable, die aus einem simple-name (§7.6.2) entstanden ist und die auf eine lokale Variable oder einen Wertparameter verweist, sofern die Variable nicht durch eine anonyme Funktion erfasst wird.
* Eine Variable, die aus einem member-access (§7.6.4) der Form V.I entstanden ist, wobei V eine feste Variable eines struct-type ist.
* Eine Variable, die aus einer pointer-indirection-expression (§18.5.1) der Form \*P, einem pointer-member-access (§18.5.2) der Form P->I oder einem pointer-element-access (§18.5.3) der Form P[E] entstanden ist.

Alle anderen Variablen werden als bewegliche Variablen klassifiziert.

Beachten Sie, dass ein statisches Feld als bewegliche Variable klassifiziert wird. Beachten Sie außerdem, dass ein ref-Parameter oder ein out-Parameter als bewegliche Variable klassifiziert wird, auch wenn das für den Parameter vorgegebene Argument eine feste Variable ist. Schließlich ist zu beachten, dass eine Variable, die durch die Dereferenzierung eines Zeigers entstanden ist, immer als feste Variable klassifiziert wird.

## Zeigerkonvertierungen

In einem unsicheren Kontext wird die Gruppe der verfügbaren impliziten Konvertierungen (§6.1) dahingehend erweitert, dass die folgenden impliziten Zeigerkonvertierungen enthalten sind:

* Konvertierungen eines beliebigen pointer-type in den Typ void\*.
* Konvertierungen des null-Literals in einen beliebigen pointer-type.

In einem unsicheren Kontext wird die Gruppe der verfügbaren expliziten Konvertierungen (§6.2) außerdem dahingehend erweitert, dass die folgenden expliziten Zeigerkonvertierungen enthalten sind:

* Konvertierungen eines beliebigen pointer-type in einen anderen pointer-type.
* Von sbyte, byte, short, ushort, int, uint, long oder ulong in einen beliebigen pointer-type.
* Von einem beliebigen pointer-type in sbyte, byte, short, ushort, int, uint, long oder ulong.

In einem unsicheren Kontext enthält die Gruppe der impliziten Standardkonvertierungen (§6.3.1) schließlich die folgenden Zeigerkonvertierungen:

* Konvertierungen eines beliebigen pointer-type in den Typ void\*.

Konvertierungen zwischen zwei Zeigertypen verändern niemals den tatsächlichen Zeigerwert. Eine Konvertierung eines Zeigertyps in einen anderen hat also keine Auswirkung auf die dem Zeiger zugrunde liegende Adresse.

Wenn bei der Konvertierung eines Zeigertyps in einen anderen der resultierende Zeiger nicht richtig auf den Typ ausgerichtet ist, auf den er zeigen soll, ist bei der Dereferenzierung des Ergebnisses das Verhalten nicht definiert. Normalerweise ist das Konzept "korrekt ausgerichtet" transitiv: Wenn ein Zeiger auf Typ A korrekt auf einen Zeiger auf Typ B ausgerichtet ist, der wiederum korrekt auf einen Zeiger auf Typ C ausgerichtet ist, dann ist der Zeiger auf Typ A korrekt auf den Zeiger auf Typ C ausgerichtet.

Betrachten Sie den folgenden Fall, in dem auf eine Variable eines bestimmten Typs mithilfe eines Zeigers auf einen anderen Typ zugegriffen wird.

char c = 'A';  
char\* pc = &c;  
void\* pv = pc;  
int\* pi = (int\*)pv;  
int i = \*pi; // undefined  
\*pi = 123456; // undefined

Wenn ein Zeigertyp in einen Zeiger auf ein Byte konvertiert wird, zeigt das Ergebnis auf das Byte mit der niedrigsten Adresse in der Variablen. Durch nachfolgende Inkrementierungen des Ergebnisses bis zur Größe der Variablen entstehen Zeiger auf die übrigen Bytes dieser Variablen. Die folgende Methode zeigt beispielsweise alle acht Bytes in einem double als hexadezimalen Wert an:

using System;

class Test  
{  
 unsafe static void Main() {  
 double d = 123.456e23;  
 unsafe {  
 byte\* pb = (byte\*)&d;  
 for (int i = 0; i < sizeof(double); ++i)  
 Console.Write("{0:X2} ", \*pb++);  
 Console.WriteLine();  
 }  
 }  
}

Die erzeugte Ausgabe hängt natürlich auch von der Reihenfolge der Bytes ab.

Zuordnungen zwischen Zeigern und Ganzzahlen sind durch die Implementierung definiert. Bei 32- und 64-Bit-CPU-Architekturen mit linearem Adressenbereich verhalten sich die Konvertierungen von Zeigern in oder aus ganzzahligen Typen normalerweise genauso wie die Konvertierungen von uint-Werten bzw. ulong-Werten in oder aus diesen ganzzahligen Typen.

### Zeigerarrays

In einem unsicheren Kontext können Arrays von Zeigern konstruiert werden. Nur einige der Konvertierungen, die auf andere Arraytypen angewendet werden können, sind für Zeigerarrays zulässig:

* Die implizite Verweiskonvertierung (§6.1.6) von einem beliebigen array-type in System.Array und die Schnittstellen, die dadurch implementiert werden, gilt auch für Zeigerarrays. Jeder Versuch jedoch, auf die Arrayelemente über System.Array oder die dadurch implementierten Schnittstellen zuzugreifen, führt zu einer Laufzeitausnahme, da Zeigertypen nicht in object konvertierbar sind.
* Die impliziten und expliziten Verweiskonvertierungen (§6.1.6, §6.2.4) von einem eindimensionalen Arraytyp S[] in System.Collections.Generic.IList<T> und seine generischen Basisschnittstellen sind nie auf Zeigerarrays anwendbar, da Zeigertypen nicht als Typargumente verwendet werden können und keine Konvertierungen von Zeigertypen in Nichtzeigertypen erfolgen.
* Die explizite Verweiskonvertierung (§6.2.4) von System.Array und den dadurch implementierten Schnittstellen in einen beliebigen array-type ist auf Zeigerarrays anwendbar.
* Die expliziten Verweiskonvertierungen (§6.2.4) von System.Collections.Generic.IList<S> und seiner Basisschnittstellen in einen eindimensionalen Arraytyp T[] sind nie auf Zeigerarrays anwendbar, da Zeigertypen nicht als Typargumente verwendet werden können und keine Konvertierungen von Zeigertypen in Nichtzeigertypen erfolgen.

Diese Einschränkungen bedeuten, dass die Erweiterung für die foreach-Anweisung über Arrays, wie in §8.8.4 beschrieben, nicht auf Zeigerarrays angewendet werden kann. Stattdessen wird eine foreach-Anweisung der Form

foreach (V v in x) embedded-statement

(wobei der Typ von x ein Arraytyp der Form T[,,…,], n die Anzahl der Dimensionen minus 1 und T oder V ein Zeigertyp ist) folgendermaßen mithilfe geschachtelter for-Schleifen erweitert:

{  
 T[,,…,] a = x;  
 V v;  
 for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)  
 for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)  
 …  
 for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {  
 v = (V)a.GetValue(i0,i1,…,in);  
 embedded-statement  
 }  
}

Die Variablen a, i0, i1, … in sind für x oder die embedded-statement oder beliebigen anderen Quellcode des Programms nicht sichtbar, ein Zugriff auf die Variablen ist ebenfalls nicht möglich. Die Variable v ist in der eingebetteten Anweisung schreibgeschützt. Wenn keine explizite Konvertierung (§18.4) von T (dem Elementtyp) in V vorhanden ist, wird ein Fehler generiert, und es werden keine weiteren Schritte ausgeführt. Wenn der Wert von x gleich null ist, wird zur Laufzeit System.NullReferenceException ausgelöst.

## Zeiger in Ausdrücken

In einem unsicheren Kontext kann ein Ausdruck einen Zeigertyp als Ergebnis erhalten. Außerhalb eines unsicheren Kontexts tritt jedoch ein Kompilierungsfehler auf, wenn ein Ausdruck einen Zeigertyp aufweist. Das bedeutet im Einzelnen, dass außerhalb eines unsicheren Kontexts ein Kompilierungsfehler auftritt, wenn ein simple-name (§7.6.2), ein member-access (§7.6.4), eine invocation-expression (§7.6.5) oder ein element-access (§7.6.6) einen Zeigertyp aufweist.

In einem unsicheren Kontext ermöglichen die primary-no-array-creation-expression (§7.6) und die unary-expression (§7.7) folgende zusätzliche Konstrukte:

primary-no-array-creation-expression:  
...  
pointer-member-access  
pointer-element-access  
sizeof-expression

unary-expression:  
...  
pointer-indirection-expression  
addressof-expression

Diese Konstruktionen werden in den folgenden Abschnitten beschrieben. Der Vorrang und die Orientierung der nicht gesicherten Operatoren werden von der Grammatik impliziert.

### Zeigerdereferenzierung

Eine pointer-indirection-expression besteht aus einem Sternchen (\*), gefolgt von einer unary-expression.

pointer-indirection-expression:  
\* unary-expression

Der unäre \*-Operator steht für die pointer indirection und wird verwendet, um die Variable zu ermitteln, auf die ein Zeiger verweist. Das Ergebnis der Auswertung von \*P entspricht einer Variablen des Typs T, wenn P ein Ausdruck mit dem Zeigertyp T\* ist. Die Anwendung des unären \*-Operators auf einen Ausdruck vom Typ void\* oder einen Ausdruck, der keinen Zeigertyp aufweist, verursacht einen Kompilierungsfehler.

Der Effekt der Anwendung des unären \*-Operators auf einen null-Zeiger ist durch die Implementierung definiert. Das bedeutet im Einzelnen, dass es keine Garantie für das Auslösen einer System.NullReferenceException gibt.

Wenn dem Zeiger ein ungültiger Wert zugewiesen wurde, ist das Verhalten des unären \*-Operators nicht definiert. Zu den ungültigen Werten für die Dereferenzierung eines Zeigers durch einen unären \*-Operator zählen Adressen, die falsch auf den referenzierten Typ ausgerichtet wurden (siehe Beispiel in §18.4), und Adressen von Variablen, deren Lebensdauer abgelaufen ist.

Für die endgültige Untersuchung der Zuweisung wird bei einer Variablen, die durch die Auswertung eines Ausdrucks der Form \*P entstanden ist, eine anfängliche Zuweisung angenommen (§5.3.1).

### Zeigermemberzugriff

Ein pointer-member-access besteht aus einer primary-expression, der ein "->"-Token und ein identifier sowie eine optionale type-argument-list folgen.

pointer-member-access:  
primary-expression -> identifier type-argument-listopt

Bei einem Zeigermemberzugriff der Form P->I, muss P ein Ausdruck eines beliebigen Zeigertyps (außer void\*) sein, und I muss ein für den Zugriff verfügbarer Member des Typs sein, auf den P zeigt.

Ein Zeigermemberzugriff der Form P->I wird exakt so ausgewertet wie (\*P).I. Eine Beschreibung des Zeigerdereferenzierungsoperators (\*) finden Sie in §18.5.1. Eine Beschreibung des Memberzugriffsoperators (.) finden Sie in §7.6.4.

In dem Beispiel

using System;

struct Point  
{  
 public int x;  
 public int y;

public override string ToString() {  
 return "(" + x + "," + y + ")";  
 }  
}

class Test  
{  
 static void Main() {  
 Point point;  
 unsafe {  
 Point\* p = &point;  
 p->x = 10;  
 p->y = 20;  
 Console.WriteLine(p->ToString());  
 }  
 }  
}

wird der ->-Operator zum Zugriff auf Felder und zum Aufrufen einer Methode einer Struktur mithilfe eines Zeigers verwendet. Da die Operation P->I genau äquivalent zu (\*P).I ist, hätte die Main-Methode auch folgendermaßen geschrieben werden können:

class Test  
{  
 static void Main() {  
 Point point;  
 unsafe {  
 Point\* p = &point;  
 (\*p).x = 10;  
 (\*p).y = 20;  
 Console.WriteLine((\*p).ToString());  
 }  
 }  
}

### Zeigerelementzugriff

Ein pointer-element-access besteht aus einer primary-no-array-creation-expression, der ein Ausdruck in eckigen Klammern ("[" und "]") folgt.

pointer-element-access:  
primary-no-array-creation-expression [ expression ]

Bei einem Zeigerelementzugriff der Form P[E] muss P ein Ausdruck eines beliebigen Zeigertyps außer void\* sein, und E muss ein Ausdruck sein, der implizit in die Typen int, uint, long oder ulong konvertiert werden kann.

Ein Zeigerelementzugriff der Form P[E] wird exakt so ausgewertet wie \*(P + E). Eine Beschreibung des Zeigerdereferenzierungsoperators (\*) finden Sie in §18.5.1. Eine Beschreibung des Zeigeradditionsoperators (+) finden Sie in §18.5.6.

In dem Beispiel

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) p[i] = (char)i;  
 }  
 }  
}

wird ein Zeigerelementzugriff verwendet, um den Zeichenpuffer in einer for-Schleife zu initialisieren. Da die Operation P[E] genau äquivalent zu \*(P + E) ist, hätte das Beispiel auch folgendermaßen geschrieben werden können:

class Test  
{  
 static void Main() {  
 unsafe {  
 char\* p = stackalloc char[256];  
 for (int i = 0; i < 256; i++) \*(p + i) = (char)i;  
 }  
 }  
}

Der Zeigerelementzugriffsoperator überprüft nicht, ob der gültige Bereich überschritten wurde. Daher ist nicht definiert, wie sich der Zugriff auf ein Element außerhalb der Begrenzungen auswirkt. Dies gilt auch für die Sprachen C und C++.

### Der address-of-Operator

Eine addressof-expression-expression besteht aus einem kaufmännischen Und-Zeichen (&) gefolgt von einer unary-expression.

addressof-expression:  
& unary-expression

Mit einem vorgegebenen Ausdruck E, der den Typ T hat und als feste Variable klassifiziert ist (§18.3), berechnet das Konstrukt &E die Adresse der Variablen, die durch E angegeben wurde. Der Typ des Ergebnisses ist T\* und als Wert klassifiziert. Es tritt ein Kompilierungsfehler auf, wenn E nicht als Variable klassifiziert ist, wenn E als schreibgeschützte lokale Variable klassifiziert ist oder wenn E eine bewegliche Variable kennzeichnet. Im letzten Fall kann eine fixed-Anweisung (§18.6) verwendet werden, um die Variable vor der Ermittlung ihrer Adresse vorübergehend zu "fixieren". Wie schon in §7.6.4 beschrieben, wird außerhalb eines Instanzkonstruktors oder statischen Konstruktors für eine Struktur oder Klasse, die ein readonly-Feld definiert, dieses Feld als Wert und nicht als Variable angesehen. Daher kann seine Adresse nicht übernommen werden. Die Adresse einer Konstanten kann dementsprechend auch nicht übernommen werden.

Das Argument des &-Operators muss nicht definitiv zugewiesen sein, aber die der Operation folgende Variable, auf die der &-Operator angewendet wird, wird im Ausführungspfad, in dem die Operation auftritt, als definitiv zugewiesen angesehen. Bei der Programmierung muss sichergestellt werden, dass die richtige Initialisierung der Variablen an dieser Stelle tatsächlich stattfindet.

In dem Beispiel

using System;

class Test  
{  
 static void Main() {  
 int i;  
 unsafe {  
 int\* p = &i;  
 \*p = 123;  
 }  
 Console.WriteLine(i);  
 }  
}

i wird nach der zur Initialisierung von p verwendeten &i-Operation als definitiv zugewiesen angesehen. Die Zuweisung zu \*p initialisiert faktisch i, aber die Einbindung dieser Initialisierung ist bei der Programmierung sicherzustellen, denn es würde kein Laufzeitfehler auftreten, wenn diese Zuweisung entfernt werden würde.

Die Regeln für die definitive Zuweisung bei &-Operatoren sind insofern vorhanden, als redundante Initialisierungen von lokalen Variablen verhindert werden können. Viele externe APIs verwenden beispielsweise einen Zeiger auf eine Struktur, die von der API gefüllt wird. Aufrufe solcher APIs geben normalerweise die Adresse einer lokalen Strukturvariablen zurück, und ohne die Regel würden redundante Initialisierungen dieser Strukturvariablen erforderlich sein.

### Inkrementieren und Dekrementieren von Zeigern

In einem unsicheren Kontext können der ++-Operator und der ‑‑-Operator (§7.6.9 und §7.7.5) auf Zeigervariablen aller Typen außer void\* angewendet werden. Folglich sind für jeden Zeigertyp T\* folgende Operatoren implizit definiert:

T\* operator ++(T\* x);

T\* operator --(T\* x);

Die Operatoren führen zu den gleichen Ergebnissen wie x + 1 und x - 1 bzw. (§18.5.6). Für eine Zeigervariable des Typs T\* addiert der ++-Operator also sizeof(T) zu der in der Variablen enthaltenen Adresse, und der ‑‑-Operator subtrahiert sizeof(T) von der in der Variablen enthaltenen Adresse.

Wenn eine Inkrement- bzw. Dekrementoperation für Zeiger die Domänengrenzen des Zeigertyps überschreitet, wird das Ergebnis durch die Implementierung definiert, aber keine Ausnahme ausgelöst.

### Zeigerarithmetik

In einem unsicheren Kontext können der +-Operator und der --Operator (§7.8.4 und §7.8.5) auf Werte aller Zeigertypen außer void\* angewendet werden. Folglich sind für jeden Zeigertyp T\* folgende Operatoren implizit definiert:

T\* operator +(T\* x, int y);  
T\* operator +(T\* x, uint y);  
T\* operator +(T\* x, long y);  
T\* operator +(T\* x, ulong y);

T\* operator +(int x, T\* y);  
T\* operator +(uint x, T\* y);  
T\* operator +(long x, T\* y);  
T\* operator +(ulong x, T\* y);

T\* operator –(T\* x, int y);  
T\* operator –(T\* x, uint y);  
T\* operator –(T\* x, long y);  
T\* operator –(T\* x, ulong y);

long operator –(T\* x, T\* y);

Mit einem Ausdruck P vom Zeigertyp T\* und einem Ausdruck N vom Typ int, uint, long oder ulong berechnen die Ausdrücke P + N und N + P den Zeigerwert vom Typ T\*, der das Ergebnis der Addition von N \* sizeof(T) und der durch P angegebenen Adresse ist. Dementsprechend berechnet der Ausdruck P - N den Zeigerwert vom Typ T\*, der das Ergebnis der Subtraktion von N \* sizeof(T) von der durch P angegebenen Adresse ist.

Mit den Ausdrücken P und Q vom Zeigertyp T\* berechnet der Ausdruck P – Q die Differenz der durch P und Q angegebenen Adressen und dividiert diese Differenz dann durch sizeof(T). Der Typ des Ergebnisses ist immer long. Faktisch wird P - Q berechnet als ((long)(P) - (long)(Q)) / sizeof(T).

Beispiel:

using System;

class Test  
{

static void Main() {  
 unsafe {  
 int\* values = stackalloc int[20];  
 int\* p = &values[1];  
 int\* q = &values[15];  
 Console.WriteLine("p - q = {0}", p - q);  
 Console.WriteLine("q - p = {0}", q - p);  
 }  
 }  
}

erzeugt die Ausgabe:

p - q = -14  
q - p = 14

Wenn eine arithmetische Operation für Zeiger die Domänengrenzen des Zeigertyps überschreitet, wird das Ergebnis entsprechend der Definition in der Implementierung beschnitten, aber keine Ausnahme ausgelöst.

### Zeigervergleich

In einem unsicheren Kontext können die Operatoren ==, !=, <, >, <= und => (§7.10) auf die Werte aller Zeigertypen angewendet werden. Die Vergleichsoperatoren für Zeiger sind:

bool operator ==(void\* x, void\* y);

bool operator !=(void\* x, void\* y);

bool operator <(void\* x, void\* y);

bool operator >(void\* x, void\* y);

bool operator <=(void\* x, void\* y);

bool operator >=(void\* x, void\* y);

Da eine implizite Konvertierung jedes Zeigertyps in den Typ void\* vorhanden ist, können Operanden jedes Zeigertyps mithilfe dieser Operatoren verglichen werden. Die Vergleichsoperatoren vergleichen die von den beiden Operanden angegebenen Adressen miteinander, als ob es sich um Ganzzahlen ohne Vorzeichen handelte.

### Der sizeof-Operator

Der sizeof-Operator gibt die Anzahl der Bytes zurück, die von einer Variablen des angegebenen Typs belegt werden. Der als Operand für sizeof angegebene Typ muss ein unmanaged-type (§18.2) sein.

sizeof-expression:  
sizeof ( unmanaged-type )

Das Ergebnis des sizeof-Operators ist ein Wert vom Typ int. Für bestimmte vordefinierte Typen gibt der sizeof-Operator einen konstanten Wert zurück, wie die folgende Tabelle zeigt.

|  |  |
| --- | --- |
| **Ausdruck** | **Ergebnis** |
| sizeof(sbyte) | 1 |
| sizeof(byte) | 1 |
| sizeof(short) | 2 |
| sizeof(ushort) | 2 |
| sizeof(int) | 4 |
| sizeof(uint) | 4 |
| sizeof(long) | 8 |
| sizeof(ulong) | 8 |
| sizeof(char) | 2 |
| sizeof(float) | 4 |
| sizeof(double) | 8 |
| sizeof(bool) | 1 |

Für alle anderen Typen ist das Ergebnis des sizeof-Operators durch die Implementierung definiert und als Wert, nicht als Konstante, klassifiziert.

Die Reihenfolge, in der Member in eine Struktur gepackt werden, ist nicht definiert.

Zu Ausrichtungszwecken können nicht benannte Füllzeichen am Anfang, innerhalb und am Ende einer Struktur vorkommen. Der Inhalt der Bits, die als Füllzeichen verwendet werden, ist unbestimmt.

Wenn die Operation auf einen dem Strukturtyp entsprechenden Operanden angewendet wird, wird als Ergebnis die gesamte Anzahl der Bytes einschließlich aller Füllzeichen in einer Variablen dieses Typs zurückgegeben.

## Die fixed-Anweisung

In einem unsicheren Kontext ermöglicht die Produktion eines embedded-statement (§8) ein zusätzliches Konstrukt, die fixed-Anweisung. Sie wird zur "Fixierung" einer beweglichen Variablen verwendet, damit ihre Adresse für die Dauer der Anweisung konstant bleibt.

embedded-statement:  
...  
fixed-statement

fixed-statement:  
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement

fixed-pointer-declarators:  
fixed-pointer-declarator  
fixed-pointer-declarators , fixed-pointer-declarator

fixed-pointer-declarator:  
identifier = fixed-pointer-initializer

fixed-pointer-initializer:  
& variable-reference  
expression

Jeder fixed-pointer-declarator deklariert eine lokale Variable eines angegebenen pointer-type und initialisiert die lokale Variable mit der von dem entsprechenden fixed-pointer-initializer errechneten Adresse. Auf eine in einer fixed-Anweisung deklarierte Variable kann in beliebigen fixed-pointer-initializer auf der rechten Seite der Variablendeklaration und im embedded-statement der fixed-Anweisung zugegriffen werden. Eine durch eine fixed-Anweisung deklarierte lokale Variable gilt als schreibgeschützt. Wenn die eingebettete Anweisung versucht, eine Änderung dieser lokalen Variablen (durch Zuweisung bzw. die Operatoren ++ und ‑‑) oder die Weitergabe als ref-Parameter oder als out-Parameter vorzunehmen, tritt während der Kompilierung ein Fehler auf.

Als fixed-pointer-initializer kommen folgende Ausdrücke infrage:

* Das Token "&" gefolgt von einer variable-reference (§5.3.3) auf eine bewegliche Variable (§18.3) eines nicht verwalteten Typs T unter der Voraussetzung, dass der Typ T\* implizit in den Zeigertyp konvertierbar ist, der in der fixed-Anweisung angegeben wurde. In diesem Fall berechnet der Initialisierer die Adresse der angegebenen Variablen, und die Variable wird für die Dauer der fixed-Anweisung mit Sicherheit an einer festen Adresse gehalten.
* Ein Ausdruck vom array-type mit Elementen des nicht verwalteten Typs T unter der Voraussetzung, dass der Typ T\* implizit in den Zeigertyp konvertierbar ist, der in der fixed-Anweisung angegeben wurde. In diesem Fall berechnet der Initialisierer die Adresse des ersten Elements im Array, und das gesamte Array wird für die Dauer der fixed-Anweisung mit Sicherheit an einer festen Adresse gehalten. Das Verhalten der fixed-Anweisung hängt von der Implementierung ab, falls der Arrayausdruck gleich NULL ist oder das Array keine Elemente enthält.
* Ein Ausdruck des Typs string unter der Voraussetzung, dass der Typ char\* implizit in den Zeigertyp konvertierbar ist, der in der fixed-Anweisung angegeben wurde. In diesem Fall berechnet der Initialisierer die Adresse des ersten Zeichens der Zeichenfolge, und die gesamte Zeichenfolge wird für die Dauer der fixed-Anweisung mit Sicherheit an einer festen Adresse gehalten. Das Verhalten der fixed-Anweisung hängt von der Implementierung ab, wenn der Zeichenfolgenausdruck gleich NULL ist.
* Ein simple-name oder member-access, der auf einen Member eines Puffers mit fester Größe in einer beweglichen Variablen verweist, sofern der Typ des Members eines Puffers mit fester Größe implizit in den in der fixed-Anweisung angegebenen Zeigertyp konvertiert werden kann. In diesem Fall berechnet der Initialisierer einen Zeiger auf das erste Element des Puffers mit fester Größe (§18.7.2), und es ist gewährleistet, dass der Puffer mit fester Größe für die Dauer der fixed-Anweisung an einer festgelegten Adresse verbleibt.

Für jede von einem fixed-pointer-initializer berechnete Adresse stellt die fixed-Anweisung sicher, dass die Variable, auf die die Adresse verweist, für die Dauer der fixed-Anweisung nicht vom Garbage Collector verschoben oder verworfen werden kann. Wenn beispielsweise die von einem fixed-pointer-initializer berechnete Adresse auf ein Feld eines Objekts oder ein Element eines Array verweist, garantiert die fixed-Anweisung, dass die enthaltene Objektinstanz innerhalb der Gültigkeitsdauer der Anweisung nicht verschoben oder verworfen wird.

Beim Programmieren muss sichergestellt werden, dass diese von den fixed-Anweisungen erstellten Zeiger nicht über die Ausführung dieser Anweisungen hinaus gültig sind. Im Fall von Zeigern, die von fixed-Anweisungen erstellt und an externe APIs übergeben werden, muss bei der Programmierung beispielsweise sichergestellt werden, dass die APIs diese Zeiger nicht speichern.

Feste Objekte können zu einer Fragmentierung des Heaps führen (da sie nicht verschoben werden können). Daher sollten Objekte nur bei absoluter Notwendigkeit fixiert werden und dann auch nur für die kürzeste Dauer, die möglich ist.

In dem Beispiel

class Test  
{  
 static int x;  
 int y;

unsafe static void F(int\* p) {  
 \*p = 1;  
 }

static void Main() {  
 Test t = new Test();  
 int[] a = new int[10];  
 unsafe {  
 fixed (int\* p = &x) F(p);  
 fixed (int\* p = &t.y) F(p);  
 fixed (int\* p = &a[0]) F(p);  
 fixed (int\* p = a) F(p);  
 }  
 }  
}

zeigt verschiedene Anwendungen der fixed-Anweisung. Die erste Anweisung fixiert und ermittelt die Adresse eines statischen Feldes, die zweite Anweisung fixiert und ermittelt die Adresse eines Instanzfelds, und die dritte Anweisung fixiert und ermittelt die Adresse eines Arrayelements. In allen Fällen würde bei der Verwendung des regulären &-Operators ein Fehler auftreten, da alle Variablen als bewegliche Variablen klassifiziert sind.

Die vierte fixed-Anweisung in dem oben stehenden Beispiel erzeugt ähnliche Ergebnisse wie die dritte Anweisung.

In diesem Beispiel für die fixed-Anweisung wird string verwendet:

class Test  
{  
 static string name = "xx";

unsafe static void F(char\* p) {  
 for (int i = 0; p[i] != '\0'; ++i)  
 Console.WriteLine(p[i]);  
 }

static void Main() {  
 unsafe {  
 fixed (char\* p = name) F(p);  
 fixed (char\* p = "xx") F(p);  
 }  
 }  
}

Eindimensionale Arrays in einem unsicheren Kontext von Arrayelementen werden in aufsteigender Indexreihenfolge gespeichert, beginnend mit dem Index 0 und endend mit dem Index Length – 1. Die Arrayelemente mehrdimensionaler Arrays werden so gespeichert, dass die Indizes in folgender Reihenfolge erhöht werden: zuerst die Indizes der am weitesten rechts stehenden Dimension, dann die der nächsten links davon liegenden Dimension usw. Innerhalb einer fixed-Anweisung, die einen Zeiger p auf eine Arrayinstanz a ermittelt, stellen die Zeigerwerte von p bis p + a.Length - 1 die Adressen der Elemente des Arrays dar. Dementsprechend stellen die Variablen im Bereich von p[0] bis p[a.Length - 1] die tatsächlichen Arrayelemente dar. Durch die Art und Weise, in der Arrays gespeichert werden, ist es möglich, ein Array beliebiger Dimension so zu behandeln, als wäre es linear.

Beispiel:

using System;

class Test  
{  
 static void Main() {  
 int[,,] a = new int[2,3,4];  
 unsafe {  
 fixed (int\* p = a) {  
 for (int i = 0; i < a.Length; ++i) // treat as linear  
 p[i] = i;  
 }  
 }

for (int i = 0; i < 2; ++i)  
 for (int j = 0; j < 3; ++j) {  
 for (int k = 0; k < 4; ++k)  
 Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);  
 Console.WriteLine();  
 }  
 }  
}

erzeugt die Ausgabe:

[0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3  
[0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7  
[0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11  
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15  
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19  
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23

In dem Beispiel

class Test  
{  
 unsafe static void Fill(int\* p, int count, int value) {  
 for (; count != 0; count--) \*p++ = value;  
 }

static void Main() {  
 int[] a = new int[100];  
 unsafe {  
 fixed (int\* p = a) Fill(p, 100, -1);  
 }  
 }  
}

wird eine fixed-Anweisung verwendet, um ein Array zu fixieren, damit seine Adresse an eine Methode weitergegeben werden kann, die einen Zeiger akzeptiert.

In dem Beispiel

unsafe struct Font  
{  
 public int size;  
 public fixed char name[32];  
}

class Test  
{  
 unsafe static void PutString(string s, char\* buffer, int bufSize) {  
 int len = s.Length;  
 if (len > bufSize) len = bufSize;  
 for (int i = 0; i < len; i++) buffer[i] = s[i];  
 for (int i = len; i < bufSize; i++) buffer[i] = (char)0;  
 }

Font f;

unsafe static void Main()  
 {  
 Test test = new Test();  
 test.f.size = 10;  
 fixed (char\* p = test.f.name) {  
 PutString("Times New Roman", p, 32);  
 }  
 }  
}

wird eine fixed-Anweisung verwendet, um einen Puffer fester Größe einer Struktur zu fixieren, damit seine Adresse als Zeiger verwendet werden kann.

Ein char\*-Wert, der durch die Fixierung einer Zeichenfolgeninstanz erzeugt wurde, zeigt immer auf eine mit null endende Zeichenfolge. Innerhalb einer fixed-Anweisung, die einen Zeiger p zu einer Zeichenfolgeninstanz s erhält, stellen die Zeigerwerte im Bereich von p bis p + s.Length - 1 die Adressen der Zeichen in der Zeichenfolge dar, und der Zeigerwert p + s.Length zeigt immer auf ein Nullzeichen (das Zeichen mit dem Wert '\0'').

Die Änderung von Objekten eines verwalteten Typs mithilfe von festen Zeigern kann nicht definierte Auswirkungen haben. Da Zeichenfolgen nicht verändert werden dürfen, muss bei der Programmierung beispielsweise sichergestellt werden, dass die Zeichen, auf die ein Zeiger auf eine feste Zeichenfolge verweist, nicht verändert werden.

Die automatische Nullterminierung von Zeichenfolgen ist insbesondere beim Aufruf externer APIs hilfreich, die Zeichenfolgen im „C-Stil“ erwarten. Dennoch ist es zulässig, dass Zeichenfolgeninstanzen Nullzeichen enthalten. Wenn ein solches Nullzeichen vorhanden ist, wird die Zeichenfolge beschnitten, sobald sie als nullterminiertes char\* behandelt wird.

## Puffer mit fester Größe

Mithilfe von Puffern mit fester Größe werden C-Inline-Arrays als Member von Strukturen deklariert. Sie sind daher insbesondere bei der Interaktion mit nicht verwalteten APIs hilfreich.

### Deklarationen von Puffern mit fester Größe

Ein Puffer mit fester Größe ist ein Member, der den Speicher für einen Puffer mit fester Größe für Variablen eines bestimmten Typs darstellt. Eine Deklaration von Puffern mit fester Größe deklariert einen oder mehrere Puffer mit fester Größe für einen bestimmten Elementtyp. Puffer mit fester Größe sind nur in Strukturdeklarationen zulässig und können nur in unsicheren Kontexten (§18.1) vorhanden sein.

struct-member-declaration:  
…  
fixed-size-buffer-declaration

fixed-size-buffer-declaration:  
attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type  
 fixed-size-buffer-declarators ;

fixed-size-buffer-modifiers:  
fixed-size-buffer-modifier  
fixed-size-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifier:  
new  
public  
protected  
internal  
private  
unsafe

buffer-element-type:  
type

fixed-size-buffer-declarators:  
fixed-size-buffer-declarator  
fixed-size-buffer-declarator , fixed-size-buffer-declarators

fixed-size-buffer-declarator:  
identifier [ constant-expression ]

Eine Deklaration von Puffern mit fester Größe kann eine Gruppe von Attributen (§17), einen new-Modifizierer (§10.2.2), eine gültige Kombination der vier Zugriffsmodifizierer (§10.2.3) und einen unsafe-Modifizierer (§18.1) enthalten. Die Attribute und Modifizierer treffen für alle Member zu, die durch die Deklaration von Puffern mit fester Größe deklariert wurden. Wenn derselbe Modifizierer mehrmals in einer Deklaration von Puffern mit fester Größe aufgeführt ist, wird ein Fehler verursacht.

Eine Deklaration von Puffern mit fester Größe darf keinen static-Modifizierer enthalten.

Der Pufferelementtyp einer Deklaration von Puffern mit fester Größe gibt den Elementtyp des bzw. der Puffer an, die durch die Deklaration eingeführt werden. Der Pufferelementtyp muss einer der vordefinierten Typen sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double oder bool sein.

Auf den Pufferelementtyp folgt eine Liste von Deklaratoren für Puffer mit fester Größe, von denen jeder einen neuen Member einführt. Ein Deklarator für einen Puffer mit fester Größe besteht aus einem Bezeichner, der den Member benennt, auf den ein konstanter Ausdruck folgt, der in die Token [ und ] eingeschlossen ist. Der konstante Ausdruck bezeichnet die Anzahl der Elemente im Member, der durch den betreffenden Deklarator für einen Puffer mit fester Größe eingeführt wird. Der Typ des konstanten Ausdrucks muss implizit in den Typ int konvertierbar sein, und bei dem Wert muss es sich um eine positive ganze Zahl ungleich 0 (null) handeln.

Die Elemente des Puffers mit fester Größe sind im Speicher in jedem Fall in der entsprechenden Reihenfolge angeordnet.

Eine Deklaration von Puffern mit fester Größe, die mehrere Puffer mit fester Größe deklariert, entspricht mehreren Deklarationen eines einzigen Puffers mit fester Größe mit denselben Attributen und Elementtypen. Beispiel:

unsafe struct A  
{  
 public fixed int x[5], y[10], z[100];  
}

äquivalent zu

unsafe struct A  
{  
 public fixed int x[5];  
 public fixed int y[10];  
 public fixed int z[100];  
}

### Puffer mit fester Größe in Ausdrücken

Die Membersuche (§7.3) nach einem Member eines Puffers mit fester Größe verläuft genau so wie die Membersuche nach einem Feld.

Auf einen Puffer mit fester Größe kann in einem Ausdruck mit einem simple-name (§7.5.2) oder mit einem member-access (§7.5.4) verwiesen werden.

Wenn mit einem einfachen Namen auf einen Puffer mit fester Größe verwiesen wird, entspricht die Wirkung der eines Memberzugriffs der Form this.I, wobei I für den Member des Puffers mit fester Größe steht.

Wenn es sich in einem Memberzugriff der Form E.I bei E um einen Strukturtyp handelt und eine Membersuche von I in diesem Strukturtyp einen Member mit fester Größe angibt, wird E.I wie folgt ausgewertet und klassifiziert:

* Wenn der Ausdruck E.I nicht in einem unsicheren Kontext vorhanden ist, tritt ein Kompilierungsfehler auf.
* Wenn E als Wert klassifiziert wird, tritt ein Kompilierungsfehler auf.
* Wenn es sich bei E jedoch um eine bewegliche Variable (§18.3) handelt und der Ausdruck E.I kein fixed-pointer-initializer (§18.6) ist, tritt ein Kompilierungsfehler auf.
* Andernfalls verweist E auf eine feste Variable, und das Ergebnis des Ausdrucks ist ein Zeiger auf das erste Element des Members des Puffers mit fester Größe I in E. Das Ergebnis ist vom Typ S\*, wobei S der Elementtyp von I ist und als Wert klassifiziert wird.

Auf die nachfolgenden Elemente des Puffers mit fester Größe kann mit Zeigeroperationen vom ersten Element aus zugegriffen werden. Im Unterschied zum Zugriff auf Arrays handelt es sich beim Zugriff auf die Elemente eines Puffers mit fester Größe um einen unsicheren Vorgang, und es erfolgt keine Bereichsüberprüfung.

Im folgenden Beispiel wird eine Struktur erstellt und mit einem Member eines Puffers mit fester Größe verwendet.

unsafe struct Font  
{  
 public int size;  
 public fixed char name[32];  
}

class Test  
{  
 unsafe static void PutString(string s, char\* buffer, int bufSize) {  
 int len = s.Length;  
 if (len > bufSize) len = bufSize;  
 for (int i = 0; i < len; i++) buffer[i] = s[i];  
 for (int i = len; i < bufSize; i++) buffer[i] = (char)0;  
 }

unsafe static void Main()  
 {  
 Font f;  
 f.size = 10;  
 PutString("Times New Roman", f.name, 32);  
 }  
}

### Überprüfung des definitiven Zuweisungszustands

Puffer mit fester Größe unterliegen nicht der Überprüfung des definitiven Zuweisungszustands (§5.3), und Member von Puffern mit fester Größe werden bei der Überprüfung des definitiven Zuweisungszustands von Strukturtypvariablen ignoriert.

Wenn die äußerste enthaltende Strukturvariable eines Members eines Puffers mit fester Größe eine statische Variable, eine Instanzvariable einer Klasseninstanz oder ein Arrayelement ist, werden die Elemente des Puffers mit fester Größe automatisch mit ihren Standardwerten (§5.2) initialisiert. In allen anderen Fällen ist der anfängliche Inhalt eines Puffers mit fester Größe nicht definiert.

## Stapelreservierung

In einem unsicheren Kontext kann eine Deklaration einer lokalen Variablen (§8.5.1) eine Initialisierung einer Stapelreservierung enthalten, die Speicher aus der Aufrufliste belegt.

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

Der unmanaged-type kennzeichnet den Typ der Elemente, die im neu reservierten Speicherort gespeichert werden, und die expression gibt die Anzahl dieser Elemente an. Zusammengenommen legen diese Angaben die erforderliche Speicherbelegungsgröße fest. Da die Größe einer Stapelreservierung nicht negativ sein kann, tritt ein Kompilierungsfehler auf, wenn die Anzahl der Elemente als constant-expression, die einen negativen Wert ergibt, angegeben wird.

Bei der Initialisierung einer Stapelreservierung der Form stackalloc T[E] muss T ein nicht verwalteter Typ (§18.2) und E ein Ausdruck vom Typ int sein. Dieses Konstrukt reserviert E \* sizeof(T) Bytes der Aufrufliste und erzeugt einen Zeiger vom Typ T\* auf den neu reservierten Block. Wenn E ein negativer Wert ist, ist das Verhalten nicht definiert. Wenn E gleich 0 ist, wurde keine Reservierung vorgenommen, und es hängt von der Implementierung ab, welcher Zeiger zurückgegeben wird. Steht für die Belegung eines Blocks der angegebenen Größe nicht genügend Speicher zur Verfügung, wird eine System.StackOverflowException ausgelöst.

Der Inhalt des neu belegten Speichers ist nicht definiert.

Initialisierer von Stapelreservierungen sind in catch-Blöcken oder finally-Blöcken (§8.10) nicht zulässig.

Es gibt keine Möglichkeit, mithilfe von stackalloc explizit freien Speicher zu belegen. Alle mithilfe der Stapelreservierung während der Ausführung eines Funktionsmembers erstellten Speicherblöcke werden automatisch verworfen, sobald der Funktionsmember ausgeführt ist. Dies entspricht der alloca-Funktion, einer Erweiterung, die oft in C- und C++-Implementierungen zu finden ist.

In dem Beispiel

using System;

class Test  
{  
 static string IntToString(int value) {  
 int n = value >= 0? value: -value;  
 unsafe {  
 char\* buffer = stackalloc char[16];  
 char\* p = buffer + 16;  
 do {  
 \*--p = (char)(n % 10 + '0');  
 n /= 10;  
 } while (n != 0);  
 if (value < 0) \*--p = '-';  
 return new string(p, 0, (int)(buffer + 16 - p));  
 }  
 }

static void Main() {  
 Console.WriteLine(IntToString(12345));  
 Console.WriteLine(IntToString(-999));  
 }  
}

wird ein stackalloc-Initialisierer in der IntToString-Methode verwendet, um einen Puffer von 16 Zeichen auf dem Stapel zu reservieren. Beim Rücksprung von der Methode wird der Puffer automatisch wieder freigegeben.

## Dynamische Speicherbelegung

Abgesehen vom stackalloc-Operator stellt C# keine vordefinierten Konstrukte für die Verwaltung von Speicher bereit, der nicht in Garbage Collection-Prozesse eingebunden ist. Solche Dienste werden normalerweise von unterstützenden Klassenbibliotheken bereitgestellt oder direkt aus dem zugrunde liegenden Betriebssystem importiert. Die im Folgenden dargestellte Memory-Klasse zeigt beispielsweise, wie von C# auf die Heapfunktionen eines zugrunde liegenden Betriebssystems zugegriffen werden kann:

using System;  
using System.Runtime.InteropServices;

public unsafe class Memory  
{  
 // Handle for the process heap. This handle is used in all calls to the  
 // HeapXXX APIs in the methods below.

static int ph = GetProcessHeap();

// Private instance constructor to prevent instantiation.

private Memory() {}

// Allocates a memory block of the given size. The allocated memory is  
 // automatically initialized to zero.

public static void\* Alloc(int size) {  
 void\* result = HeapAlloc(ph, HEAP\_ZERO\_MEMORY, size);  
 if (result == null) throw new OutOfMemoryException();  
 return result;  
 }

// Copies count bytes from src to dst. The source and destination  
 // blocks are permitted to overlap.

public static void Copy(void\* src, void\* dst, int count) {  
 byte\* ps = (byte\*)src;  
 byte\* pd = (byte\*)dst;  
 if (ps > pd) {  
 for (; count != 0; count--) \*pd++ = \*ps++;  
 }  
 else if (ps < pd) {  
 for (ps += count, pd += count; count != 0; count--) \*--pd = \*--ps;  
 }  
 }

// Frees a memory block.

public static void Free(void\* block) {  
 if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();  
 }

// Re-allocates a memory block. If the reallocation request is for a  
 // larger size, the additional region of memory is automatically  
 // initialized to zero.

public static void\* ReAlloc(void\* block, int size) {  
 void\* result = HeapReAlloc(ph, HEAP\_ZERO\_MEMORY, block, size);  
 if (result == null) throw new OutOfMemoryException();  
 return result;  
 }

// Returns the size of a memory block.

public static int SizeOf(void\* block) {  
 int result = HeapSize(ph, 0, block);  
 if (result == -1) throw new InvalidOperationException();  
 return result;  
 }

// Heap API flags

const int HEAP\_ZERO\_MEMORY = 0x00000008;

// Heap API functions

[DllImport("kernel32")]  
 static extern int GetProcessHeap();

[DllImport("kernel32")]  
 static extern void\* HeapAlloc(int hHeap, int flags, int size);

[DllImport("kernel32")]  
 static extern bool HeapFree(int hHeap, int flags, void\* block);

[DllImport("kernel32")]  
 static extern void\* HeapReAlloc(int hHeap, int flags,  
 void\* block, int size);

[DllImport("kernel32")]  
 static extern int HeapSize(int hHeap, int flags, void\* block);  
}

Das folgende Beispiel verwendet die Memory-Klasse:

class Test  
{  
 static void Main() {  
 unsafe {  
 byte\* buffer = (byte\*)Memory.Alloc(256);  
 try {  
 for (int i = 0; i < 256; i++) buffer[i] = (byte)i;  
 byte[] array = new byte[256];  
 fixed (byte\* p = array) Memory.Copy(buffer, p, 256);   
 }  
 finally {  
 Memory.Free(buffer);  
 }  
 for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);  
 }  
 }  
}

Das Beispiel belegt 256 Bytes Speicher mithilfe von Memory.Alloc und initialisiert den Speicherblock mit aufeinander folgenden Werten von 0 bis 255. Dann wird ein Bytearray mit 256 Elementen belegt und Memory.Copy verwendet, um den Inhalt des Speicherblocks in das Bytearray zu kopieren. Schließlich wird der Speicherblock mithilfe von Memory.Free wieder freigegeben und der Inhalt des byte-Arrays an der Konsole ausgegeben.

1. Dokumentationskommentare

C# bietet Programmierern einen Mechanismus, mit dem sie ihren Code durch eine spezielle Kommentarsyntax mit XML-Text dokumentieren können. In Quellcodedateien können Kommentare mit einer bestimmten Form ein Tool anweisen, aus diesen Kommentaren und den vorangehenden Quellcodeelementen XML-TextXML zu generieren. Kommentare mit dieser Syntax werden Dokumentationskommentare genannt. Sie müssen unmittelbar vor einem benutzerdefinierten Typ (z. B. Klasse, Delegat oder Schnittstelle) oder einem Member (z. B. Feld, Ereignis, Eigenschaft oder Methode) stehen. Das Tool zur XML-Generierung wird als Dokumentations-Generator bezeichnet. (Dabei kann es sich um den C#-Compiler selbst handeln, dies ist jedoch nicht zwingend.) Die vom Dokumentations-Generator erzeugte Ausgabe wird als Dokumentationsdatei bezeichnet. Eine Dokumentationsdatei wird als Eingabe in eine Dokumentationsanzeige verwendet, ein Tool, das eine visuelle Aufbereitung von Typinformationen und der zugehörigen Dokumentation liefern soll.

In dieser Spezifikation wird vorgeschlagen, eine Gruppe von Tags für Dokumentationskommentare zu verwenden. Die Verwendung dieser Tags ist jedoch nicht obligatorisch, und optional können auch andere Tags verwendet werden, solange regelkonforme XML-Daten entstehen.

* 1. Einführung

Kommentare mit einer bestimmten Form können ein Tool anweisen, aus diesen Kommentaren und den vorangehenden Quellcodeelementen XML-TextXML zu generieren. Diese Kommentare können einzeilige Kommentare sein, die von drei Schrägstrichen (///) angeführt werden, oder getrennte Kommentare, die mit einem Schrägstrich und zwei Sternchen (/\*\*) beginnen. Sie müssen unmittelbar vor einem benutzerdefinierten Typ (z. B. Klasse, Delegat oder Schnittstelle) oder einem Member (z. B. Feld, Ereignis, Eigenschaft oder Methode) stehen, den sie kommentieren. Attributabschnitte (§17.2) gelten als Teil einer Deklaration, daher müssen Dokumentationskommentare den Attributen vorangestellt werden, die auf Typen oder Member angewendet werden.

**Syntax:**

single-line-doc-comment:  
/// input-charactersopt

delimited-doc-comment:  
/\*\* delimited-comment-textopt \*/

Wenn in einem single-line-doc-comment neben dem aktuellen single-line-doc-comment der einzelnen single-line-doc-commentss nach den ///-Zeichen ein whitespace-Zeichen folgt, dann wird dieses whitespace-Zeichnen nicht in die XML-Ausgabe eingefügt.

Wenn es sich bei dem ersten Zeichen in der zweiten Zeile in einem delimited-doc-comment, das kein Leerraumzeichen ist, um ein Sternchen handelt und dasselbe Muster optionaler Leerraumzeichen und Sternchen zu Beginn jeder Zeile innerhalb des delimited-doc-comments wiederholt wird, werden die Zeichen dieses Wiederholungsmusters nicht in die XML-Ausgabe aufgenommen. Das Muster kann sowohl hinter als auch vor dem Sternchen Leerraumzeichen enthalten.

**Beispiel:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>  
///  
public class Point   
{  
 /// <summary>method <c>draw</c> renders the point.</summary>  
 void draw() {…}  
}

Der Text innerhalb der Dokumentationskommentare muss den XML-Regeln (http://www.w3.org/TR/REC-xml) entsprechen. Falls das XML-Format nicht regelkonform ist, wird eine Warnung ausgegeben und ein Kommentar in die Dokumentationsdatei eingefügt, der auf diesen Fehler hinweist.

Obwohl Entwickler ihre eigenen Tags erstellen können, wurde ein empfohlenes Tagset in §A.2 definiert. Einige der empfohlenen Tags haben eine besondere Bedeutung:

* Das <param>-Tag wird zur Beschreibung von Parametern verwendet. Bei Verwendung dieses Tags muss der Dokumentationsgenerator überprüfen, ob der angegebene Parameter vorhanden ist und ob alle Parameter in den Dokumentationskommentaren beschrieben werden. Falls diese Überprüfung fehlschlägt, gibt der Dokumentationsgenerator eine Warnung aus.
* Das cref-Attribut kann an jedes Tag angefügt werden, um einen Verweis auf ein Codeelement bereitzustellen. Der Dokumentationsgenerator muss überprüfen, ob dieses Codeelement vorhanden ist. Falls diese Überprüfung fehlschlägt, gibt der Dokumentationsgenerator eine Warnung aus. Bei der Suche nach einem in einem cref-Attribut beschriebenen Namen muss der Dokumentations-Generator die Sichtbarkeit des Namespaces entsprechend der im Quellcode festgelegten using-Anweisungen beachten. Bei generischen Codeelementen kann die normale generische Syntax (also "List<T>") nicht verwendet werden, da diese ungültigen XML-Code erzeugt. Sie können geschweifte Klammern anstelle von eckigen Klammern verwenden (also "List{T}") oder die XML-Escapesyntax anwenden (also "List&lt;T&gt;").
* Das <summary>-Tag wird von der Dokumentationsanzeige verwendet, um zusätzliche Informationen über einen Typ oder einen Member anzuzeigen.
* Das <include>-Tag enthält Informationen aus einer externen XML-Datei.

Beachten Sie, dass die Dokumentationsdatei keine vollständigen Informationen zu Typen und Membern liefert (sie umfasst z. B. keine Typinformationen). Um vollständige Informationen zu einem Typ oder Member zu erhalten, muss die Dokumentationsdatei in Verbindung mit der Reflektion des aktuellen Typs oder Members verwendet werden.

* 1. Empfohlene Tags

Der Dokumentationsgenerator muss alle gemäß den XML-Regeln zulässigen Tags akzeptieren und verarbeiten. Die folgenden Tags stellen in Benutzerdokumentationen häufig verwendete Funktionen bereit: (Selbstverständlich werden auch andere Tags unterstützt.)

|  |  |  |
| --- | --- | --- |
| **Tag** | **Abschnitt** | **Zweck** |
| <c> | A.2.1 | Anzeigen von Text in einer codeähnlichen Schriftart |
| <code> | A.2.2 | Definieren einer oder mehrerer Quellcode- oder Programmausgabezeilen |
| <example> | A.2.3 | Angeben eines Beispiels |
| <exception> | A.2.4 | Kennzeichnen der von einer Methode auslösbaren Ausnahmen |
| <include> | A.2.5 | Einfügen von XML aus einer externen Datei |
| <list> | A.2.6 | Erstellen einer Liste oder Tabelle |
| <para> | A.2.7 | Ermöglichen der Textstrukturierung |
| <param> | A.2.8 | Beschreiben eines Parameters für eine Methode oder einen Konstruktor |
| <paramref> | A.2.9 | Kennzeichnen eines Wortes als Parametername |
| <permission> | A.2.10 | Dokumentieren des Sicherheitszugriffs eines Members |
| <remark> | A.2.11 | Beschreiben zusätzlicher Informationen über einen Typ |
| <returns> | A.2.12 | Beschreiben des Rückgabewertes einer Methode |
| <see> | A.2.13 | Angeben eines Links |
| <seealso> | A.2.14 | Erstellen eines Siehe auch-Eintrags |
| <summary> | A.2.15 | Beschreiben eines Typs oder eines Members eines Typs |
| <value> | A.2.16 | Beschreiben einer Eigenschaft |
| <typeparam> |  | Beschreiben eines generische Typparameters |
| <typeparamref> |  | Kennzeichnen eines Wortes als Typparametername |

* + 1. <c>

Dieses Tag bietet einen Mechanismus, durch den angegeben wird, dass ein Textfragment innerhalb einer Beschreibung in einer bestimmten Schriftart formatiert werden soll, z. B. der eines Codeblocks. Für tatsächliche Codezeilen sollten Sie <code> verwenden (§A.2.2).

**Syntax:**

<c>*text*</c>

**Beispiel:**

/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>

public class Point   
{  
 // ...  
}

* + 1. <code>

Dieses Tag wird verwendet, um eine oder mehrere Zeilen des Quellcodes oder der Programmausgabe in einer bestimmten Schriftart auszugeben. Für kleinere Codefragmente in erläuterndem Text sollte <c> (§A.2.1) verwendet werden.

**Syntax:**

<code>*source code or program output*</code>

**Beispiel:**

/// <summary>This method changes the point's location by  
/// the given x- and y-offsets.  
/// <example>For example:  
/// <code>  
/// Point p = new Point(3,5);  
/// p.Translate(-1,3);  
/// </code>  
/// results in <c>p</c>'s having the value (2,8).  
/// </example>  
/// </summary>

public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
}

* + 1. <example>

Dieses Tag lässt in einem Kommentar Beispielcode zu, um anzugeben, wie eine Methode oder ein anderer Bibliotheksmember verwendet wird. In der Regel würde dies auch die Verwendung des <code>-Tags (§A.2.2) einschließen.

**Syntax:**

<example>*description*</example>

**Beispiel:**

Ein Beispiel finden Sie unter <code> (§A.2.2).

* + 1. <exception>

Dieses Tag bietet eine Möglichkeit, die Ausnahmen zu dokumentieren, die von einer Methode ausgelöst werden können.

**Syntax:**

<exception cref="*member*">*description*</exception>

wobei gilt:

cref="*member*"

Der Name eines Members. Der Dokumentationsgenerator überprüft, ob der angegebene Member vorhanden ist, und übersetzt member in den erforderlichen Elementnamen in der Dokumentationsdatei.

*description*

Entspricht einer Beschreibung der Umstände, unter denen die Ausnahme ausgelöst wird.

**Beispiel:**

public class DataBaseOperations  
{  
 /// <exception cref="MasterFileFormatCorruptException"></exception>  
 /// <exception cref="MasterFileLockedOpenException"></exception>  
 public static void ReadRecord(int flag) {  
 if (flag == 1)  
 throw new MasterFileFormatCorruptException();  
 else if (flag == 2)  
 throw new MasterFileLockedOpenException();  
 // …  
 }   
}

* + 1. <include>

Dieses Tag ermöglicht die Übernahme von Informationen aus einem externen XML-Dokument in die Quellcodedatei. Die externe Datei muss ein regelkonformes XML-Dokument sein, auf das ein XPath-Ausdruck angewendet wird, um den in das Dokument einzufügenden XML-Code festzulegen. Das <include>-Tag wird dann durch den ausgewählten XML-Code aus dem externen Dokument ersetzt.

**Syntax:**

<include file="filename" path="xpath" />

wobei gilt:

file="filename"

Der Dateiname einer externen XML-Datei. Der Dateiname wird in Bezug auf die Datei mit dem include-Tag interpretiert.

path="xpath"

Ein XPath-Ausdruck, der einen Teil des XML-Codes aus der externen XML-Datei auswählt.

**Beispiel:**

Wenn der Quellcode folgende Deklaration enthält:

/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/\*' />  
public class IntList { … }

und die externe Datei "docs.xml" folgenden Inhalt aufweist:

<?xml version="1.0"?>  
<extradoc>  
 <class name="IntList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
 <class name="StringList">  
 <summary>  
 Contains a list of integers.  
 </summary>  
 </class>  
</extradoc>

dann wird dieselbe Dokumentation ausgegeben, als ob der Quellcode folgenden Inhalt aufweisen würde:

/// <summary>  
/// Contains a list of integers.  
/// </summary>  
public class IntList { … }

* + 1. <list>

Mit diesem Tag wird eine Liste oder Tabelle von Elementen erstellt. Darin ist ggf. ein <listheader>-Block enthalten, der die Überschriftenzeile einer Tabelle oder einer Definitionsliste definiert. (Bei der Definition einer Tabelle muss nur ein Eintrag für *term* in der Überschrift angegeben werden.)

Jedes Element der Liste wird mit einem <item>-Block angegeben. Beim Erstellen einer Definitionsliste müssen sowohl *term* als auch *description* angegeben werden. Für eine Tabelle, Aufzählung oder nummerierte Liste muss jedoch nur *description* angegeben werden.

**Syntax:**

<list type="bullet" | "number" | "table">  
 <listheader>  
 <term>*term*</term>  
 <description>*description*</description>  
 </listheader>  
 <item>  
 <term>*term*</term>  
 <description>*description*</description>  
 </item>  
 …  
 <item>  
 <term>*term*</term>  
 <description>*description*</description>  
 </item>  
</list>

wobei gilt:

*term*

Der zu definierende Begriff, dessen Definition in *description* festgelegt ist.

*description*

Ein Element in einer Aufzählung oder nummerierten Liste oder die Definition eines *term*.

**Beispiel:**

public class MyClass  
{  
 /// <summary>Here is an example of a bulleted list:  
 /// <list type="bullet">  
 /// <item>  
 /// <description>Item 1.</description>  
 /// </item>  
 /// <item>  
 /// <description>Item 2.</description>  
 /// </item>  
 /// </list>  
 /// </summary>  
 public static void Main () {  
 // ...  
 }  
}

* + 1. <para>

Dieses Tag wird innerhalb anderer Tags, z. B. <summary> (§A.2.11) oder <returns> (§A.2.12), verwendet und ermöglicht die Strukturierung von Text.

**Syntax:**

<para>*content*</para>

wobei gilt:

*content*

Der Text des Absatzes.

**Beispiel:**

/// <summary>This is the entry point of the Point class testing program.  
/// <para>This program tests each method and operator, and  
/// is intended to be run after any non-trvial maintenance has  
/// been performed on the Point class.</para></summary>  
public static void Main() {  
 // ...  
}

* + 1. <param>

Dieses Tag dient zur Beschreibung eines Parameters für eine Methode, einen Konstruktor oder einen Indexer.

**Syntax:**

<param name="*name*">*description*</param>

wobei gilt:

*name*

Der Name des Parameters.

*description*

Eine Beschreibung des Parameters.

**Beispiel:**

/// <summary>This method changes the point's location to  
/// the given coordinates.</summary>  
/// <param name="xor">the new x-coordinate.</param>  
/// <param name="yor">the new y-coordinate.</param>  
public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

* + 1. <paramref>

Mit diesem Tag wird angegeben, dass ein Wort als Parameter verwendet wird. Die Dokumentationsdatei kann verarbeitet werden, um den Parameter in besonderer Weise zu formatieren.

**Syntax:**

<paramref name="*name*"/>

wobei gilt:

*name*

Der Name des Parameters.

**Beispiel:**

/// <summary>This constructor initializes the new Point to  
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
/// <param name="xor">the new Point's x-coordinate.</param>  
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

* + 1. <permission>

Dieses Tag ermöglicht es, den Sicherheitszugriff eines Members zu dokumentieren.

**Syntax:**

<permission cref="*member*">*description*</permission>

wobei gilt:

cref="*member*"

Der Name eines Members. Der Dokumentations-Generator überprüft, ob das angegebene Codeelement vorhanden ist, und übersetzt member in den in der Dokumentationsdatei enthaltenen kanonischen Elementnamen.

*description*

Eine Beschreibung des Zugriffs auf den Member.

**Beispiel:**

/// <permission cref="System.Security.PermissionSet">Everyone can  
/// access this method.</permission>

public static void Test() {  
 // ...  
}

* + 1. <remark>

Mit diesem Tag werden zusätzliche Informationen zu einem Typ angegeben. (Verwenden Sie <summary> (§A.2.15), um den Typ selbst und die Member eines Typs zu beschreiben.)

**Syntax:**

<remark>*description*</remark>

wobei gilt:

*description*

Der Text der Bemerkung.

**Beispiel:**

/// <summary>Class <c>Point</c> models a point in a   
/// two-dimensional plane.</summary>  
/// <remark>Uses polar coordinates</remark>  
public class Point   
{  
 // ...  
}

* + 1. <returns>

Mit diesem Tag wird der Rückgabewert einer Methode beschrieben.

**Syntax:**

<returns>*description*</returns>

wobei gilt:

*description*

Eine Beschreibung des Rückgabewertes.

**Beispiel:**

/// <summary>Report a point's location as a string.</summary>  
/// <returns>A string representing a point's location, in the form (x,y),  
/// without any leading, trailing, or embedded whitespace.</returns>  
public override string ToString() {  
 return "(" + X + "," + Y + ")";  
}

* + 1. <see>

Mithilfe dieses Tags kann ein Link in einen Text eingefügt werden. Verwenden Sie <seealso> (§A.2.14), um Text anzugeben, der innerhalb eines Siehe auch-Abschnitts angezeigt werden soll.

**Syntax:**

<see cref="*member*"/>

wobei gilt:

cref="*member*"

Der Name eines Members. Der Dokumentations-Generator überprüft, ob das angegebene Codeelement vorhanden ist, und ändert member in den in der generierten Dokumentationsdatei enthaltenen Elementnamen.

**Beispiel:**

/// <summary>This method changes the point's location to  
/// the given coordinates.</summary>  
/// <see cref="Translate"/>  
public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
}

/// <summary>This method changes the point's location by  
/// the given x- and y-offsets.  
/// </summary>  
/// <see cref="Move"/>  
public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
}

* + 1. <seealso>

Mithilfe dieses Tags kann ein Eintrag für den Siehe auch-Abschnitt erstellt werden. Verwenden Sie <see> (§A.2.13), um einen Link aus dem Text anzugeben.

**Syntax:**

<seealso cref="*member*"/>

wobei gilt:

cref="*member*"

Der Name eines Members. Der Dokumentations-Generator überprüft, ob das angegebene Codeelement vorhanden ist, und ändert member in den in der generierten Dokumentationsdatei enthaltenen Elementnamen.

**Beispiel:**

/// <summary>This method determines whether two Points have the same  
/// location.</summary>  
/// <seealso cref="operator=="/>  
/// <seealso cref="operator!="/>  
public override bool Equals(object o) {  
 // ...  
}

* + 1. <summary>

Dieses Tag kann verwendet werden, um einen Typ oder einen Member eines Typs zu beschreiben. Verwenden Sie <remark> (§A.2.11), um den Typ selbst zu beschreiben.

**Syntax:**

<summary>*description*</summary>

wobei gilt:

*description*

Eine Zusammenfassung des Typs oder Members.

**Beispiel:**

/// <summary>This constructor initializes the new Point to (0,0).</summary>  
public Point() : this(0,0) {  
}

* + 1. <value>

Mit diesem Tag kann eine Eigenschaft beschrieben werden.

**Syntax:**

<value>*property* *description*</value>

wobei gilt:

*property description*

Eine Beschreibung für die Eigenschaft.

**Beispiel:**

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
public int X  
{  
 get { return x; }  
 set { x = value; }  
}

* + 1. <typeparam>

Dieses Tag dient zur Beschreibung eines generischen Typparameters für eine Klasse, eine Struktur, eine Schnittstelle, einen Delegaten oder eine Methode.

**Syntax:**

<typeparam name="*name*">*description*</typeparam>

wobei gilt:

*name*

Der Name des Typparameters.

*description*

Eine Beschreibung des Typparameters.

**Beispiel:**

/// <summary>A generic list class.</summary>  
/// <typeparam name="T">The type stored by the list.</typeparam>  
public class MyList<T> {  
 ...  
}

* + 1. <typeparamref>

Mit diesem Tag wird angegeben, dass ein Wort als Typparameter verwendet wird. Die Dokumentationsdatei kann verarbeitet werden, um den Typparameter in besonderer Weise zu formatieren.

**Syntax:**

<typeparamref name="*name*"/>

wobei gilt:

*name*

Der Name des Typparameters.

**Beispiel:**

/// <summary>This method fetches data and returns a list of <typeparamref name=”T”> ”/>”> .</summary>  
/// <param name="string">query to execute</param>  
  
public List<T> FetchData<T>(string query) {  
 ...  
}

* 1. Verarbeiten der Dokumentationsdatei

Für jedes Element im Quellcode, das mit einem Dokumentationskommentar markiert ist, erstellt der Dokumentations-Generator eine ID-Zeichenfolge. Durch diese ID-Zeichenfolge wird ein Quellcodeelement eindeutig identifiziert. Anhand einer ID-Zeichenfolge kann in einem Dokumentationsviewer das entsprechende Metadaten-/Reflektionselement identifiziert werden, auf das sich die Dokumentation bezieht.

Die Dokumentationsdatei ist keine hierarchische Darstellung des Quellcodes. Es handelt sich vielmehr um eine unstrukturierte Liste, in der für jedes Element eine ID-Zeichenfolge generiert wird.

* + 1. ID-Zeichenfolgenformat

Die folgenden Regeln werden vom Dokumentationsgenerator beim Generieren der ID-Zeichenfolgen beachtet:

* Die Zeichenfolge darf keine Leerräume enthalten.
* Der erste Teil der Zeichenfolge kennzeichnet die Art des zu dokumentierenden Members durch ein einzelnes Zeichen, gefolgt von einem Doppelpunkt. Die folgenden Memberarten sind definiert:

|  |  |
| --- | --- |
| **Zeichen** | **Beschreibung** |
| E | Ereignis |
| F | Feld |
| M | Methode (einschließlich Konstruktoren, Destruktoren und Operatoren) |
| N | Namespace |
| P | Eigenschaft (einschließlich Indexern) |
| T | Typ (z. B. Klasse, Delegat, Enumeration, Schnittstelle und Struktur) |
| ! | Fehlerzeichenfolge; der Rest der Zeichenfolge liefert Informationen über den Fehler. Vom Dokumentationsgenerator werden z. B. Fehlerinformationen für Links erstellt, die nicht aufgelöst werden können. |

* Beim zweiten Teil der Zeichenfolge handelt es sich um den voll gekennzeichneten Namen eines Elements, beginnend mit dem Namespace-Stammverzeichnis. Der Name des Elements, die umschließenden Typen und der Namespace sind durch Punkte getrennt. Wenn der Name des Elements selbst Punkte enthält, werden sie durch das Zeichen # (U+0023) ersetzt. (Es wird davon ausgegangen, dass kein Element dieses Zeichen in seinem Namen aufweist.)
* Bei Methoden und Eigenschaften mit Argumenten folgt die in Klammern eingeschlossene Argumentliste. Sind keine Argumente vorhanden, werden keine Klammern verwendet. Die Argumente werden durch Trennzeichen voneinander getrennt. Die Codierung der einzelnen Argumente entspricht der einer CLI-Signatur:
* Argumente werden durch ihren Dokumentationsnamen dargestellt, der auf dem vollqualifizierten Namen basiert und folgendermaßen angepasst wird:

Argumenten, die einen generischen Typ darstellen, wird das Zeichen „’“ angehängt, gefolgt von der Anzahl der Typparameter.

Bei Argumenten mit dem Modifizierer out oder ref folgt auf den Typnamen ein @-Zeichen. Argumente, die anhand ihres Werts oder über params übergeben werden, haben keine spezielle Notation.

Arrayargumente werden als [ lowerbound : size , … , lowerbound : size ] dargestellt, wobei die Anzahl der Kommas durch Rang – 1 berechnet wird und die untere Grenze und die Größe jeder Dimension, sofern bekannt, dezimal dargestellt werden. Wenn keine untere Grenze oder Größe angegeben ist, wird sie ausgelassen. Wenn die untere Grenze und die Größe für eine bestimmte Dimension ausgelassen werden, wird der ":" ebenfalls ausgelassen. Verzweigte Arrays werden durch ein eckiges Klammerpaar "[]" für jede Ebene dargestellt.

Argumente mit einem anderen Zeigertyp als void werden mit einem \* hinter dem Typnamen dargestellt. Ein void-Zeiger wird durch den Typnamen System.Void dargestellt.

Argumente, die auf generische Typparameter verweisen, die in Typen definiert sind, werden mit dem Zeichen „`“ codiert, auf das der nullbasierte Index des Typparameters folgt.

Argumente, die in Methode definierte generische Typparameter verwenden, verwenden anstelle des für Typen verwendeten „`“ ein doppeltes schließendes Anführungszeichen „``“.

Argumente, die auf konstruierte generische Typen verweisen, werden mit dem generischen Typ codiert, auf den „{“, eine durch Trennzeichen getrennte Liste von Typargumenten und „}“ folgt.

* + 1. ID-Zeichenfolgenbeispiele

In den folgenden Beispielen wird jeweils ein C#-Codefragment zusammen mit der ID-Zeichenfolge dargestellt, die aus den einzelnen Quellcodeelementen erstellt wird, die Dokumentationskommentare unterstützen:

* Typen werden anhand der vollqualifizierten Namen dargestellt, die um generische Informationen erweitert werden.

enum Color { Red, Blue, Green }

namespace Acme  
{  
 interface IProcess {...}

struct ValueType {...}

class Widget: IProcess  
 {  
 public class NestedClass {...}

public interface IMenuItem {...}

public delegate void Del(int i);

public enum Direction { North, South, East, West }  
 }

class MyList<T>  
 {  
 class Helper<U,V> {...}  
 }  
}

"T:Color"  
"T:Acme.IProcess"  
"T:Acme.ValueType"  
"T:Acme.Widget"  
"T:Acme.Widget.NestedClass"  
"T:Acme.Widget.IMenuItem"  
"T:Acme.Widget.Del"  
"T:Acme.Widget.Direction"  
”T:Acme.MyList`1”  
”T:Acme.MyList`1.Helper`2”

* Felder werden anhand der vollqualifizierten Namen dargestellt:

namespace Acme  
{  
 struct ValueType  
 {  
 private int total;  
 }

class Widget: IProcess  
 {  
 public class NestedClass  
 {  
 private int value;  
 }

private string message;  
 private static Color defaultColor;  
 private const double PI = 3.14159;  
 protected readonly double monthlyAverage;  
 private long[] array1;  
 private Widget[,] array2;  
 private unsafe int \*pCount;  
 private unsafe float \*\*ppValues;  
 }  
}

"F:Acme.ValueType.total"  
"F:Acme.Widget.NestedClass.value"  
"F:Acme.Widget.message"  
"F:Acme.Widget.defaultColor"  
"F:Acme.Widget.PI"  
"F:Acme.Widget.monthlyAverage"  
"F:Acme.Widget.array1"  
"F:Acme.Widget.array2"  
"F:Acme.Widget.pCount"  
"F:Acme.Widget.ppValues"

* Konstruktoren

namespace Acme  
{  
 class Widget: IProcess  
 {  
 static Widget() {...}

public Widget() {...}

public Widget(string s) {...}  
 }  
}

"M:Acme.Widget.#cctor"  
"M:Acme.Widget.#ctor"  
"M:Acme.Widget.#ctor(System.String)"

* Destruktoren

namespace Acme  
{  
 class Widget: IProcess  
 {  
 ~Widget() {...}  
 }  
}

"M:Acme.Widget.Finalize"

* Methoden

namespace Acme  
{  
 struct ValueType  
 {  
 public void M(int i) {...}  
 }

class Widget: IProcess  
 {  
 public class NestedClass  
 {  
 public void M(int i) {...}  
 }

public static void M0() {...}  
 public void M1(char c, out float f, ref ValueType v) {...}  
 public void M2(short[] x1, int[,] x2, long[][] x3) {...}  
 public void M3(long[][] x3, Widget[][,,] x4) {...}  
 public unsafe void M4(char \*pc, Color \*\*pf) {...}  
 public unsafe void M5(void \*pv, double \*[][,] pd) {...}  
 public void M6(int i, params object[] args) {...}  
 }

class MyList<T>  
 {  
 public void Test(T t) { }  
 }

class UseList  
 {  
 public void Process(MyList<int> list) { }  
 public MyList<T> GetValues<T>(T inputValue) { return null; }  
 }  
}

"M:Acme.ValueType.M(System.Int32)"  
"M:Acme.Widget.NestedClass.M(System.Int32)"  
"M:Acme.Widget.M0"  
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"  
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"  
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"  
"M:Acme.Widget.M4(System.Char\*,Color\*\*)"  
"M:Acme.Widget.M5(System.Void\*,System.Double\*[0:,0:][])"  
"M:Acme.Widget.M6(System.Int32,System.Object[])"  
”M:Acme.MyList`1.Test(`0)”  
”M:Acme.UseList.Process(Acme.MyList{System.Int32})”  
”M:Acme.UseList.GetValues``(``0)”

* Eigenschaften und Indexer

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public int Width { get {...} set {...} }  
 public int this[int i] { get {...} set {...} }  
 public int this[string s, int i] { get {...} set {...} }  
 }  
}

"P:Acme.Widget.Width"  
"P:Acme.Widget.Item(System.Int32)"  
"P:Acme.Widget.Item(System.String,System.Int32)"

* Ereignisse

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public event Del AnEvent;  
 }  
}

"E:Acme.Widget.AnEvent"

* Unäre Operatoren

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x) {...}  
 }  
}

"M:Acme.Widget.op\_UnaryPlus(Acme.Widget)"

Die vollständigen Funktionsnamen für unäre Operatoren lauten wie folgt: op\_UnaryPlus, op\_UnaryNegation, op\_LogicalNot, op\_OnesComplement, op\_Increment, op\_Decrement, op\_True und op\_False.

* Binäre Operatoren

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static Widget operator+(Widget x1, Widget x2) {...}  
 }  
}

"M:Acme.Widget.op\_Addition(Acme.Widget,Acme.Widget)"

Die vollständigen Funktionsnamen für binäre Operatoren lauten wie folgt: op\_Addition, op\_Subtraction, op\_Multiply, op\_Division, op\_Modulus, op\_BitwiseAnd, op\_BitwiseOr, op\_ExclusiveOr, op\_LeftShift, op\_RightShift, op\_Equality, op\_Inequality, op\_LessThan, op\_LessThanOrEqual, op\_GreaterThan und op\_GreaterThanOrEqual.

* Konvertierungsoperatoren haben ein nachgestelltes "~", auf das der Rückgabetyp folgt.

namespace Acme  
{  
 class Widget: IProcess  
 {  
 public static explicit operator int(Widget x) {...}  
 public static implicit operator long(Widget x) {...}  
 }  
}

"M:Acme.Widget.op\_Explicit(Acme.Widget)~System.Int32"  
"M:Acme.Widget.op\_Implicit(Acme.Widget)~System.Int64"

* 1. Ein Beispiel
     1. C#-Quellcode

Das folgende Beispiel zeigt den Quellcode einer Point-Klasse:

namespace Graphics  
{  
  
/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.  
/// </summary>  
public class Point   
{

/// <summary>Instance variable <c>x</c> represents the point's  
 /// x-coordinate.</summary>  
 private int x;

/// <summary>Instance variable <c>y</c> represents the point's  
 /// y-coordinate.</summary>  
 private int y;

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>  
 public int X  
 {  
 get { return x; }  
 set { x = value; }  
 }

/// <value>Property <c>Y</c> represents the point's y-coordinate.</value>  
 public int Y  
 {  
 get { return y; }  
 set { y = value; }  
 }

/// <summary>This constructor initializes the new Point to  
 /// (0,0).</summary>  
 public Point() : this(0,0) {}

/// <summary>This constructor initializes the new Point to  
 /// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
 /// <param><c>xor</c> is the new Point's x-coordinate.</param>  
 /// <param><c>yor</c> is the new Point's y-coordinate.</param>  
 public Point(int xor, int yor) {  
 X = xor;  
 Y = yor;  
 }

/// <summary>This method changes the point's location to  
 /// the given coordinates.</summary>  
 /// <param><c>xor</c> is the new x-coordinate.</param>  
 /// <param><c>yor</c> is the new y-coordinate.</param>  
 /// <see cref="Translate"/>  
 public void Move(int xor, int yor) {  
 X = xor;  
 Y = yor;  
 }

/// <summary>This method changes the point's location by  
 /// the given x- and y-offsets.  
 /// <example>For example:  
 /// <code>  
 /// Point p = new Point(3,5);  
 /// p.Translate(-1,3);  
 /// </code>  
 /// results in <c>p</c>'s having the value (2,8).  
 /// </example>  
 /// </summary>  
 /// <param><c>xor</c> is the relative x-offset.</param>  
 /// <param><c>yor</c> is the relative y-offset.</param>  
 /// <see cref="Move"/>  
 public void Translate(int xor, int yor) {  
 X += xor;  
 Y += yor;  
 }

/// <summary>This method determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>o</c> is the object to be compared to the current object.  
 /// </param>  
 /// <returns>True if the Points have the same location and they have  
 /// the exact same type; otherwise, false.</returns>  
 /// <seealso cref="operator=="/>  
 /// <seealso cref="operator!="/>  
 public override bool Equals(object o) {  
 if (o == null) {  
 return false;  
 }

if (this == o) {  
 return true;  
 }

if (GetType() == o.GetType()) {  
 Point p = (Point)o;  
 return (X == p.X) && (Y == p.Y);  
 }  
 return false;  
 }

/// <summary>Report a point's location as a string.</summary>  
 /// <returns>A string representing a point's location, in the form (x,y),  
 /// without any leading, training, or embedded whitespace.</returns>  
 public override string ToString() {  
 return "(" + X + "," + Y + ")";  
 }

/// <summary>This operator determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>p1</c> is the first Point to be compared.</param>  
 /// <param><c>p2</c> is the second Point to be compared.</param>  
 /// <returns>True if the Points have the same location and they have  
 /// the exact same type; otherwise, false.</returns>  
 /// <seealso cref="Equals"/>  
 /// <seealso cref="operator!="/>  
 public static bool operator==(Point p1, Point p2) {  
 if ((object)p1 == null || (object)p2 == null) {  
 return false;  
 }  
  
 if (p1.GetType() == p2.GetType()) {  
 return (p1.X == p2.X) && (p1.Y == p2.Y);  
 }  
  
 return false;  
 }

/// <summary>This operator determines whether two Points have the same  
 /// location.</summary>  
 /// <param><c>p1</c> is the first Point to be compared.</param>  
 /// <param><c>p2</c> is the second Point to be compared.</param>  
 /// <returns>True if the Points do not have the same location and the  
 /// exact same type; otherwise, false.</returns>  
 /// <seealso cref="Equals"/>  
 /// <seealso cref="operator=="/>  
 public static bool operator!=(Point p1, Point p2) {  
 return !(p1 == p2);  
 }

/// <summary>This is the entry point of the Point class testing  
 /// program.  
 /// <para>This program tests each method and operator, and  
 /// is intended to be run after any non-trvial maintenance has  
 /// been performed on the Point class.</para></summary>  
 public static void Main() {  
 // class test code goes here  
 }  
}  
}

* + 1. Resultierende XML

Im Folgenden wird die Ausgabe dargestellt, die von einem Dokumentations-Generator aus dem oben angegebenen Quellcode für die Point-Klasse erzeugt wird:

<?xml version="1.0"?>  
<doc>  
 <assembly>  
 <name>Point</name>  
 </assembly>  
 <members>  
 <member name="T:Graphics.Point">  
 <summary>Class <c>Point</c> models a point in a two-dimensional  
 plane.  
 </summary>  
 </member>

<member name="F:Graphics.Point.x">  
 <summary>Instance variable <c>x</c> represents the point's  
 x-coordinate.</summary>  
 </member>

<member name="F:Graphics.Point.y">  
 <summary>Instance variable <c>y</c> represents the point's  
 y-coordinate.</summary>  
 </member>

<member name="M:Graphics.Point.#ctor">  
 <summary>This constructor initializes the new Point to  
 (0,0).</summary>  
 </member>

<member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">  
 <summary>This constructor initializes the new Point to  
 (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
 <param><c>xor</c> is the new Point's x-coordinate.</param>  
 <param><c>yor</c> is the new Point's y-coordinate.</param>  
 </member>

<member name="M:Graphics.Point.Move(System.Int32,System.Int32)">  
 <summary>This method changes the point's location to  
 the given coordinates.</summary>  
 <param><c>xor</c> is the new x-coordinate.</param>  
 <param><c>yor</c> is the new y-coordinate.</param>  
 <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>  
 </member>

<member  
 name="M:Graphics.Point.Translate(System.Int32,System.Int32)">  
 <summary>This method changes the point's location by  
 the given x- and y-offsets.  
 <example>For example:  
 <code>  
 Point p = new Point(3,5);  
 p.Translate(-1,3);  
 </code>  
 results in <c>p</c>'s having the value (2,8).  
 </example>  
 </summary>  
 <param><c>xor</c> is the relative x-offset.</param>  
 <param><c>yor</c> is the relative y-offset.</param>  
 <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>  
 </member>

<member name="M:Graphics.Point.Equals(System.Object)">  
 <summary>This method determines whether two Points have the same  
 location.</summary>  
 <param><c>o</c> is the object to be compared to the current  
 object.  
 </param>  
 <returns>True if the Points have the same location and they have  
 the exact same type; otherwise, false.</returns>  
 <seealso  
 cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member name="M:Graphics.Point.ToString">  
 <summary>Report a point's location as a string.</summary>  
 <returns>A string representing a point's location, in the form  
 (x,y),  
 without any leading, training, or embedded whitespace.</returns>  
 </member>

<member  
 name="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same  
 location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points have the same location and they have  
 the exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member  
 name="M:Graphics.Point.op\_Inequality(Graphics.Point,Graphics.Point)">  
 <summary>This operator determines whether two Points have the  
 same  
 location.</summary>  
 <param><c>p1</c> is the first Point to be compared.</param>  
 <param><c>p2</c> is the second Point to be compared.</param>  
 <returns>True if the Points do not have the same location and  
 the  
 exact same type; otherwise, false.</returns>  
 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>  
 <seealso  
 cref="M:Graphics.Point.op\_Equality(Graphics.Point,Graphics.Point)"/>  
 </member>

<member name="M:Graphics.Point.Main">  
 <summary>This is the entry point of the Point class testing  
 program.  
 <para>This program tests each method and operator, and  
 is intended to be run after any non-trvial maintenance has  
 been performed on the Point class.</para></summary>  
 </member>

<member name="P:Graphics.Point.X">  
 <value>Property <c>X</c> represents the point's  
 x-coordinate.</value>  
 </member>

<member name="P:Graphics.Point.Y">  
 <value>Property <c>Y</c> represents the point's  
 y-coordinate.</value>  
 </member>  
 </members>  
</doc>

1. Grammatik

Dieser Anhang enthält Zusammenfassungen der im Hauptdokument beschriebenen lexikalischen und syntaktischen Grammatiken sowie der Grammatikerweiterungen für nicht gesicherten Code. Die Grammatikproduktionen werden hier in derselben Reihenfolge aufgeführt wie im Hauptdokument.

* 1. Lexikalische Grammatik

input:  
input-sectionopt

input-section:  
input-section-part  
input-section input-section-part

input-section-part:  
input-elementsopt new-line  
pp-directive

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

* + 1. Zeilenabschlusszeichen

new-line:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

* + 1. Kommentare

comment:  
single-line-comment  
delimited-comment

single-line-comment:  
// input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Next line character (U+0085)  
Line separator character (U+2028)  
Paragraph separator character (U+2029)

delimited-comment:  
/\* delimited-comment-textopt asterisks /

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
/  
asterisksopt not-slash-or-asterisk

asterisks:  
\*  
asterisks \*

not-slash-or-asterisk:  
Any Unicode character except / or \*

* + 1. Leerraum

whitespace:  
Any character with Unicode class Zs  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)

* + 1. Token

token:  
identifier  
keyword  
integer-literal  
real-literal  
character-literal  
string-literal  
operator-or-punctuator

* + 1. Escapesequenzen für Unicode-Zeichen

unicode-escape-sequence:  
\u hex-digit hex-digit hex-digit hex-digit  
\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

* + 1. Bezeichner

identifier:  
available-identifier  
@ identifier-or-keyword

available-identifier:  
An identifier-or-keyword that is not a keyword

identifier-or-keyword:  
identifier-start-character identifier-part-charactersopt

identifier-start-character:  
letter-character  
\_ (the underscore character U+005F)

identifier-part-characters:  
identifier-part-character  
identifier-part-characters identifier-part-character

identifier-part-character:  
letter-character  
decimal-digit-character  
connecting-character  
combining-character  
formatting-character

letter-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl   
A unicode-escape-sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:  
A Unicode character of classes Mn or Mc   
A unicode-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:  
A Unicode character of the class Nd   
A unicode-escape-sequence representing a character of the class Nd

connecting-character:   
A Unicode character of the class Pc  
A unicode-escape-sequence representing a character of the class Pc

formatting-character:   
A Unicode character of the class Cf  
A unicode-escape-sequence representing a character of the class Cf

* + 1. Schlüsselwörter

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

* + 1. Literale

literal:  
boolean-literal  
integer-literal  
real-literal  
character-literal  
string-literal  
null-literal

boolean-literal:  
true  
false

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits integer-type-suffixopt

decimal-digits:  
decimal-digit  
decimal-digits decimal-digit

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of  
U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:  
0x hex-digits integer-type-suffixopt  
0X hex-digits integer-type-suffixopt

hex-digits:  
hex-digit  
hex-digits hex-digit

hex-digit: one of  
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

real-literal:  
decimal-digits . decimal-digits exponent-partopt real-type-suffixopt  
. decimal-digits exponent-partopt real-type-suffixopt  
decimal-digits exponent-part real-type-suffixopt  
decimal-digits real-type-suffix

exponent-part:  
e signopt decimal-digits  
E signopt decimal-digits

sign: one of  
+ -

real-type-suffix: one of  
F f D d M m

character-literal:  
' character '

character:  
single-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-character:  
 Any character except ' (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of  
\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:  
\x hex-digit hex-digitopt hex-digitopt hex-digitopt

string-literal:  
regular-string-literal  
verbatim-string-literal

regular-string-literal:  
" regular-string-literal-charactersopt "

regular-string-literal-characters:  
regular-string-literal-character  
regular-string-literal-characters regular-string-literal-character

regular-string-literal-character:  
single-regular-string-literal-character  
simple-escape-sequence  
hexadecimal-escape-sequence  
unicode-escape-sequence

single-regular-string-literal-character:  
Any character except " (U+0022), \ (U+005C), and new-line-character

verbatim-string-literal:  
@" verbatim-string-literal-charactersopt "

verbatim-string-literal-characters:  
verbatim-string-literal-character  
verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:  
single-verbatim-string-literal-character  
quote-escape-sequence

single-verbatim-string-literal-character:  
any character except "

quote-escape-sequence:  
""

null-literal:  
null

* + 1. Operatoren und Markierungszeichen

operator-or-punctuator: one of  
{ } [ ] ( ) . , : ;  
+ - \* / % & | ^ ! ~  
= < > ? ?? :: ++ -- && ||  
-> == != <= >= += -= \*= /= %=  
&= |= ^= << <<= =>

right-shift:  
>|>

right-shift-assignment:  
>|>=

* + 1. Präprozessordirektiven

pp-directive:  
pp-declaration  
pp-conditional  
pp-line  
pp-diagnostic  
pp-region   
pp-pragma

conditional-symbol:  
Any identifier-or-keyword except true or false

pp-expression:  
whitespaceopt pp-or-expression whitespaceopt

pp-or-expression:  
pp-and-expression  
pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression:  
pp-equality-expression  
pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression

pp-equality-expression:  
pp-unary-expression  
pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression  
pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

pp-unary-expression:  
pp-primary-expression  
! whitespaceopt pp-unary-expression

pp-primary-expression:  
true  
false  
conditional-symbol  
( whitespaceopt pp-expression whitespaceopt )

pp-declaration:  
whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line  
whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line:  
whitespaceopt single-line-commentopt new-line

pp-conditional:  
pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif

pp-if-section:  
whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-sectionopt

pp-elif-sections:  
pp-elif-section  
pp-elif-sections pp-elif-section

pp-elif-section:  
whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-sectionopt

pp-else-section:  
whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

pp-endif:  
whitespaceopt # whitespaceopt endif pp-new-line

conditional-section:  
input-section  
skipped-section

skipped-section:  
skipped-section-part  
skipped-section skipped-section-part

skipped-section-part:  
skipped-charactersopt new-line  
pp-directive

skipped-characters:  
whitespaceopt not-number-sign input-charactersopt

not-number-sign:  
Any input-character except #

pp-diagnostic:  
whitespaceopt # whitespaceopt error pp-message  
whitespaceopt # whitespaceopt warning pp-message

pp-message:  
new-line  
whitespace input-charactersopt new-line

pp-region:  
pp-start-region conditional-sectionopt pp-end-region

pp-start-region:  
whitespaceopt # whitespaceopt region pp-message

pp-end-region:  
whitespaceopt # whitespaceopt endregion pp-message

pp-line:  
whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator:  
decimal-digits whitespace file-name   
decimal-digits  
default   
hidden

file-name:  
" file-name-characters "

file-name-characters:  
file-name-character  
file-name-characters file-name-character

file-name-character:  
Any input-character except "

pp-pragma:  
whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line

pragma-body:  
pragma-warning-body

pragma-warning-body:  
warning whitespace warning-action  
warning whitespace warning-action whitespace warning-list

warning-action:  
disable  
restore

warning-list:  
decimal-digits  
warning-list whitespaceopt , whitespaceopt decimal-digits

* 1. Syntaktische Grammatik
     1. Grundlegende Konzepte

namespace-name:  
namespace-or-type-name

type-name:  
namespace-or-type-name

namespace-or-type-name:  
identifier type-argument-listopt  
namespace-or-type-name . identifier type-argument-listoptqualified-alias-member

* + 1. Typen

type:  
value-type  
reference-type   
type-parameter

value-type:  
struct-type  
enum-type

struct-type:  
type-name  
simple-type   
nullable-type

simple-type:  
numeric-type  
bool

numeric-type:  
integral-type  
floating-point-type  
decimal

integral-type:  
sbyte  
byte  
short  
ushort  
int  
uint  
long  
ulong  
char

floating-point-type:  
float  
double

nullable-type:  
non-nullable-value-type ?

non-nullable-value-type:  
type

enum-type:  
type-name

reference-type:  
class-type  
interface-type  
array-type  
delegate-type

class-type:  
type-name  
object  
dynamic  
string

interface-type:  
type-name

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

delegate-type:  
type-name

type-argument-list:  
< type-arguments >

type-arguments:  
type-argument  
type-arguments , type-argument

type-argument:  
type

type-parameter:  
identifier

* + 1. Variablen

variable-reference:  
expression

* + 1. Ausdrücke

argument-list:  
argument  
argument-list , argument

argument:  
argument-nameopt argument-value

argument-name:  
identifier :

argument-value:  
expression  
ref variable-reference  
out variable-reference

primary-expression:   
primary-no-array-creation-expression  
array-creation-expression

primary-no-array-creation-expression:  
literal  
simple-name  
parenthesized-expression  
member-access  
invocation-expression  
element-access  
this-access  
base-access  
post-increment-expression  
post-decrement-expression  
object-creation-expression  
delegate-creation-expression  
anonymous-object-creation-expression  
typeof-expression  
 checked-expression  
unchecked-expression   
default-value-expression  
anonymous-method-expression

simple-name:  
identifier type-argument-listopt

parenthesized-expression:  
( expression )

member-access:  
primary-expression . identifier type-argument-listopt  
predefined-type . identifier type-argument-listopt  
qualified-alias-member . identifier

predefined-type: one of  
bool byte char decimal double float int long  
object sbyte short string uint ulong ushort

invocation-expression:  
primary-expression ( argument-listopt )

element-access:  
primary-no-array-creation-expression [ argument-list ]

this-access:  
this

base-access:  
base . identifier  
base [ argument-list ]

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression --

object-creation-expression:  
new type ( argument-listopt ) object-or-collection-initializeropt   
new type object-or-collection-initializer

object-or-collection-initializer:  
object-initializer  
collection-initializer

object-initializer:  
{ member-initializer-listopt }  
{ member-initializer-list , }

member-initializer-list:  
member-initializer  
member-initializer-list , member-initializer

member-initializer:  
identifier = initializer-value

initializer-value:  
expression  
object-or-collection-initializer

collection-initializer:  
{ element-initializer-list }  
{ element-initializer-list , }

element-initializer-list:  
element-initializer  
element-initializer-list , element-initializer

element-initializer:  
non-assignment-expression  
{ expression-list }

expression-list:  
expression  
expression-list , expression

array-creation-expression:  
new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt  
new array-type array-initializer   
new rank-specifier array-initializer

delegate-creation-expression:  
new delegate-type ( expression )

anonymous-object-creation-expression:  
new anonymous-object-initializer

anonymous-object-initializer:  
{ member-declarator-listopt }  
{ member-declarator-list , }

member-declarator-list:  
member-declarator  
member-declarator-list , member-declarator

member-declarator:  
simple-name  
member-access  
identifier = expression

typeof-expression:  
typeof ( type )  
typeof ( unbound-type-name )  
typeof ( void )

unbound-type-name:  
identifier generic-dimension-specifieropt  
identifier :: identifier generic-dimension-specifieropt  
unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:  
< commasopt >

commas:  
,  
commas ,

checked-expression:  
checked ( expression )

unchecked-expression:  
unchecked ( expression )

default-value-expression:  
default ( type )

unary-expression:  
primary-expression  
+ unary-expression  
- unary-expression  
! unary-expression  
~ unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression

pre-increment-expression:  
++ unary-expression

pre-decrement-expression:  
-- unary-expression

cast-expression:  
( type ) unary-expression

multiplicative-expression:  
unary-expression  
multiplicative-expression \* unary-expression  
multiplicative-expression / unary-expression  
multiplicative-expression % unary-expression

additive-expression:  
multiplicative-expression  
additive-expression + multiplicative-expression  
additive-expression – multiplicative-expression

shift-expression:  
additive-expression   
shift-expression << additive-expression  
shift-expression right-shift additive-expression

relational-expression:  
shift-expression  
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression  
relational-expression is type  
relational-expression as type

equality-expression:  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression

and-expression:  
equality-expression  
and-expression & equality-expression

exclusive-or-expression:  
and-expression  
exclusive-or-expression ^ and-expression

inclusive-or-expression:  
exclusive-or-expression  
inclusive-or-expression | exclusive-or-expression

conditional-and-expression:  
inclusive-or-expression  
conditional-and-expression && inclusive-or-expression

conditional-or-expression:  
conditional-and-expression  
conditional-or-expression || conditional-and-expression

null-coalescing-expression:  
conditional-or-expression  
conditional-or-expression ?? null-coalescing-expression

conditional-expression:  
null-coalescing-expression  
null-coalescing-expression ? expression : expression

lambda-expression:  
anonymous-function-signature => anonymous-function-body

anonymous-method-expression:  
delegate explicit-anonymous-function-signatureopt block

anonymous-function-signature:  
explicit-anonymous-function-signature   
implicit-anonymous-function-signature

explicit-anonymous-function-signature:  
( explicit-anonymous-function-parameter-list*opt* )

explicit-anonymous-function-parameter-list:  
explicit-anonymous-function-parameter  
explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

explicit-anonymous-function-parameter:  
anonymous-function-parameter-modifieropt type identifier

anonymous-function-parameter-modifier:   
ref  
out

implicit-anonymous-function-signature:  
( implicit-anonymous-function-parameter-list*opt* )  
implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:  
implicit-anonymous-function-parameter  
implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:  
identifier

anonymous-function-body:  
expression  
block

query-expression:  
from-clause query-body

from-clause:  
from typeopt identifier in expression

query-body:  
query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:  
query-body-clause  
query-body-clauses query-body-clause

query-body-clause:  
from-clause  
let-clause  
where-clause  
join-clause  
join-into-clause  
orderby-clause

let-clause:  
let identifier = expression

where-clause:  
where boolean-expression

join-clause:  
join typeopt identifier in expression on expression equals expression

join-into-clause:  
join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:  
orderby orderings

orderings:  
ordering  
orderings , ordering

ordering:  
expression ordering-directionopt

ordering-direction:  
ascending  
descending

select-or-group-clause:  
select-clause  
group-clause

select-clause:  
select expression

group-clause:  
group expression by expression

query-continuation:  
into identifier query-body

assignment:  
unary-expression assignment-operator expression

assignment-operator:  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment

expression:   
non-assignment-expression  
assignment

non-assignment-expression:  
conditional-expression  
lambda-expression  
query-expression

constant-expression:  
expression

boolean-expression:  
expression

* + 1. Anweisungen

statement:  
labeled-statement  
declaration-statement  
embedded-statement

embedded-statement:  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement   
yield-statement

block:  
{ statement-listopt }

statement-list:  
statement  
statement-list statement

empty-statement:  
;

labeled-statement:  
identifier : statement

declaration-statement:  
local-variable-declaration ;  
local-constant-declaration ;

local-variable-declaration:  
local-variable-type local-variable-declarators

local-variable-type:  
type  
var

local-variable-declarators:  
local-variable-declarator  
local-variable-declarators , local-variable-declarator

local-variable-declarator:  
identifier  
identifier = local-variable-initializer

local-variable-initializer:  
expression  
array-initializer

local-constant-declaration:  
const type constant-declarators

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

expression-statement:  
statement-expression ;

statement-expression:  
invocation-expression  
object-creation-expression  
assignment  
post-increment-expression  
post-decrement-expression  
pre-increment-expression  
pre-decrement-expression

selection-statement:  
if-statement  
switch-statement

if-statement:  
if ( boolean-expression ) embedded-statement  
if ( boolean-expression ) embedded-statement else embedded-statement

switch-statement:  
switch ( expression ) switch-block

switch-block:  
{ switch-sectionsopt }

switch-sections:  
switch-section  
switch-sections switch-section

switch-section:  
switch-labels statement-list

switch-labels:  
switch-label  
switch-labels switch-label

switch-label:  
case constant-expression :  
default :

iteration-statement:  
while-statement  
do-statement  
for-statement  
foreach-statement

while-statement:  
while ( boolean-expression ) embedded-statement

do-statement:  
do embedded-statement while ( boolean-expression ) ;

for-statement:  
for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:  
local-variable-declaration  
statement-expression-list

for-condition:  
boolean-expression

for-iterator:  
statement-expression-list

statement-expression-list:  
statement-expression  
statement-expression-list , statement-expression

foreach-statement:  
foreach ( local-variable-type identifier in expression ) embedded-statement

jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement

break-statement:  
break ;

continue-statement:  
continue ;

goto-statement:  
goto identifier ;  
goto case constant-expression ;  
goto default ;

return-statement:  
return expressionopt ;

throw-statement:  
throw expressionopt ;

try-statement:  
try block catch-clauses  
try block finally-clause  
try block catch-clauses finally-clause

catch-clauses:  
specific-catch-clauses general-catch-clauseopt  
specific-catch-clausesopt general-catch-clause

specific-catch-clauses:  
specific-catch-clause  
specific-catch-clauses specific-catch-clause

specific-catch-clause:  
catch ( class-type identifieropt ) block

general-catch-clause:  
catch block

finally-clause:  
finally block

checked-statement:  
checked block

unchecked-statement:  
unchecked block

lock-statement:  
lock ( expression ) embedded-statement

using-statement:  
using ( resource-acquisition ) embedded-statement

resource-acquisition:  
local-variable-declaration  
expression

yield-statement:  
yield return expression ;  
yield break ;

* + 1. Namespaces

compilation-unit:  
extern-alias-directivesopt using-directivesopt global-attributesopt  
 namespace-member-declarationsopt

namespace-declaration:  
namespace qualified-identifier namespace-body ;opt

qualified-identifier:  
identifier  
qualified-identifier . identifier

namespace-body:  
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

extern-alias-directives:  
extern-alias-directive  
extern-alias-directives extern-alias-directive

extern-alias-directive:  
extern alias identifier ;

using-directives:  
using-directive  
using-directives using-directive

using-directive:  
using-alias-directive  
using-namespace-directive

using-alias-directive:  
using identifier = namespace-or-type-name ;

using-namespace-directive:  
using namespace-name ;

namespace-member-declarations:  
namespace-member-declaration  
namespace-member-declarations namespace-member-declaration

namespace-member-declaration:  
namespace-declaration  
type-declaration

type-declaration:  
class-declaration  
struct-declaration  
interface-declaration  
enum-declaration  
delegate-declaration

qualified-alias-member:  
identifier :: identifier type-argument-listopt

* + 1. Klassen

class-declaration:  
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
 class-baseopt type-parameter-constraints-clausesopt class-body ;opt

class-modifiers:  
class-modifier  
class-modifiers class-modifier

class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed  
static

type-parameter-list:  
< type-parameters >

type-parameters:  
attributesopt type-parameter  
type-parameters , attributesopt type-parameter

type-parameter:  
identifier

class-base:  
: class-type  
: interface-type-list  
: class-type , interface-type-list

interface-type-list:  
interface-type  
interface-type-list , interface-type

type-parameter-constraints-clauses:  
type-parameter-constraints-clause  
type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:  
where type-parameter : type-parameter-constraints

type-parameter-constraints:  
primary-constraint  
secondary-constraints  
constructor-constraint  
primary-constraint , secondary-constraints  
primary-constraint , constructor-constraint  
secondary-constraints , constructor-constraint  
primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:  
class-type  
class  
struct

secondary-constraints:  
interface-type  
type-parameter  
secondary-constraints , interface-type  
secondary-constraints , type-parameter

constructor-constraint:  
new ( )

class-body:  
{ class-member-declarationsopt }

class-member-declarations:  
class-member-declaration  
class-member-declarations class-member-declaration

class-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
destructor-declaration  
static-constructor-declaration  
type-declaration

constant-declaration:  
attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers:  
constant-modifier  
constant-modifiers constant-modifier

constant-modifier:  
new  
public  
protected  
internal  
private

constant-declarators:  
constant-declarator  
constant-declarators , constant-declarator

constant-declarator:  
identifier = constant-expression

field-declaration:  
attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:  
field-modifier  
field-modifiers field-modifier

field-modifier:  
new  
public  
protected  
internal  
private  
static  
readonly  
volatile

variable-declarators:  
variable-declarator  
variable-declarators , variable-declarator

variable-declarator:  
identifier  
identifier = variable-initializer

variable-initializer:  
expression  
array-initializer

method-declaration:  
method-header method-body

method-header:  
attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers:  
method-modifier  
method-modifiers method-modifier

method-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

return-type:  
type  
void

member-name:  
identifier  
interface-type . identifier

method-body:  
block  
;

formal-parameter-list:  
fixed-parameters  
fixed-parameters , parameter-array  
parameter-array

fixed-parameters:  
fixed-parameter  
fixed-parameters , fixed-parameter

fixed-parameter:  
attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:  
= expression

parameter-modifier:  
ref  
out  
this

parameter-array:  
attributesopt params array-type identifier

property-declaration:  
attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:  
property-modifier  
property-modifiers property-modifier

property-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

member-name:  
identifier  
interface-type . identifier

accessor-declarations:  
get-accessor-declaration set-accessor-declarationopt  
set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:  
attributesopt accessor-modifieropt  get accessor-body

set-accessor-declaration:  
attributesopt accessor-modifieropt set accessor-body

accessor-modifier:  
protected  
internal  
private  
protected internal  
internal protected

accessor-body:  
block  
;

event-declaration:  
attributesopt event-modifiersopt event type variable-declarators ;  
attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:  
event-modifier  
event-modifiers event-modifier

event-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

event-accessor-declarations:  
add-accessor-declaration remove-accessor-declaration  
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:  
attributesopt add block

remove-accessor-declaration:  
attributesopt remove block

indexer-declaration:  
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:  
indexer-modifier  
indexer-modifiers indexer-modifier

indexer-modifier:  
new  
public  
protected  
internal  
private   
virtual  
sealed  
override  
abstract  
extern

indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

operator-declaration:  
attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:  
operator-modifier  
operator-modifiers operator-modifier

operator-modifier:  
public  
static  
extern

operator-declarator:  
unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

unary-operator-declarator:  
type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of  
+ - ! ~ ++ -- true false

binary-operator-declarator:  
type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:  
+  
-  
\*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
<=

conversion-operator-declarator:  
implicit operator type ( type identifier )  
explicit operator type ( type identifier )

operator-body:  
block  
;

constructor-declaration:  
attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:  
constructor-modifier  
constructor-modifiers constructor-modifier

constructor-modifier:  
public  
protected  
internal  
private  
extern

constructor-declarator:  
identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:  
: base ( argument-listopt )  
: this ( argument-listopt )

constructor-body:  
block  
;

static-constructor-declaration:  
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers:  
externopt static  
static externopt

static-constructor-body:  
block  
;

destructor-declaration:  
attributesopt externopt ~ identifier ( ) destructor-body

destructor-body:  
block  
;

* + 1. Strukturen

struct-declaration:  
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
 struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier

struct-modifier:  
new  
public  
protected  
internal  
private

struct-interfaces:  
: interface-type-list

struct-body:  
{ struct-member-declarationsopt }

struct-member-declarations:  
struct-member-declaration  
struct-member-declarations struct-member-declaration

struct-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
static-constructor-declaration  
type-declaration

* + 1. Arrays

array-type:  
non-array-type rank-specifiers

non-array-type:  
type

rank-specifiers:  
rank-specifier  
rank-specifiers rank-specifier

rank-specifier:  
[ dim-separatorsopt ]

dim-separators:  
,  
dim-separators ,

array-initializer:  
{ variable-initializer-listopt }  
{ variable-initializer-list , }

variable-initializer-list:  
variable-initializer  
variable-initializer-list , variable-initializer

variable-initializer:  
expression  
array-initializer

* + 1. Schnittstellen

interface-declaration:  
attributesopt interface-modifiersopt partialopt interface   
 identifier variant-type-parameter-listopt interface-baseopt  
 type-parameter-constraints-clausesopt interface-body ;opt

interface-modifiers:  
interface-modifier  
interface-modifiers interface-modifier

interface-modifier:  
new  
public  
protected  
internal  
private

variant-type-parameter-list:  
< variant-type-parameters >

variant-type-parameters:  
attributesopt variance-annotationopt  type-parameter  
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:  
in  
out

interface-base:  
: interface-type-list

interface-body:  
{ interface-member-declarationsopt }

interface-member-declarations:  
interface-member-declaration  
interface-member-declarations interface-member-declaration

interface-member-declaration:  
interface-method-declaration  
interface-property-declaration  
interface-event-declaration  
interface-indexer-declaration

interface-method-declaration:  
attributesopt newopt return-type identifier type-parameter-list  
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

interface-property-declaration:  
attributesopt newopt type identifier { interface-accessors }

interface-accessors:  
attributesopt get ;  
attributesopt set ;  
attributesopt get ; attributesopt set ;  
attributesopt set ; attributesopt get ;

interface-event-declaration:  
attributesopt newopt event type identifier ;

interface-indexer-declaration:  
attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

* + 1. Enumerationen

enum-declaration:  
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:  
: integral-type

enum-body:  
{ enum-member-declarationsopt }  
{ enum-member-declarations , }

enum-modifiers:  
enum-modifier  
enum-modifiers enum-modifier

enum-modifier:  
new  
public  
protected  
internal  
private

enum-member-declarations:  
enum-member-declaration  
enum-member-declarations , enum-member-declaration

enum-member-declaration:  
attributesopt identifier  
attributesopt identifier = constant-expression

* + 1. Delegaten

delegate-declaration:  
attributesopt delegate-modifiersopt delegate return-type   
 identifier variant-type-parameter-listopt   
 ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:  
delegate-modifier  
delegate-modifiers delegate-modifier

delegate-modifier:  
new  
public  
protected  
internal  
private

* + 1. Attribute

global-attributes:  
global-attribute-sections

global-attribute-sections:  
global-attribute-section  
global-attribute-sections global-attribute-section

global-attribute-section:  
[ global-attribute-target-specifier attribute-list ]  
[ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:  
global-attribute-target :

global-attribute-target:  
assembly  
module

attributes:  
attribute-sections

attribute-sections:  
attribute-section  
attribute-sections attribute-section

attribute-section:  
[ attribute-target-specifieropt attribute-list ]  
[ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:  
attribute-target :

attribute-target:  
field  
event  
method  
param  
property  
return  
type

attribute-list:  
attribute  
attribute-list , attribute

attribute:  
attribute-name attribute-argumentsopt

attribute-name:  
 type-name

attribute-arguments:  
( positional-argument-listopt )  
( positional-argument-list , named-argument-list )  
( named-argument-list )

positional-argument-list:  
positional-argument  
positional-argument-list , positional-argument

positional-argument:  
argument-nameopt attribute-argument-expression

named-argument-list:  
named-argument  
named-argument-list , named-argument

named-argument:  
identifier = attribute-argument-expression

attribute-argument-expression:  
expression

* 1. Grammatische Erweiterungen für unsicheren Code

class-modifier:  
...  
unsafe

struct-modifier:  
...  
unsafe

interface-modifier:  
...  
unsafe

delegate-modifier:  
...  
unsafe

field-modifier:  
...  
unsafe

method-modifier:  
...  
unsafe

property-modifier:  
...  
unsafe

event-modifier:  
...  
unsafe

indexer-modifier:  
...  
unsafe

operator-modifier:  
...  
unsafe

constructor-modifier:  
...  
unsafe

destructor-declaration:  
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body  
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body

static-constructor-modifiers:  
externopt unsafeopt static  
unsafeopt externopt static  
externopt static unsafeopt   
unsafeopt static externopt  
static externopt unsafeopt  
static unsafeopt externopt

embedded-statement:  
...  
unsafe-statement   
fixed-statement

unsafe-statement:  
unsafe block

type:  
...  
pointer-type

pointer-type:  
unmanaged-type \*  
void \*

unmanaged-type:  
type

primary-no-array-creation-expression:  
...  
pointer-member-access  
pointer-element-access  
sizeof-expression

unary-expression:  
...  
pointer-indirection-expression  
addressof-expression

pointer-indirection-expression:  
\* unary-expression

pointer-member-access:  
primary-expression -> identifier type-argument-listopt

pointer-element-access:  
primary-no-array-creation-expression [ expression ]

addressof-expression:  
& unary-expression

sizeof-expression:  
sizeof ( unmanaged-type )

fixed-statement:  
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement

fixed-pointer-declarators:  
fixed-pointer-declarator  
fixed-pointer-declarators , fixed-pointer-declarator

fixed-pointer-declarator:  
identifier = fixed-pointer-initializer

fixed-pointer-initializer:  
& variable-reference  
expression

struct-member-declaration:  
…  
fixed-size-buffer-declaration

fixed-size-buffer-declaration:  
attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type  
 fixed-size-buffer-declarators ;

fixed-size-buffer-modifiers:  
fixed-size-buffer-modifier  
fixed-size-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifier:  
new  
public  
protected  
internal  
private  
unsafe

buffer-element-type:  
type

fixed-size-buffer-declarators:  
fixed-size-buffer-declarator  
fixed-size-buffer-declarator fixed-size-buffer-declarators

fixed-size-buffer-declarator:  
identifier [ constant-expression ]

local-variable-initializer:  
…  
stackalloc-initializer

stackalloc-initializer:  
stackalloc unmanaged-type [ expression ]

1. Referenzen

Unicode Consortium. The Unicode Standard, Version 3.0. Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE-Standard 754-1985. Erhältlich unter <http://www.ieee.org>.

ISO/IEC. C++. ANSI/ISO/IEC 14882:1998.